NAVARA

Customer Verification Engine

Automated Customer Verification for a Peer-to-Peer Lending Platform

M.R. Comans
O.N. de Haas
D.A.J. Oudejans
E. de Smidt

TU Delft

Customer Verification Engine

Automated Customer Verification for a Peer-to-Peer Lending Platform

by

M.R. Comans
O.N. de Haas
D.A.J. Oudejans
E. de Smidt

to obtain the degree of Bachelor of Science

at the Delft University of Technology

Project duration:	April 23, 2019 – June 25, 2019	
Thesis committee:	M. Finavaro Aniche, PhD	TU Delft, coach
	H. Wang, PhD	TU Delft, bachelor project coordinator
	O.W. Visser, MSc	TU Delft, bachelor project coordinator
	Y.J. Candel, MSc	NAVARA, client

Preface

With this document we present our final report for the Bachelor End Project, which marks the final deliverable for the curriculum Bachelor of Science in Computer Science and Engineering at Delft University of Technology. The report includes information on the development process of the Customer Verification Engine for a peer to peer lending platform developed by NAVARA. The report covers the research that has been done and the design choices that have been made over the course of this project.

This project has been a very educational experience, and it would not have been possible without the following people: First of all, we would like to sincerely thank our colleagues from NAVARA, as we thoroughly enjoyed working with them over the course of this project. We are thankful to them for making this project possible and for fulfilling our needs. Secondly, we would like to thank our client and mentor from NAVARA, Yves Candel, for giving us the opportunity to see what it is like to work in a professional setting, while always being a helpful mentor. Lastly, we would like to thank our coach from Delft University of Technology, Maurício Aniche, for his advice and feedback on the educational aspect of this project.

Martijn Comans
Olav de Haas
Daan Oudejans
Emiel de Smidt

Delft, June 2019

Summary

There is a lot involved in providing a loan as a company, mostly in terms of legalities and risk management. As a lender it is important to have a clear record of the customers applying for a loan, as this helps assessing the risk that comes with providing a loan. Furthermore, it is required by law to know who it is that you are providing a loan to. To achieve this, loan providers gather a variety of personal and financial information. The gathering of such information has traditionally been a time consuming practice, both for the customer and the lender. The customer is required to manually find and submit information, and in turn the lender has to verify that the received information is not fraudulent or incorrect. If collection of personal information, payrolls and credits could be done in an automated way, both the customer and the lender will benefit greatly.

We have designed and developed the Customer Verification Engine, the CVE, in order to solve this time consuming process of collecting and submitting documents. The CVE is capable of cleverly combining several external data sources, creating a clear record of the customer. While previously the customer had to manually provide a large set of documents, it is now done at the push of a button. Furthermore, by having a system where the information is retrieved, rather than provided by the customer, the verification becomes significantly more reliable, as there is little to no room for the customer to provide fraudulent information.

The CVE is a robust and scalable system that is capable of handling unexpected behavior both in terms of input and connection to external sources. An extensive test suite verifies correct behavior of the CVE under both normal and unexpected circumstances. The information gathered by the CVE will be relied upon to determine whether or not a customer is eligible for a loan. As the CVE will be continued to be worked upon, we have put effort into making it extendable for future developers. Using the extensive documentation and the modularity of the system, it should be straightforward for future developers to add new integrations with external parties to the CVE.

Contents

1

Introduction

Lending money as a for-profit company to consumers is not without risks. The simple definition of credit risk is the risk that a debtor defaults on a loan because they are failing to make payments [1]. The company is ultimately responsible for these risks. In order to cover the risks, the company will raise the interest on the loans it provides. To keep this risk as small as possible, various measures can be taken by companies, such as gathering information about the customer and performing credit checks. The more a credit provider can trust a customer to repay their loan, the lower the interest rates can be. In order to limit these risks, apart from the measures mentioned above, we can attempt to use information retrieved from the customer's paycheck.

Gathering these kinds of information is traditionally done using a lot of manual work: the customer has to gather all the required documents and information and the lender has to verify these documents. An example of such a document could be a copy of a paycheck, in order to prove that the consumer requesting a loan has a certain amount of income. The lender also has to take into account that these documents might be forged or tampered with. This process is very time consuming, error prone, and prone to fraud, which makes it expensive for the lender and in turn for the customer. Many information aggregations can be automated to speed up the process, such as identity verification and gathering information about someone's financial situation. In the end when all the information is available, the customer can be verified for a loan. Some of this information is even mandatory to keep track of, according to Dutch law [2]. Not only will this result in lower interest rates, but potentially also in higher customer satisfaction. Doing most of the work for the customer in a matter of minutes will greatly simplify the application process.

In order to solve the problems of slow verifications and possibly fraudulent applications in an efficient manner, we have developed the Customer Verification Engine (CVE). In the final product, the CVE will be responsible for determining whether or not a customer is eligible for a loan. Our goal was to build a solid foundation for the CVE for future developers to build upon. The CVE achieves this by analyzing information retrieved from several external sources, checking if all preconditions are met. Furthermore, it gathers all the personal information that is required by Dutch law, to have a clear record of the customer.

The CVE has been designed with security and robustness to malicious or erroneous input in mind, providing reliable results at all times. Using modern services and tools, we have ensured that the system is highly scalable and secure. With an extensive test suite, we validated correct behavior. Integration with many code analysis tools, like Better Code Hub, enforced us to write high quality and maintainable code.

In this report we will provide an overview of the functioning of the CVE in chapter 2, followed by an in-depth implementation analysis of different components in chapter 3. Furthermore, we will cover the process of developing the software in chapter 4 and the measures we took to ensure high quality in chapter 5. We will reflect on the feedback from our colleagues on our product and the process in chapter 6. Finally, we will conclude and discuss our process in chapter 7 and chapter 8.

2 Approach

The Customer Verification Engine is a major component within the back end for the peer-to-peer lending platform. Providing a seamless experience to the customer is of high importance, which means that the verification procedure has to be as automated as possible while also being transparent about how the result was generated. Moreover, to protect the lender from fraud and risks, the verification has to be reliable as well. Lastly, the verification has to comply with Dutch law, which mandates performing a credit check with the Bureau Krediet Registratie (BKR) [3]. In this chapter, we discuss how we designed the different parts of the CVE to work together in order to perform the verification process of a customer.

2.1. Basic customer information

Naturally, we need basic personal information of a customer. This starts with an e-mail address when the customer first creates their account. There are two ways in which we can get the customer's basic personal information. Obviously, one way of doing this is getting the customer to fill in a form. This is easy to implement and to process, but is prone to mistakes by the customer and takes some time. Moreover, there is a risk that the customer enters information that is not their own.

Another solution is to use iDIN [4]. The platform as a whole uses an external identity provider for managing user accounts. The identity provider is also able to integrate iDIN. iDIN is an identity verification initiative by Dutch banks. Since banks already have their customer's basic personal information on file and verified, they can provide this information to other organizations requesting it (with authorization from the customer). This information is therefore very reliable, resistant to user error and guaranteed to belong to the customer. On top of that, the process is very quick, since the customer only has to authorize the sharing of their personal information through the familiar interface of their bank. Lastly, iDIN is very transparent to the customer since it shows what data will be shared.

2.2. Paycheck verification

This basic personal information previously gathered is also registered with employers. To verify if a paycheck belongs to a person, the basic way to check if a paycheck belongs to a particular person would be to compare the personal information on the paycheck with the personal information on file. In case this paycheck is provided by the customer, the risk that this document has been tampered with is not negligible. A way to solve this would be to directly go to the employer, and access their payroll system to check if the person in question is present in their database. Although this requires a collaboration with employers, this would solve two issues previously mentioned issues at the same time: the customer has to do less work, and the lender can be more certain that a customer is who they say they are. Moreover, the salary of the customer can be accessed, which means that the lender can help keep the customer from taking loans that they can not financially handle.

Luckily, most companies do not maintain their payroll on their own. Companies mostly use readily available software solutions for payroll management. Some of these software packages can be interfaced with through an Application Programming Interface (API). This enables us to write software modules that interact with different payroll software APIs, which would essentially cover multiple companies at the same time. Since more employers, which use different payroll software, might join the lending platform, it is important that this system is modular and allows for different implementations. The APIs can use different formats and endpoints. However, they should all act and be treated in the same way within the CVE. Therefore, it would be wise to implement a generic interface that can be used for any type of API and still provide the same information. This increases modularity, yet it might be difficult, since the APIs might not return the same information.

2.3. Credit check

The BKR registers loans with the person's name, address and date of birth [5]. This also matches the data that we already have from gathering the customer's basic personal information. Consequentially, we can perform a credit check at the same time as the paycheck verification. Through an API for the BKR, the CVE is able to automatically retrieve the information registered at the BKR on the customer. As previously mentioned, this step is mandatory according to Dutch law [3]. Other information that the BKR provides, include: the kind of loan, the amount and the first and last month of payment [5].

2.4. Bank account verification

Verifying the customer's International Bank Account Number (IBAN) is an important step in the verification process. This is important for two main reasons, the first one being fraud prevention. A customer should be unable to specify a bank account that is not their own. The second reason is the prevention of mistakes. A simple typographical error could result in significant problems, especially since it concerns money. We can approach the verification of a bank account in two different ways.

The first way of verifying a bank account could be to retrieve it from the paycheck of the customer as well. To automate their payroll, companies enter the bank accounts of their employees in the payroll software. The software can then generate a list of transactions that should be performed in order to pay the employees. Within the payroll software, we could check to which bank account the customer's salary is transferred. An additional benefit of this approach is that it can be performed at the same time as when verifying the other paycheck information.

Another way of verifying a customer's bank account could be to let the customer make a small payment. Since we are working on a product for the Dutch market, the obvious solution would be to implement iDeal [6]. iDeal is a Dutch online payment method that allows a consumer to make payments through their own bank. After an iDeal payment has been made, the payee can see the IBAN of the customer, which makes it suitable for checking whether a bank account truly belongs to a person. Moreover, it prevents mistakes and fraud since the customer does not have to fill in their bank account details anymore. The disadvantage of this approach is that it requires some additional operations from the customer.

2.5. Customer approval

Of course, like any other website, the platform has to deal with legal requirements and documents, such as the terms of service or an authorization to withdraw money from a bank account. A responsibility of the CVE is to securely save records of the customer's agreements, including a timestamp of when the customer set the agreement. Furthermore, as it is crucial that the CVE operates on the correct data, it has been made a requirement that the customer is able to verify that the automatically retrieved personal information is correct. If for example the situation occurs when information retrieved from the payroll software is flagged as incorrect by the customer, it is their responsibility to contact their employer and correct the information. Only when this has been done, the customer can start the verification process again.

3 Design and Implementation

It is paramount that the implementation of the Customer Verification Engine is maintainable, scalable and secure since the platform is dealing with a customer's personal as well as financial data. In this chapter, we describe how we turned the approach described in chapter 2 into a system that also fulfills the above requirements. Designing the system involved continuous research about these topics and the tools that we have used, and continuous iterations to refine what we had already built.

3.1. Architecture

The CVE is built on a stack consisting of C#, .NET Core, Azure, Azure Functions and Azure SQL database. This choice was made because NAVARA already had a large amount of (positive) experience with Azure. Furthermore, Azure allows us to run our system and store data on servers that are located in The Netherlands. Our supervisor advised us to look into using Azure Functions, a serverless approach to writing and executing software. Serverless means that the developer does not have to specify *how* the software is run. The developer just provides *what* needs to be run, i.e. a function, and specifies when the function should be triggered. We decided to use Azure Functions based on a number of benefits:

- Serverless means that we could focus on programming instead of managing the environment.
- Functions are scalable: Azure can automatically start new Function instances and do load balancing. This is very useful for making our software future-proof.
- Functions can be triggered in several ways, including:

  HTTP Trigger is useful for writing our API endpoints for the CVE.

  Timed Trigger is useful for tasks that need to be run in the background on an interval, such as keeping our database up-to-date.

  Service Bus Trigger is useful in combination with a Service Bus. We use Azure Service Buses as a queue for Functions to start background jobs, such as the retrieval of salary data. This architecture is described in more detail in subsection 3.6.1.

We decided to use .NET Core in C# since it nicely integrates with other Azure services and Azure SQL Databases. Moreover, we all liked the challenge of learning a new language and new technologies to broaden our knowledge. C# syntax is not far off from Java syntax, so learning C# was relatively easy, which made the decision to use this development stack significantly easier.

The CVE is split into three components: the CVE library, the Data Access Layer (DAL) and the web API. We made the decision to separate the library from the API so that the library can be reused with other back ends or web APIs. It can also be easily replaced by another library since it is loosely coupled with the web API. The data access layer is used for database models.

3.2. Web API

The API is developed mostly following the Representational State Transfer (REST) architectural style [7, Chapter 5]. Our API is stateless, which means that every request to the server from the client contains all the information necessary to process the request. Therefore, the server does not have to store client context between requests. A stateless API is useful for clarity during development and in production since the full context of the request can be seen by looking at the request itself. Moreover, this is important from a scalability perspective, since we can spread a large number of requests over many different systems without them having to share request context.

The data format used by the requests and responses of the web API is in JavaScript Object Notation (JSON) format [8]. JSON is a widely used human-readable object notation in web-based applications. Many programming languages support JSON and JavaScript integrates with it nicely. We chose this format because it is easy to use for any type of program that integrates with our API.

3.3. Data Access Layer

The database is at the heart of the CVE and is a crucial element. We chose to use an Azure SQL Database since we wanted to use a relational model. We favored consistency and reliability over flexibility that you might gain when using a non-relational database, like NoSQL or a document store. The database is integrated into our code using a Data Access Layer (DAL) module. The DAL module is written with Entity Framework (EF) [9]. Entity Framework is an object-relational mapper for .NET. In other words, it allowed us to map database entries to .NET objects. We specified the database design in the DAL using a code-first approach with EF. Code-first means that you write the classes for the database tables in the codebase and EF generates tables and schemas from those classes. Furthermore, EF allowed us to migrate the database after design changes had been made while the database was in use. A migration can be executed on a database while the database contains data. EF stores a history of all migrations, so that migrations could easily be added and reverted, which provides consistency and safety. This turned out to be especially useful in a development environment and will be even more valuable once the CVE goes into production. Furthermore, during the first phase of the project, a data seeder has been created that seeds the database with data that could be used for testing purposes, ensuring that the database was usable at all times during development.

3.4. Payroll Software Integration

The Customer Verification Engine required many external sources for automation of the registration progress. One of these sources was Payroll software or Human Resources management software. The idea of the platform is to make lending accessible for as many employees as possible, so many HR software providers had to be integrated using their APIs. We implemented a generic interface for these APIs in the CVE library, which can be used anywhere in the CVE API as presented in Figure 3.1. This makes it easy to implement any type of API, as long as they supply the necessary information. Every API that implements the interface must accept a customer info request object, so they can be identified in the API. This object consists of:

- Last name
- Last name prefix
- Date of birth
- Postal code
- House number

Using all of these five fields reduces the chance for any collisions when searching for customers. The only collision that could occur would be twins, who live at the same address and work at the same company. Interface implementations must also return the found personal information, contracts and salary information of the customer.

The employers and their respective HR management software are all stored in the database. When a customer registers in the CVE, the customer also selects their employer. The CVE API then looks up the correct payroll

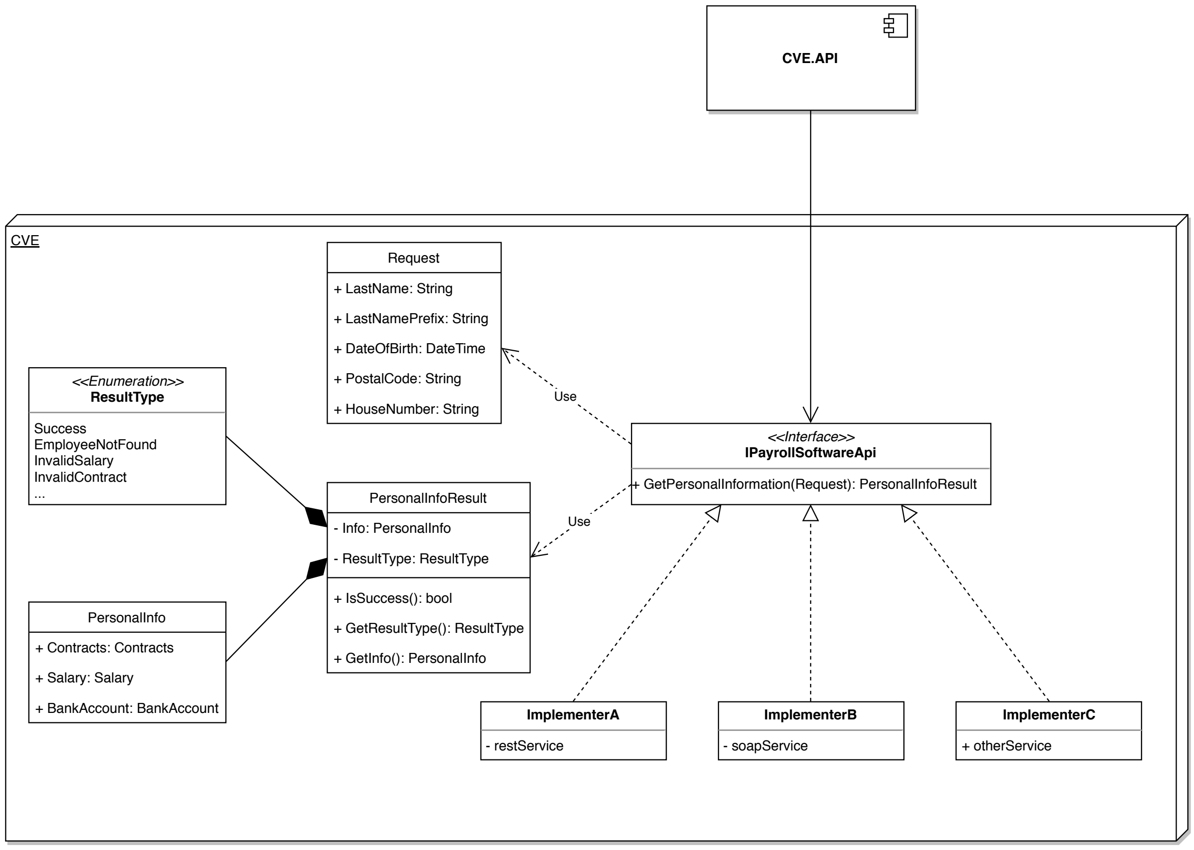

Figure 3.1: Class diagram of the interface for payroll software in UML. The interface defines one method, namely `GetPersonalInformationApi`. The interface utilizes `Request` as input and `PersonalInfo` as output. The three implementers are example implementers. They use for example a SOAP or REST API. The CVE web API is not concerned with which implementers there are. The CVE web API can be composed of multiple interfaces in the background jobs, which run at the same time.

software API by querying the database and acquire the interface implementation for further use.

We have experienced that payroll software solutions can differ greatly from each other. Not only the APIs can differ a lot, but also the content that the different solutions store and send. Some solutions would give responses in Dutch, while other responses would be in English. This is why we had to expand our implementations of the `IPayrollSoftwareApi` to make use of universal types that worked for every different implementation. For example, we implemented enums for types such as marital status, contract types and salary types. This allowed us to write conversion methods for the different implementations to convert the types from the particular payroll software into something that we could use universally.

3.5. Credits

The Customer Verification Engine had as a requirement that it retrieves all existing credits of a customer, as it is vital information for the product. In the Netherlands, all credits are registered at the BKR, offering both people as well as credit providers insight in outstanding debts of individuals. They offer multiple services, but the main interest for the CVE was getting insights into the financial status by retrieving the credits. The retrieval of the credits is initiated by starting a background job from subsection 3.6.1, as the operation can take quite a while and therefore it is favorable to run in the background. Once, completed, the credits are filtered and stored in the database.

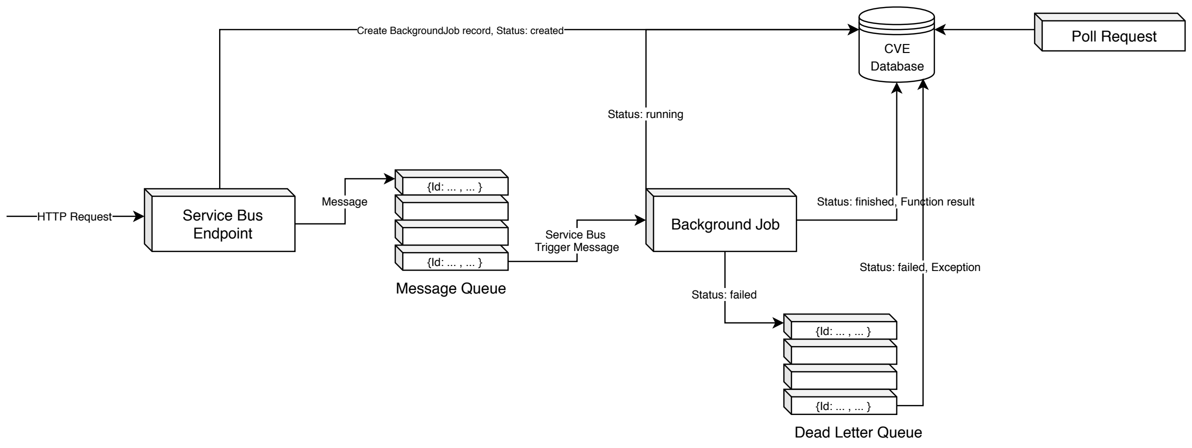

Figure 3.2: An high level diagram of the flow of the service bus.

3.6. Verification Architecture

In the Customer Verification Engine, some verification steps might take longer than one minute. In our case, such time-consuming processes were the steps where we relied on external parties for information retrieval. For example, when accessing payroll software via its API, it can take some time to get information for a single person depending on the number of employers a company has registered in its payroll software. These steps need to be run while the customer is waiting on the website. It would be unwise to leave an HTTP request open for more than half a minute, as most browsers would timeout the connection. We cannot run these jobs in an Azure function triggered by an HTTP trigger, as such a function can only run while the HTTP request is open, i.e. the function ends when a response is sent back to the client. Furthermore, we would like to be able to run such operations while the user is moving forward through the onboarding process. Lastly, it would be nice if these operations can be run at the same time, to save the customer some waiting time.

At first we considered WebSockets [10]. WebSocket is a network communication protocol providing two-way communication between client and server. So the client would start the verification step and keep a WebSocket open to receive results when verifications finish. Unfortunately, WebSockets is not the best solution for the CVE for two reasons. The first being that WebSocket connections are not stateless, this would be inconsistent with the rest of the API. The second reason was that Azure Functions has limited support for WebSockets, since it is essentially a long open connection, whereas Azure functions are short-lived by design. In this section, we will elaborate on how we came to a solution that does not suffer from the drawbacks of using WebSockets.

3.6.1. Background Jobs

In order to solve the problem of having long connections, we extracted long-running processes to run in separate background jobs. The background jobs are other Azure functions that use so-called Service Bus Triggers. When a background job needs to be executed, a message will be sent to a service bus queue that runs on Azure. The functions with service bus triggers watch for these messages in their respective queues and begin executing when a message arrives in the queue. The queue message includes all the necessary information for the job since this system is stateless like the web API. A high-level scheme of this is presented in Figure 3.2.

Messages can be sent to the queue for example through HTTP trigger functions. Before sending the message to the queue, a `BackgroundJob` entry is created in the database, which contains a status. Initially, this status is set to `Created`. After the service bus trigger function fetches the message, the status of the `BackgroundJob` entry is updated to `Running`. When the job is finished, the status of the job is updated to `Finished`. Of course, we also had to take into account that exceptional behavior could occur. An example of exceptional behavior could be that service bus has trouble connecting to the external parties such as a payroll software web API. In case this happens, the background job gets terminated and a new job that is exactly the same will be started, to retry the operation. In total, this will be tried 10 times so that if a job, for example, encounters a timeout due to the other server being overloaded, the job still has a chance to pass the next time. If the job could not be successfully completed in 10 tries, the message will end up in a *dead letter queue*. A dead letter queue is a special type of queue that contains messages that failed to be handled. In order to deal with these dead letters, we implemented a function that is triggered when messages arrive in the dead letter queue. This function is responsible for setting the job status to `Failed` and logging the exceptions that occurred for any developers that wish to examine these exceptions.

3.6.2. Verifications

A verification in the CVE is an aggregation of multiple background jobs. We have designed an HTTP endpoint that creates a verification, which in turn creates and starts a set of background jobs. The goal of this endpoint was that once all the required basic personal information has been gathered, the entire verification process could be initiated with a single API call. The HTTP trigger function returns an identifier for the verification process to the client and then closes the HTTP connection. The client (which will be the customer's browser) can poll the status of the verification periodically, for example every 5 seconds, using the identifier returned after the creation of the verification. This can also be seen in Figure 3.3.

There are three different results for a verification: `Accepted`, `Rejected` or `Under Review`. The last option will be set when the verification could not be performed automatically and manual intervention is necessary. This could happen due to various unforeseen circumstances, such as a failed job due to a network problem. Since we are dealing with a variety of external data sources, a lot of other unforeseen problems can occur that will all result in an `Under Review` result. For instance, external data sources might treat and store data differently than we do. A simple example could be postal code notation. Dutch postal codes look like: `2628 XE`. Sometimes, they might be written without whitespace: `2628XE`. Every piece of software is written in a different way, so programmers of our external sources might have used different notations. If a mismatch would occur, we would want a human to take a look at the problem. Of course, we tried to take these exceptional cases into account in order to make the matching process as reliable as possible. The postal code mismatch problem is easily circumvented by stripping all whitespace when comparing codes, which is why the CVE will be able to handle different postal code formats without issues. Still, this example illustrates a problem that might result in an `Under review` result. Even though we tried to take all of the options and edge cases into account, incompatibilities will definitely still arise. These cases should also be reviewed by a human. Some issues cannot be prevented at all, such as human error by the company managing their payroll. Payroll software are tools that are made to be used by humans, and humans might make mistakes or use the software in different ways. If there is a typographical error in the address of a customer, for example, we cannot find them. This should also be manually reviewed.

Verifications with the `Under Review` result could be retrieved by a back office application. A human employee should then manually review the problem and perform the appropriate actions. Consequently, the front end should present the customer with a page explaining that manual review of their application is required.

Thus, the verification result is an aggregate of the status of the related background jobs. If at least one background job rejected, then the verification rejects. For the verification, it is also saved why a background job resulted in rejection or failure, so that the customer and the back office can be informed about why the customer was rejected. If all the background jobs accept, then the complete verification is accepted. When the verification is finished and accepted, the client can access the resources that are fetched and stored in the database by the various background jobs, which is for example salary information.

3.7. Bank Account Verification

One of the approaches mentioned in section 2.4 was to use iDeal. To integrate iDeal, we had to use a payment provider that is also an iDeal Partner. Since there are transaction costs associated with iDeal payments, we discussed with the stakeholders on what payment provider to use. Mollie ended up as the payment provider of choice. This meant that we had to integrate the Mollie API into the CVE.

We found a library by GitHub user Viincenttt that provides a .NET/C# implementation of the Mollie API [11]. The library seemed well-maintained, clear and usable, so we decided to use it. In order to provide the cus-

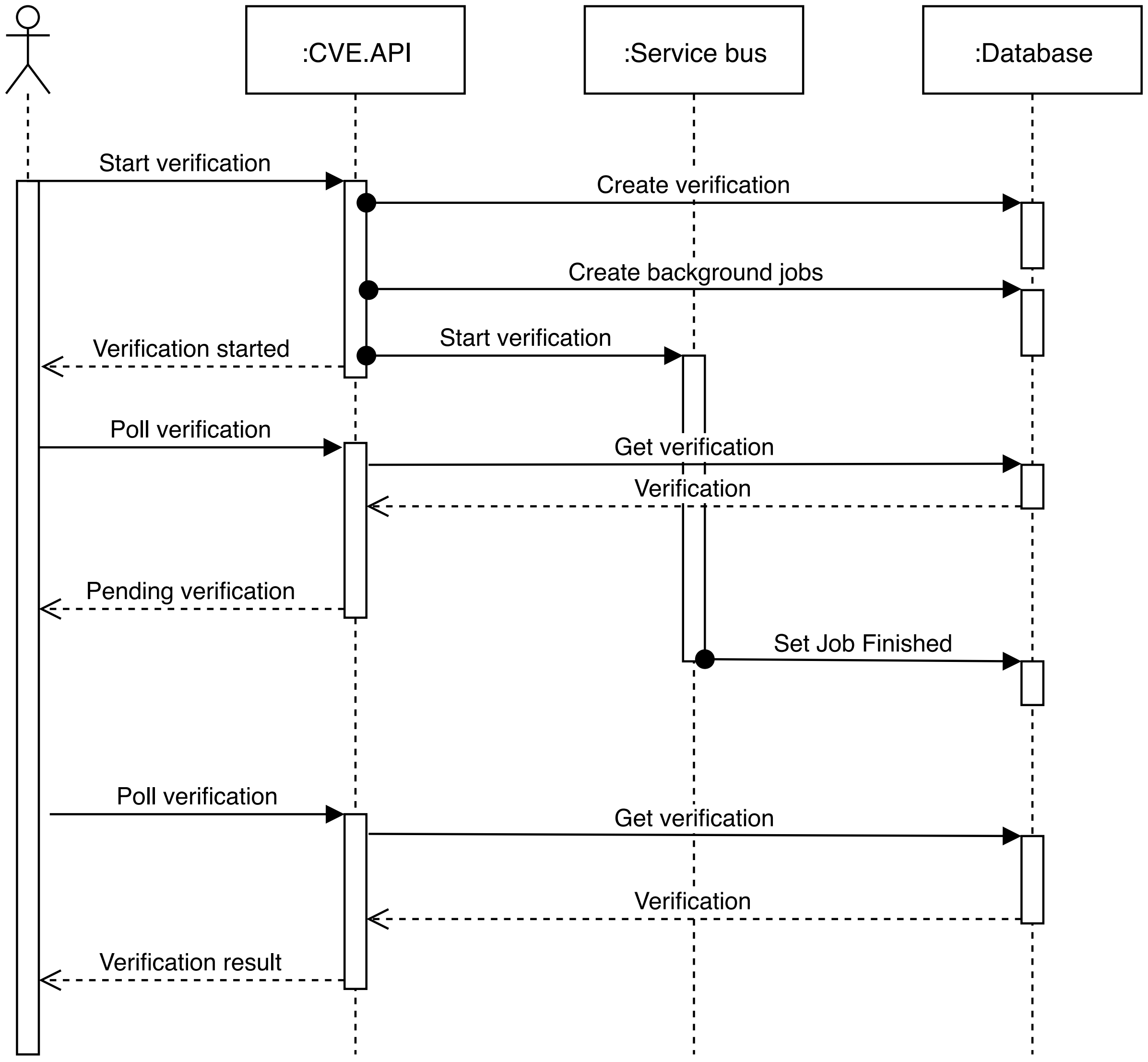

Figure 3.3: A sequence diagram of a verification started by the customer. The customer initiates the verification by making a request to the CVE web API. The web service creates a new verification and the required background jobs in the database. The web service starts the service bus accordingly. The customer then performs the first poll request, but the service bus is not finished yet, so the customer receives a pending verification. Meanwhile, the service bus finishes its job and updates the database accordingly. The next time the customer polls, the verification is finished and the result is presented to the customer.

tomer with the easiest flow possible, the website will immediately present the available iDeal banks to the customer. The other option would be to forward the customer to Mollie for picking the bank, which is easier to implement but would require an extra action from the customer. The following flow has been implemented:

1. The front end sends a request to the CVE in order to retrieve the iDeal issuers that Mollie supports. Our colleagues from the front end made sure that the customer was then presented with a screen where they could select their bank.
2. After the user has chosen their bank, the front end sends a POST request with the chosen issuer and a redirect URL to the CVE in order to initiate the iDeal payment.
3. The CVE creates a database row for the payment, and generates an ID for the payment.
4. The CVE sends a request to Mollie, with the following information:
    - The ID of the payment created in the previous step embedded in the metadata.
    - The issuer chosen by the customer.
    - A price of 0,01 Euro.
    - A description that will show up on the customer's bank statement.
    - The redirect URL sent by the front end.
    - The webhook URL that points to a webhook integrated in the CVE.
5. Mollie responds with the initial payment status and a checkout URL. The initial payment status gets saved in the database and the checkout URL is sent to the front end. The front end then directs the customer's browser to the checkout URL, which points to the iDeal environment of the chosen issuer.
6. The customer goes through the iDeal flow of their bank. The CVE does not know anything about what the customer is doing at this point.
7. After the iDeal flow of the bank has ended (or the payment status has changed in some other way), Mollie calls the webhook that we specified in step 4. For security reasons, this call only contains the Mollie-provided payment ID, which is not stored in our database. This prevents anyone from being able to tamper with the payment information known to the CVE.
8. The CVE then does a request to Mollie with the Mollie-provided payment ID, in order to retrieve status and other information of the payment. The database then gets updated with the new payment status if the status has changed, and the bank account that the customer has used if the status is `Paid`.
9. Mollie redirects the customer to the redirect URL that the front end specified in step 2.
10. The front end then sends a request to the CVE for the payment status. If the status is `Paid`, the bank account has been verified and the customer can continue. Otherwise, the customer either has to cancel or retry.

The other approach mentioned in 2.4 was to retrieve the customer's bank account automatically from the customer's paycheck. The benefit of this approach is that it does not require any additional action from the customer, which is why we implemented it as well. The retrieval happens during the payroll software background job. Since the different payroll software that we have seen allow for multiple bank accounts (and these bank accounts do not necessarily belong to the customer), we had to select the bank account where the employee's net salary is sent to. This bank account was then considered by the CVE to be verified to belong to the customer, but not yet authorized for use. This means that the customer still has to give authorization for the use of this automatically retrieved bank account number.

3.8. Security

As the Customer Verification Engine would be dealing with sensitive personal information it was very important that said data would be guaranteed to be safe and contain no vulnerabilities. No piece of software is guaranteed to be completely secure, yet it remains essential that measures are taken.

3.8.1. Authentication and Authorization

Under no circumstance should there be any doubt on the outcome of the CVE. Therefore, it was crucial that during every single step in the progress authorization took place. Since the platform makes use of an external identity provider, authentication will be handled for the CVE. Any requests arriving at the public web API would first have to pass a gateway, and the customer will already be authenticated at this point. This gateway will perform any authentication steps necessary and automatically insert a unique identifier for the particular customer connected to the account performing the request. By including this identifier, the CVE would be able to ensure that operations are only performed on the customer and their related data. Furthermore, this prevents the CVE from exposing private information of other customers by forging malicious requests.

3.8.2. Identity Document Storage

In order to verify the identity of a customer, the CVE needs to store a picture of an identity document, i.e. passport or ID card. As this is a very sensitive document for a customer, it is important that this is stored in a secure way.

When a customer uploads an identity document to the CVE, the document is encrypted and stored in Azure storage. The key that can be used for decryption is stored separately. This means that if an attacker gains access to the Azure storage account, they are not able to see and use the identity documents.

For the encryption we use the Advanced Encryption Standard (AES) algorithm [12], [13]. We chose this algorithm since it is the standard in encryption according to the National Institute of Standards and Technology (NIST). The algorithm has a clear description of the building blocks and design principles used, and was selected by NIST from an open competition for block cipher algorithms.

3.9. Reliability

It is highly desired that the CVE operates reliably, and that it is a robust system that will not fail under unexpected input or events.

3.9.1. Erroneous Input

In many systems, unexpected input (either on purpose or not) can result in unexpected behavior. Receiving a phone number when you expect an email address should under no circumstances result in storing this in the database. To avoid this Data Transfer Objects (DTOs) have been created that force the client's input to strictly cohere with expected formats [14]. If it happens to be the case that a received request contains undesired input, an appropriate error message will be sent back to the client.

3.9.2. Consistent Error Response

The API makes use of consistent error responses in the JSON format as described in section 3.2. Whenever the client makes an invalid request the API responds with the correct HTTP status code, for example, 400, 404 or 422, and returns a JSON object which contains a message describing what went wrong. This implementation makes it accessible for the client to process invalid requests.

3.9.3. External Data Sources

As mentioned, the CVE heavily relies on external data sources for verification and data retrieval. However, as we cannot guarantee a constant reliable connection with external servers, measures had to be taken to ensure that under such circumstances the CVE will act appropriately. In section 3.6.1, it is described how background jobs act when encountered failing operations. A challenge with using the external data sources was the matching of a customer to an entry in the respective database. In order to make a match that was reliable, a large set of personally identifiable data were compared. For illustration, imagine a scenario where two twin brothers with the same initials, who both work for the same employer, still both live with their parents. It becomes evident that matching on only postal code and name would not be sufficient.

3.9.4. Logging

To enhance maintainability, errors and warnings should be logged. We log all exceptions that occur, even if they are caught and handled properly. We have implemented a tool called Rollbar which allows developers to find and solve issues as soon as they happen [15]. All logs and exceptions are sent to Rollbar if they have a warning level or higher. The Azure Functions also log information about how and when they are triggered. This information can also be used as a log of important actions, such as a log when the customer agreed with the terms and conditions.

4 Development Process

Besides designing and implementing the CVE, we put a great amount of effort into refining the development process. We worked on the CVE as a separate module in a professional development team. In this chapter, we will discuss the tools and methods we used for a convenient and efficient development process.

4.1. Communication and Scrum

Over the course of this project, we tried to work according to the Scrum framework [16]. In the first few weeks of the project, this was not really working out yet. The backlog was not yet clearly defined due to some uncertainties around the product requirements and due to dependencies on external parties. Moreover, we were just not very experienced with Scrum yet. A few weeks in, some changes in the project structure were made and some NAVARA colleagues moved between teams. Together with our colleagues from NAVARA, we decided to start properly using Scrum, and they would teach us everything that we did not know yet. This period taught us many aspects of using Scrum in a real development environment, together with colleagues that were experienced working with Scrum.

We worked with sprint cycles of two weeks, which corresponded to the time between demos. At the beginning of the sprints, we did a sprint planning with the whole team. This could easily take one or two hours since we would plan, expand, categorize and estimate tasks. The estimating of time needed was done using *story points*, by playing a game called *Scrum poker*. For every user story, we would each choose a card with a number, representing how much points we thought a story was worth relative to other stories. At the same time we would reveal our cards. This way we would not be influenced by other's choices. Then we would have to agree on a number, which would result in some insightful discussions if the estimates of the team members differed.

Every morning, we did a daily stand-up meeting (daily scrum) in order to briefly discuss what everyone was working on and if there were any blocking issues. On the days that we were working remotely, we joined the stand-up via an online video call instead. This proved to be very useful: since the work of our NAVARA colleagues sometimes depended on the CVE, we often had to align our tasks and expectations.

At the end of every sprint, we did a sprint retrospective. The purpose of the retrospective is to keep a consistent vision for the team and to solve problems within the team. During the retrospective, we would do various activities, such as writing on post-its what properties we thought a good product should have. We could then arrange them to represent to what extent we have realized those properties.

The backlog and sprint progress were tracked using Atlassian's Jira, an extensive tool built for issue tracking and agile project management [17]. Backlog items were determined together with our NAVARA colleagues. Some user stories resulted from design decisions made by the management. Other user stories were defined by ourselves during the project, since we were free to come up with our own ideas.

Normally, we went to NAVARA's office two times per week, where we met with the other team members. For daily communication when we were not at the office, as well as for sharing files and code snippets, we used

NAVARA's Slack environment. We had our own group for the CVE, as well as groups with our colleagues from NAVARA.

4.2. Version Control

Git was our version control system and our code was hosted on GitHub. Over the course of our project, we used our own master branch separate from the rest of the team. This is elaborated upon in section 4.4. We decided to make this branch protected, which means that code could not directly be pushed to this branch. To merge code to our master branch, we agreed to create pull requests that had to be reviewed by everyone in our group before it was allowed to be merged. The code was extensively reviewed, but also the results from the continuous integration pipeline (see section 4.3) and from Better Code Hub (see subsection 5.1.3). A more detailed description of our code reviews can be found in section 5.2. We worked with feature branches, corresponding to the user stories that were listed in Jira. This meant that not only the feature had to be implemented, but the feature also had to be well-tested and documented.

4.3. Continuous Integration

Our GitHub repository was connected to Azure DevOps, which ran an extensive pipeline with various tasks on our code [18]. The pipeline consisted of the following tasks:

Restore and Build First of all, all dependencies were installed and the whole codebase was built. Since our static analysis tools also ran during the build stage, these were run during this task as well.

Test In the next step, the full test suite was run. This task also generated code coverage data. If one or more tests failed, the whole pipeline would result in a failure.

Generate coverage report The final task generated a coverage report, with the code coverage data generated during the test stage. This report was then immediately published to Azure DevOps. This allowed us to easily check the code coverage of newly created pull requests for example, which in turn made us more aware of our code quality and kept us thinking about testing.

4.4. Collaboration & Working within the NAVARA Team

Since we needed to create a separate module for this project that still integrated with the rest of the codebase of this complete product, we decided to use the same repository as the rest of the NAVARA team. To keep our work strictly separate, we decided together with NAVARA on the following system:

- The CVE module would be built as a separate project module and packages, within the `*.CVE.*` namespace. For the duration of this project, our NAVARA colleagues were not allowed to write or edit code within the CVE module to keep our work our own.
- We developed against our own master branch, which was called `develop-cve`. Using this branch as a master, we were free to develop in any way that we wanted. We occasionally merged this branch into the real master, by creating a pull request. This pull request was then reviewed by our supervisor and colleagues in order to ensure that it satisfied their needs, mainly in terms of functionality.

4.5. Documentation

Writing documentation was of high importance for this project. The product was still in the early stages of development, so our code would serve as the foundation for any future work. This means that not only did we put a lot of effort into writing solid code, but also into writing solid documentation. Paired with the documentation, it should be fairly easy for someone looking at our code for the first time to figure out what is going on. Since the CVE heavily relies on other APIs that are often poorly documented, we needed to clearly write about how we interact with those APIs. We wrote documentation in the following ways:

Code comments The obvious way of documenting code is writing comments. We decided to write comments wherever the code was not self-explanatory. This means that the code is not easily understand-able after taking a quick look at it. An example of self-explanatory code might be short (under 10 lines of code) helper units that only do one thing, which is clear from the method name.

OpenAPI The web API contains over 15 web endpoints. It would cost a lot of time and effort to read through the entire code for endpoints in order to decide what they accept and return. This might be even more difficult for developers who have never worked with Azure functions. Therefore, we have created and maintained an OpenAPI documentation document [19]. OpenAPI is an open format for specifying API documentation in a structured way. This document contains documentation for the URIs, headers, parameters, body, responses and examples. This documentation is very useful for developers that want to integrate with an API. Multiple tools can be found that are able to generate client code in many languages, based on OpenAPI specifications [20].

Markdown More general explanations that did not fit in the above categories were written in markdown files in a `docs` directory in the root of our project. Markdown allowed us to write our documentation in a structured way, that is easily readable, especially on GitHub. The main purpose of these markdown files was to help future developers understand the CVE, and what was required to work with it.

Diagrams We made various diagrams that were put in the `docs` directory as well. These include sequence diagrams, a database model and class diagrams.

5
Quality Assurance

Quality assurance is of great importance in a collaborative environment in order to keep the produced code maintainable. Especially in our case, where our product is a module that is part of a larger product. Other developers within the team will have to use our code, build upon it and maintain it. This is why we highly valued maintainability, and by extension quality assurance. In this chapter, we will explain what measures we have taken in order to measure and maintain the quality of our work.

5.1. Static Analysis

One of the measures we have taken was introducing static code analysis tools from the beginning of the development process. Static code analysis tools perform an analysis by only looking at the code without running it, hence the name *static*.

5.1.1. Code Style

To achieve consistent code style, the StyleCop analyzer has been integrated in the project [21]. StyleCop is a static analysis tool for C# that enforces a certain coding style, ensuring that the written code follows strict guidelines regarding design and style. At first, all StyleCop guidelines were set to generate warnings on build. During our development we discovered that some style guides were not useful while others were trivial and easy to fix. We configured the most common and easy to fix warnings to error on build, in order to make sure the continuous integration build would result in failure. Some warnings were incompatible with the frameworks we were using, so we had to hide those warnings from the build.

Furthermore, we integrated FxCop analyzer, which runs code analysis during compiler execution for common bugs or design flaws [22]. Both tools allowed us to easily find and fix the defects.

5.1.2. Code Metrics

Besides code style and design, we added Code Metrics, a tool to keep track of code metrics [22]. This tool calculates a maintainability index of the code base, based on cyclomatic complexity, lines of code and Halstead volume [23], [24]. The maintainability index reported by Code Metrics was **87**. The tool uses a similar but different calculation than Oman, Hagemeister, and Ash did for this index. The tool reports an index between 0 and 100 where less than 20 is not maintainable and the code should be revised.

5.1.3. Better Code Hub

NAVARA already used Better Code Hub by the Software Improvement Group (SIG) for their development, so we could easily integrate it in our workflow [25]. We had Better Code Hub set up to verify all of our pushed work and pull requests. During the first few weeks of the project, it was hard to adhere to the Better Code

Hub standards since we were still learning C# and were setting up the project. After about three weeks, the codebase started to become more consistent and we started to become more strict with the Better Code Hub guidelines. In GitHub, we set the Better Code Hub feedback to be blocking, so if the changes made in the PR were insufficient, we made the author update the code to satisfy the guidelines. This helped us to keep a maintainable code base starting fairly early on during the development phase. From the third week until the last week we maintained a 10 out of 10 score on Better Code Hub.

5.1.4. SIG Code Evaluation

During our project we also sent the code to SIG for evaluation. Once halfway during our project and once at the end to evaluate the improvements we made based on the feedback that we received after the first evaluation. The original feedback from both evaluations can be found in Appendix C.

First Evaluation SIG

The feedback that was given after the first feedback consisted of the following key points:

- **Unit size**: The first major argument concerned the size of some units of code. It was recommended to keep said units as short as possible, as long units indicate that there were too many responsibilities for one unit of code. Reducing the size would result in more maintainable, and better separation of functionality. However, it should be noted that some development functions have been analyzed, and regarded as too long. These development functions would not be present in the final product, and were only temporarily present in the codebase to serve as reference.
- **Unit complexity**: The second argument concerned the complexity of units of code. This metric does not necessarily mean that the functionality of the code is complex, but is complex in the way that it is written. It is recommended that code does not have a high complexity. Lower complexity increases readability and maintainability. The units in our code base that were considered too complex were methods that either have too many responsibilities or grew over time, meaning they started out simple but became more complex when new functionality was added.

In order to incorporate the feedback concerning unit size, we have made the decision to put more effort in keeping future units as short as possible, as well as revising previously written code that we considered to be too long. The units were shortened by identifying sections that had a clear distinction from the rest of the unit, then the sections were split into multiple methods. This made the long units easier to read. Another consequence was that the new separated units reduced code duplication and were more testable. Any new long units of code in features were picked up by code review, so we could prevent any decline in this metric.

For reducing the unit complexity, we used the Better Code Hub analyzer from subsection 5.1.3 to identify the most complex units with a cyclomatic complexity of at least 10. Most of the time these units were long and could be resolved by using the method for reducing unit size. Other solutions include simplifying the logic. Code logic sometimes became too complex, since new features would be added, without looking at the current implementations.

Second Evaluation SIG

At the time of writing, we have not yet received the second evaluation from SIG so no comments can be given.

5.2. Code Review

Before any changes could be included in the codebase, it had to go through careful inspection from the rest of the team members. Any member had the right to request changes, make suggestions, or even block it entirely. The request for changes was usually about matters that could not be picked up by the automated tools, such as algorithmic design, naming of variables, quality of documentation and tests. A pull request was required to have approvals from all team members that did not implement the feature before it could be merged. In code review we applied the Boy Scouts rule: “leave the campground cleaner than you found it” [26, p. 123]. The same applies to software. Whenever someone requested for review, we made sure that the author did not leave any code smells behind in the same units of code where they made changes, which could easily be fixed. This resulted in more maintainable code.

By doing the aforementioned inspection, team members had a firm understanding of the codebase at all times. Frequently, a code review sparked discussions about the implementation and design choices. Such discussions were likely to result in some changes, or change of plans for future work.

5.3. Testing

It has been made a requirement that any new changes are tested to a high extent. By utilizing unit tests, we tried to both verify correct behavior under normal circumstances, and put the code's robustness to the test when erroneous input was presented. Furthermore, having a properly tested codebase gave us the reassurance that new code or changes were not breaking for any previously written code, utilizing regression testing as a safety measure [27]. The tests have been included in the build pipeline, resulting in failure if one of the tests does not pass. On several occasions, the failure of unit tests gave us insights in flaws that were present in our code.

5.3.1. Tools

To create robust and maintainable we used various .NET libraries for writing tests. The framework we used to write and structure the tests is xUnit [28]. xUnit allowed us to write many tests without writing much boilerplate code. If tests failed, xUnit showed us where and what went wrong. The second framework we used is Moq [29]. Moq is a mocking framework for .NET. This makes it easy to mock dependencies of classes and methods and verify that these dependencies are used in a correct way. With this method of testing we could easily mock the database and test methods that use a connection to a database without actually running a database instance. This reduced dependencies for tests and did not break other tests.

5.3.2. Metrics

As described in section 4.3, our pipeline generated coverage reports of our code after running tests. Using these reports we could compare coverage before and after new features were implemented. The statistics gathered from these reports can be found in Table 5.1.

Classes	112
Total lines of production code	5859
Total lines of test code	5683
Total amount of tests	250
Assert density	**4%**
Test code percentage	**97%**
Covered lines	1580
Coverable lines	1951
Line coverage	**80.9%**
Covered branches	451
Total branches	641
Branch coverage	**70.3%**

Table 5.1: Table showing metrics with relation to tests.

The total coverage is not 100%. In fact, we aimed for 80% line coverage with as many lines of test code as production code [26, Chapter 10]. This is because the codebase contains many trivial units of code that can be tested, but testing them makes no sense [26, Chapter 10]. Examples in our code base include getter and setter methods for classes. Based on the amount of time and maintainability of writing test code for trivial methods, we agreed not to write unit tests. Other code we decided not to write tests for are code entry points. This code was responsible for setting up the runtime environment. It would be too much effort to write and maintain tests, so we decided to test startup code manually, since this code would be run anytime we started the API manually.

The Better Code Hub integration also keeps track of assert density and the lines of test code compared to production code. Better Code Hub reported a **97%** test code percentage with **4%** assert density. This means that there exists almost equal amount of test code compared to production code. Assert density means that 4% of the test code consists of assertions. Assertions are statements in code that check for correct code state. Better Code Hub suggests that, for a code base of our size, test code percentage should be above 50% and assert density above 1%.

5.4. Usability/Acceptance Testing

As the CVE is a back end application, usability and acceptance testing with customers of the platform would not make sense. In addition, the platform was not ready to be made publicly available yet. The CVE is a piece of software that is meant to be interacted with through a web API, by the front end of the platform. This is why it made sense to us to do usability and acceptance testing with the other developers of the platform, in order to see what worked well for them and what could use improvement. Throughout the project, we continuously sought feedback from the front end developers. Over the course of the project the other developers already developed using many of the CVE API endpoints, which resulted in more feedback which we could then immediately start implementing. For example, the developers would discover that responses did not contain some required fields or they experienced exceptional behavior from the API. They would communicate the errors with us, so we could reproduce and fix them. This feedback was received both verbally and through GitHub discussions. In chapter 6, we discuss the feedback that we received from developers integrating with our web API during the last week of the project.

6 Feedback and Reflection

In order to get a clear impression of our performance and the quality of the CVE, we have conducted a survey among fellow developers and colleagues from our project within NAVARA. The feedback that we received was very positive, yet still highlighted that there was room for improvement. The total number of responses was four, having received a response from each developer in the team.

6.1. Feedback on the Product

The following are answers to the question of what the strengths of the product were:

- "*Good separation of concerns, application of the object-oriented paradigm*"
- "*Just learned C# and had better code standards than 90% of the products I've seen*"
- "*Good documentation, high code quality. The CVE is a complex application but it has been handled well with all the different dependencies on external suppliers etc. Structure of all the classes is well thought out, and standard guidelines are for the most part well followed.*"
- "*Clean code, a good amount of automated tests, as well as keeping future extensions on integrations with other systems in mind.*"

On the other hand, our colleagues also had some feedback on how the CVE could be improved:

- "*Could use some inline documentation for classes and methods*"
- "*Sometimes a little sloppy with details, such as forgetting to update a Swagger file or forgetting to add certain objects to a call. More attention to the details before delivering can prevent this. When changing certain configurations such as Connection Values etc, it could be made more clear to the front end team what needs to be updated because they depend on that information. Some database choices seem a little over-engineered/over generic to me, but this is not entirely to blame on you as a team, but also on choices that were made at the start of the project.*"
- "*If only the CI/CD would've been fully set up... It would've been perfect!*"

What we can conclude from this feedback is that we have delivered a product that is considered to be of high quality and well designed. Furthermore, valuable feedback has been provided that we can take with us to future projects. The main message is that we should communicate and discuss changes in more detail, as this will result in a more efficient development process.

Lastly, we also asked to provide a rating for the product on a scale of one to ten. The average result given was an 8.8 out of 10.

6.2. Feedback on the Team

A major focus point for us during the project was gaining experience in working in a professional development team. Even though we had our strictly separated module, we still frequently collaborated with other team members.

The following are answers to the question of what the strengths of the team were:

- "*Nice teamwork, and an open mind towards feedback and suggestions*"
- "*Flexible and fun to work with. Want to do things well instead of just making something work. Nice balance between arguing and getting things done*"
- "*Good internal collaboration and communication, good involvement in the project with thinking out features and issues. Good motivation and when needed asking for help.*"
- "*Good pro-active attitude, you work hard and pose a lot of questions.*"

However, there was some room for improvement according to our colleagues:

- "*Next time you could take initiative to improve the project management (e.g. using Scrum properly) whenever you encounter problems*"
- "*Trying to be more of one team, instead of a separate backend/student team that mostly collaborates with the rest of the team during meetings.*"
- "*You could have been a little more assertive in selecting business partners for integration. Not exactly the responsibility of the team, however it would've been a nice opportunity to do a little extra.*"

The given feedback reflects our experience quite accurately. During the first phase of the project, there was a lack of structure within the entire team since the project just started. This resulted in the work being done inefficiently, losing valuable time. We were a bit hesitant to take on this role, as there were more experienced people from NAVARA present as well. However, we now all realize that it is a good idea to step forward in such situations.

Unfortunately, the second answer indicates that we should have communicated more clearly about the requirement from the TU Delft to have a strictly separated codebase. This specific team member has joined the project at a later date, possibly explaining the confusion. Ideally, an issue or user story would have been dealt with from start until finish.

The average rating that was given was a 9 out of 10, indicating a high satisfaction rate among our colleagues.

"*Impressive job for a bachelor product without real working experience!*"
– Milan Steenwinkel

"*Very pro-active team with a fresh perspective on software engineering and a pleasure to work with!*"
– Yves Candel

7 Conclusion

The verification of a customer for a lender can be a slow process, due to the historically manual nature of this process. Customers need to manually provide information and documents when applying for a loan, which also allows for fraud.

We have designed and implemented the Customer Verification Engine (CVE), which automates the verification of customers when they apply for a personal loan. It integrates with external services, such as the payroll software of the employer of the customer and the Dutch Bureau of Credit Registration, to retrieve relevant financial information for the verification. Because of this, new customers do not need to manually provide documents, making the sign-up process very efficient. The CVE results in lower interest rates for customers, since it is able to make a more accurate risk assessment because the automatically gathered data is much less prone to fraud.

The different components of the CVE, including API, database and service bus, run on the Azure cloud platform which makes it easily scalable and deployable. As the CVE will continue to be worked upon, the codebase has been designed to make it relatively easy for future developers to add new integrations and features. This is achieved by having a system that is modular, in addition to being well-documented. Furthermore, the CVE is integrated with many code analysis tools, which ease the process of development for future developers. Finally, An extensive test suite validates correct behavior, and will ensure that any new additions will not break previously written code.

8
Discussion

Over the course of the project, we tried designing a solid foundation for the CVE. However, further research is still necessary. In this final chapter, we will discuss some recommendations for the CVE and necessary further research and implementations.

8.1. Ethical Issues and Considerations

Given the data that the CVE is dealing with, it should be no surprise that privacy and security should always be a top priority. Throughout the development we encountered moments where we were wondering whether or not it was ethically responsible to store certain data. A lot of regulations and legalities come into play when you are dealing with lending, or even personal data for that matter. As privacy and legalities are concerned, the three most important questions that we had to ask ourselves during the process were the following:

- **What data do we need to store?** Some data simply needs to be stored, such as basic personal information. Some data is also mandatory to store for lenders by Dutch law or for example the authorization from a customer to make automatic bank account withdrawals.
- **Are we allowed to store this data?** There are legal limits to what data can be stored and what not. An example of a legal limit is the recently introduced GDPR [30]. It disallows storing data that is not directly needed for business operations.
- **Do we really need this data?** There have been several occasions where using a larger set of personal information would help us, whilst not being crucial for servings its purpose. In such situations, we had to investigate whether or not the benefits outweigh the downsides.

During this project, we tried to find a balance on what data we were going to store based on the three questions above. Obviously, if some data was necessary or mandatory, we save it. If we had some data though that we were allowed to store, but we did not necessarily need it, we simply did not save it.

Lastly, we believe that the current state of the CVE is a good solution for the problem of bias in high impact decision making. If the verification process was done by a team, there would be a risk that assessors unconsciously take factors into account that should not influence decision making. Examples of such factors can include a name indicating the ethnicity of a customer, or gender. As the verification is fully automatized, no bias is directly introduced by human factors during the verification process. However, if the CVE were to implement risk management for their customers using machine learning, the system could become biased based on the data that the CVE currently stores. We recommend that if machine learning would ever be considered, it should be preceded by extensive research on bias in verifying customers [31].

8.2. Future Work and Recommendations

As the duration of the Bachelor End Project is only around ten weeks, and the product will continue to be worked on, we have continuously put effort into making the code both future proof and easy to work with or build upon for future developers. The codebase of the CVE has been well documented, including OpenAPI documentation for front end developers. The things we expect to be worked on by others include:

Back office API An important element of the CVE that still needs to be built is an API for the back office. This would allow customer support members to easily help customers that encounter issues during the onboarding. For example, as mentioned in subsection 3.6.2, this is where the verifications that are flagged as `Under Review` would end up. The back office should also be able to manage the customer's personal data, or for example, delete customer's data in order to comply with the GDPR's right to be forgotten [32]. Still, there might be a legal obligation to keep some data, which is allowed under GDPR and should be taken into account when designing the deletion procedures [33].

Integration partners As the product grows, we expect more payroll software to become partners with the product. The APIs of new partners need to be included in the CVE, which is a task for future developers. We have tried to make this process as effortless as possible.

BKR VIS-check This service by the BKR checks whether an identity document is stolen or not. It would be an extra measure that can be taken against fraud. We thought about automatically reading the document number from the uploaded identity document right before it was encrypted in order to save the customer some work and to prevent mistakes. To easily integrate this into the rest of the CVE, this could be done using Azure's OCR service within the Computer Vision API [34].

Over the course of this project, we encountered a few topics that need further research or should be looked into.

GDPR compliance It is essential for a company that is working with sensitive personal data to comply with GDPR. We have attempted to take GDPR into account as much as possible when designing the various aspects of personal data storage and retrieval. Still, we are not legal experts in any way and we might have missed some details. To ensure that the CVE fully complies with regulations, especially GDPR, an audit should be done by someone with the appropriate legal knowledge.

Security Security is another important aspect when handling sensitive personal data. As can be read throughout this report, we kept security in mind when designing the CVE. Even though we tried our best, there might still be mistakes and bugs in our code or we might have overlooked something. We recommend performing a security audit before taking the CVE into production. This can, for example, be done through penetration testing.

8.3. Lessons learned

Understanding payroll software Using and understanding the different payroll software solutions was not as straightforward as we thought it would be. It took us a lot of time to experiment with the different solutions that we had access to. We had to find out how we could interact with it through their APIs in the ways we wanted, and we had to find out how we could translate between different types, such as contract types. This took more time than expected since we had to read customer support documents and sometimes even had to contact the customer support of the payroll software solution. This taught us to better take into account that understanding something unfamiliar, and in particular other software, might take more effort than you are used to, especially if you are unfamiliar with the type of software.

External dependencies Over the course of this project, we had some external dependencies. We were depending on external companies for providing us with a test environment for example, which could take more time than anticipated. Sometimes we even had problems with these test environments that required more communication and delays. Again, this taught us to take into account that you can not simply assume that external parties will deliver on time, or will do anything perfectly.

Writing maintainable code takes time We have put a lot of effort into making the CVE easy to work on for future developers. We have taken several measures to improve the maintainability: Clear documentation, descriptive naming of variables and methods and modular design. It was important to occasionally step back and take a more critical look at things that made sense for us, but might not be as self-explanatory to those not involved in the discussions motivating certain design choices. Even though this benefits the project in the long run, we were often surprised by how much extra time it takes to write maintainable code. This project taught us both the importance of maintainable code, as well as the non-triviality of the task.

A

Database design

This appendix is an extension to chapter 3. For confidentiality reasons, this appendix has not been included in the public version of this document.

B
Project Description

The following text was the original project description as found on BEPSys:

> We are building a peer-to-peer lending platform which is aimed at helping consumers getting rid of their debts. A key element of this project is the identification and verification of both identity and employment documents provided by either the consumer or their employer. These documents require data mining and verification of the authenticity of said documents.
>
> The students will work on developing an engine handling many of these aspects. Acquisition of said documents will also be attempted using API connections with employer administration software solutions which may or may not have a well-documented API or no API at all.
>
> Design decisions still to be taken and discovered are how to extract data from the documents, either by simply 'reading' the documents through code, applying optical character recognition and/or machine learning. Using these technologies, a risk classification could be defined using machine learning.
>
> Several technologies will be used and some are still to be decided upon, but will include at least .NET Core, NodeJS, Git, Microsoft Azure cloud platform, Azure Functions, Azure DevOps, Microsoft SQL Server, BetterCodeHub, Angular, JavaScript, TypeScript, SCSS. This project is subject to strict confidentiality. An NDA will be signed by the students working on it. They will work on-site at NAVARA in Driebergen-Rijsenburg and a location closer to TU Delft.

C
SIG Code Evaluation

The following are the original evaluations from the first and second feedback from the Software Improvement Group. As the communication with SIG was in Dutch, so was the feedback.

C.1. First Evaluation Feedback

De code van het systeem scoort 3.8 sterren op ons onderhoudbaarheidsmodel, wat betekent dat de code marktgemiddeld onderhoudbaar is. We zien Unit Size en Unit Complexity vanwege de lagere deelscores als mogelijke verbeterpunten.

Bij Unit Size wordt er gekeken naar het percentage code dat bovengemiddeld lang is. Dit kan verschillende redenen hebben, maar de meest voorkomende is dat een methode te veel functionaliteit bevat. Vaak was de methode oorspronkelijk kleiner, maar is deze in de loop van tijd steeds verder uitgebreid. De aanwezigheid van commentaar die stukken code van elkaar scheiden is meestal een indicator dat de methode meerdere verantwoordelijkheden bevat. Het opsplitsen van dit soort methodes zorgt er voor dat elke methode een duidelijke en specifieke functionele scope heeft. Daarnaast wordt de functionaliteit op deze manier vanzelf gedocumenteerd via methodenamen.

Voorbeelden in jullie project:

- `PayrollSoftwareFuncs.GetPersonalInfo(HttpRequest)`
- `Credit.PatchCredit(HttpRequest, long)`

Voor Unit Complexity wordt er gekeken naar het percentage code dat bovengemiddeld complex is. Dit betekent overigens niet noodzakelijkerwijs dat de functionaliteit zelf complex is: vaak ontstaat dit soort complexiteit per ongeluk omdat de methode te veel verantwoordelijkheden bevat, of doordat de implementatie van de logica onnodig complex is. Het opsplitsen van dit soort methodes in kleinere stukken zorgt ervoor dat elk onderdeel makkelijker te begrijpen, makkelijker te testen is, en daardoor eenvoudiger te onderhouden wordt. Door elk van de functionaliteiten onder te brengen in een aparte methode met een beschrijvende naam kan elk van de onderdelen apart getest worden, en wordt de overall flow van de methode makkelijker te begrijpen. Bij grote en complexe methodes kan dit gedaan worden door het probleem dat in de methode wordtd opgelost in deelproblemen te splitsen, en elk deelprobleem in een eigen methode onder te brengen. De oorspronkelijke methode kan vervolgens deze nieuwe methodes aanroepen, en de uitkomsten combineren tot het uiteindelijke resultaat.

Voorbeelden in jullie project:

- `Customer.FindOrCreateCustomer(HttpRequest)`
- `PersonalInfo.Equals(object)`

De aanwezigheid van testcode is in ieder geval veelbelovend. De hoeveelheid testcode ziet er ook goed uit, hopelijk lukt het om naast toevoegen van nieuwe productiecode ook nieuwe tests te blijven schrijven.

Over het algemeen is er dus nog wat verbetering mogelijk, hopelijk lukt het om dit tijdens de rest van de ontwikkelfase te realiseren.

C.2. Second Evaluation Feedback

The second evaluation feedback was not available at the time of writing.

D Infosheet

On the next page, the single page infosheet of our project can be found.

Customer Verification Engine

/ Reliable Customer Verification for a Peer-to-Peer Lending Platform

Problem

There is a lot involved in providing a loan, both in terms of legalities and risk management. As a lender it is important to have a clear record of the customers applying for a loan, as this helps assessing the risk that comes with providing a loan. Furthermore, it is required by law to know who it is that you are providing a loan to. To achieve this, loan providers gather a large sum of personal and financial information. The gathering of such information has traditionally been a time consuming practice, both for the customer and the lender. The customer is required to manually find and submit information, and in turn the lender has to verify that the received information is not fraudulent or incorrect. If collection of personal information, payrolls and credits could be done in an automated way, both the customer and the lender will benefit greatly.

Solution

We have designed and developed the Customer Verification Engine (CVE) in order to solve this time consuming process of collecting and submitting documents. The CVE is capable of cleverly combining several external data sources, creating a clear record of the customer. While previously the customer had to manually provide a large set of documents, it is now done at the push of a button. Furthermore, by having a system where the information is retrieved, rather than provided by the customer, the verification becomes significantly more reliable, as there is little to no room for the customer to provide fraudulent information.

The CVE is a robust and scalable system that is capable of handling unexpected behaviour both in terms of input and connection to external sources. An extensive test suite verifies correct behaviour of the CVE under both normal and unexpected circumstances. The information gathered by the CVE will be relied on to determine whether or not a customer is eligible for a loan. As the CVE will be continued to be worked upon, we have put effort into making it extendable for future developers. Using the extensive documentation and the modularity of the system, it should be effortless for future developers to add new integrations with external parties to the CVE.

A Bachelor Thesis by:
M.R. Comans
O.N. de Haas
D.A.J. Oudejans
E. de Smidt

Coach: M. Finavaro Aniche, PhD — TU Delft
Client: Y.J. Candel, MSc — NAVARA

This project was commissioned by NAVARA
Project duration: April 23, 2019 - June 25, 2019
Bachelor thesis can be found at: https://repository.tudelft.nl

Glossary

.NET Core Is a software framework and library created by Microsoft.

Application Programming Interface Is a particular set of rules and specifications that a software program can follow to access and make use of the services and resources provided by another particular software program that implements that API.

Azure Is a cloud computing service created by Microsoft.

Azure Functions Are small pieces of code that are event driven and run on the Azure platform.

Back end In web development, the back end is the part of the program that runs on the server. The back end includes the code that accesses the database, i.e. the Data Access Layer..

Bureau Krediet Registratie Dutch credit bureau.

cyclomatic complexity Is a software metric used to indicate the complexity of a program. It is a quantitative measure of many linearly independent paths there are in a section of code.

Data Access Layer Is a software layer that provides access to data stored in persistent storage such as a database.

endpoint Is a web address (URL) at which clients of a specific service can gain access to it. By referencing that URL, clients can get to operations provided by that service.

Entity Framework Is a .NET framework for mapping relational database objects to .NET objects.

enum Is shorthand for enumerated types, a data type which is a named set of values..

NoSQL Is a database mechanism for storing data in means other than relational tables like SQL.

Scrum A framework for effective team collaboration..

sequence diagram Is a diagram that shows object interactions arranged in time sequence.

SQL database Is a database that stores data in relational tables and uses the SQL (Structured Query Language) to query the tables.

Acronyms

API	Application Programming Interface.
BKR	Bureau Krediet Registratie.
CVE	Customer Verification Engine.
DAL	Data Access Layer.
DTO	Data Transfer Object.
EF	Entity Framework.
HTTP	Hypertext Transfer Protocol.
IBAN	International Bank Account Number.
JSON	JavaScript Object Notation.
REST	Representational State Transfer.
UML	Unified Modeling Language.

Bibliography

[1] Basel Committee on Banking Supervision, *Principles for the Management of Credit Risk*. Basel Committee on Banking Supervision, 2000, pp. 1–2. [Online]. Available: `https://www.bis.org/publ/bcbs75.pdf`.

[2] *Besluit Gedragstoezicht financiële ondernemingen Wft*, Apr. 2019. [Online]. Available: `https://wetten.overheid.nl/jci1.3:c:BWBR0020421&z=2019-04-01&g=2019-04-01`.

[3] BKR, *Wat is een bkr toetsing?*, Original document in Dutch. [Online]. Available: `https://www.bkr.nl/wat-is-een-bkr-toetsing` (visited on 06/20/2019).

[4] iDIN B.V., *Idin*. [Online]. Available: `https://www.idin.nl/en/` (visited on 06/25/2019).

[5] Nibud, *Het bkr: Bureau krediet registratie*, Original document in Dutch. [Online]. Available: `https://www.nibud.nl/consumenten/het-bkr-bureau-krediet-registratie/` (visited on 06/20/2019).

[6] C. iDEAL BV, *Ideal*. [Online]. Available: `https://www.ideal.nl/en/` (visited on 06/25/2019).

[7] R. T. Fielding and R. N. Taylor, *Architectural styles and the design of network-based software architectures*. University of California, Irvine Doctoral dissertation, 2000, vol. 7.

[8] ECMA, *ECMA-404: The JSON Data Interchange Syntax*, Dec. 2017. [Online]. Available: `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf`.

[9] ASP.NET, *Entity Framework Core*, version 2.2.4, 2019. [Online]. Available: `https://github.com/aspnet/EntityFrameworkCore`.

[10] I. Fette and A. Melnikov, “The websocket protocol”, RFC Editor, RFC 6455, Dec. 2011. [Online]. Available: `https://www.rfc-editor.org/rfc/rfc6455.txt`.

[11] Viincenttt, *MollieApi*, version 2.0.6, 2019. [Online]. Available: `https://github.com/Viincenttt/MollieApi`.

[12] J. Daemen and V. Rijmen, “Announcing the advanced encryption standard (AES)”, *Federal Information Processing Standards Publication*, Nov. 2001. [Online]. Available: `https://csrc.nist.gov/csrc/media/publications/fips/197/final/documents/fips-197.pdf`.

[13] J. Daemen and V. Rijmen, *The design of Rijndael: AES – the advanced encryption standard*. Springer Science & Business Media, 2013.

[14] Microsoft, *Data transfer objects*. [Online]. Available: `https://docs.microsoft.com/en-us/aspnet/web-api/overview/data/using-web-api-with-entity-framework/part-5` (visited on 06/22/2019).

[15] Rollbar, *Rollbar website*. [Online]. Available: `https://rollbar.com/` (visited on 06/24/2019).

[16] K. Schwaber and J. Sutherland, *The scrum guide*, 2017. [Online]. Available: `https://www.scrumguides.org/docs/scrumguide/v2017/2017-Scrum-Guide-US.pdf`.

[17] Atlassian, *Jira*. [Online]. Available: `https://www.atlassian.com/software/jira` (visited on 06/24/2019).

[18] Microsoft Azure, *Azure devops*. [Online]. Available: `https://azure.microsoft.com/en-in/services/devops/` (visited on 06/24/2019).

[19] OpenAPI Initiative, *OpenAPI Specification*, version 3.0.2, 2018. [Online]. Available: `https://github.com/OAI/OpenAPI-Specification`.

[20] Smartbear Software, *Swagger website*. [Online]. Available: `https://swagger.io` (visited on 06/18/2019).

[21] .NET Analyzers, *StyleCopAnalyzers*, version 1.1.118, 2019. [Online]. Available: `https://github.com/DotNetAnalyzers/StyleCopAnalyzers`.

[22] .NET Foundation, *Roslyn Analyzers*, version 2.9.3, 2019. [Online]. Available: `https://github.com/dotnet/roslyn-analyzers`.

[23] P. Oman, J. Hagemeister, and D. Ash, “A definition and taxonomy for software maintainability”, *Moscow, ID, USA, Tech. Rep*, pp. 91–08, 1992.

[24] M. H. Halstead, *Elements of Software Science*. New York: Elsevier, 1977.

[25] Software Improvement Group (SIG), *Better Code Hub Website*, 2019. [Online]. Available: `https://bettercodehub.com` (visited on 06/21/2019).

[26] J. Visser, S. Rigal, G. Wijnholds, P. van Eck, and R. van der Leek, *Building Maintainable Software, C# Edition: Ten Guidelines for Future-Proof Code*. O'Reilly Media, Inc., 2016.

[27] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal, "A study of effective regression testing in practice", in *PROCEEDINGS The Eighth International Symposium On Software Reliability Engineering*, IEEE, 1997, pp. 264–274.

[28] .Net Foundation, *xUnit*, version 2.4.1, 2018. [Online]. Available: `https://github.com/xunit/xunit`.

[29] Moq, *Moq4*, version 4.11.0, 2019. [Online]. Available: `https://github.com/moq/moq4`.

[30] European Union, *General data protection regulation*, May 2016. [Online]. Available: `https://eur-lex.europa.eu/eli/reg/2016/679/2016-05-04` (visited on 06/23/2019).

[31] R. J. Mooney, "Comparative experiments on disambiguating word senses: An illustration of the role of bias in machine learning", *arXiv preprint cmp-lg/9612001*, 1996.

[32] European Union, *Art. 17 gdpr - right to erasure ('right to be forgotten')*. [Online]. Available: `https://gdpr-info.eu/art-17-gdpr/` (visited on 06/23/2019).

[33] European Union, *Do we always have to delete personal data if a person asks?* [Online]. Available: `https://ec.europa.eu/info/law/law-topic/data-protection/reform/rules-business-and-organisations/dealing-citizens/do-we-always-have-delete-personal-data-if-person-asks_en` (visited on 06/23/2019).

[34] Microsoft, *Microsoft optical character recognition*. [Online]. Available: `https://azure.microsoft.com/en-us/services/cognitive-services/computer-vision/` (visited on 06/24/2019).