NEXT LEVEL GRADING

Replacing the CrashFix user interface - Final report

R.W.T. (Rik) van Spreuwel

Preface

This document represents the outcome of the internship project *'Replacing the Crash-Fix user interface'*. This report is written in the context of the internship I have executed during my fifth semester of the ICT & Software Engineering Bachelor program at the Fontys University of Applied Sciences. In commission of Ellips B.V., I have worked on the project and this report from xxx.

During this project, I did research on the CrashFix application and the means needed to improve or replace this system. Also, I created applications to replace parts of this CrashFix application. These tasks were performed at Ellips, under the supervision of my company supervisor, XXX

I would like to thank XXX for his guidance and feedback during this project. Your expertise in software engineering has been an invaluable aid. I also want to thank all my other colleagues for their cooperation and five pleasant months at Ellips. Lastly, I would like to thank XXX from the Fontys for his guidance and feedback.

Contents

Abstract

Ellips is a company which focuses on the development of optical grading & sorting technologies for the fruit and vegetable market. The company's software is called TrueSort and allows users to grade crops based on a variety of features. Alongside their grading software, the company currently uses an open source application called CrashFix to collect and analyze crashes that take place in their software located at the end-users. Unfortunately, the CrashFix application is no longer being maintained by its owner. The lack of maintenance on the application led to the application becoming slower, unstable, and lacking in new functionalities. To increase the productivity of the company's developers when analyzing these crashes, the company wanted to investigate the old system and replace parts of this system to create a faster, more stable and maintainable application. This was translated to an internship assignment of which the main research question became: "*How can the CrashFix system be adapted to fix current issues, prevent future issues, extend the system with additional required functionality, and have it maintained by Ellips software development teams?*".

The approach used for this assignment was to first do research on the old CrashFix system to properly define what parts of the system we were going to replace. After defining this, research was performed to choose what technologies to use for building the new components. The last step was to create these new components using those technologies.

The initial research concluded that almost all parts of the system were not maintainable and broken, and therefore needed replacement. The only part that didn't seem to have any issues was the part of the system that processed crashes and created reports of these crashes in the database.

Basically, to replace those parts of the system, a new backend and front-end have been created. The backend consists of an ASP.NET Core application that includes Web API, which uses a new and less complex database that is built by the same application using the Entity Framework Core. The same application contains a background task which periodically synchronizes the new database with the newly processed reports of the old database. The front-end application is a website, built using the Angular framework.

The ideal situation would be to completely replace the CrashFix system. Therefore, it is recommended to eventually create a new application, or expand one of the new applications, to collect and process the incoming crashes. Doing so will replace the last remaining part of the old system, as well as the service that synchronizes both databases in the new system. When all the parts of the CrashFix system are replaced, the company can expand and maintain the application by themselves, remaining independent of outside parties on this subject.

Abstract (Nederlands)

Ellips is een bedrijf dat focust op de ontwikkeling van optische beoordelings- en sorteertechnologie voor de groente- en fruit markt. De software die het bedrijf ontwikkelt, heet TrueSort en geeft de gebruikers de mogelijkheid om gewassen te beoordelen op verschillende kenmerken. Naast hun beoordelingssoftware gebruikt het bedrijf momenteel de open source applicatie CrashFix voor het verzamelen en analyseren van crashes die plaatsvinden in hun software bij de eind gebruikers. Helaas wordt de CrashFix applicatie niet meer onderhouden door de eigenaar. Het gebrek aan onderhoud aan de applicatie heeft ertoe geleid dat de applicatie langzamer wordt, vaker crasht en er geen nieuwe functionaliteiten meer uitgebracht worden. Om de productiviteit van de ontwikkelaars van het bedrijf te verhogen tijdens het analyseren van deze crashes wilde het bedrijf onderzoeken hoe het oude systeem in elkaar zit en welke onderdelen vervangen moeten worden om dit systeem sneller, stabieler en onderhoudbaar te maken. Dit was vertaald naar een stageopdracht waarbinnen de hoofdvraag het volgende is: "*Hoe kan het CrashFix systeem aangepast worden om huidige problemen op te lossen, problemen in de toekomst te voorkomen, het systeem uit te breiden met nieuwe functionaliteiten en het onderhoudbaar te maken voor de software ontwikkeling teams binnen Ellips*".

De aanpak van deze opdracht begon bij het onderzoeken van het oude CrashFix systeem om goed te definiëren welke componenten van het systeem we gingen vervangen. Hierna is er onderzoek gedaan naar welke technologieën het beste gebruikt konden worden voor het maken van deze componenten. De laatste stap was om deze nieuwe componenten te maken.

Het eerste onderzoek concludeerde dat de meeste onderdelen van het CrashFix systeem niet onderhoudbaar zijn en problemen hebben waardoor ze toe zijn aan vervanging. Het enige component van het systeem dat geen problemen lijkt te hebben is het deel van de applicatie dat de crashes verwerkt en hiervan rapporten maakt in de database.

Om deze componenten te vervangen is er een nieuwe backend en front-end gemaakt. De backend bestaat uit een ASP.NET Core applicatie waarin zich een Web API bevind die een nieuwe en minder complexe database gebruikt. Deze database wordt gebouwd door dezelfde applicatie middels het Entity Framework Core. Dezelfde applicatie bevat een achtergrondtaak die periodiek de nieuwe database synchroniseert met de nieuwe data in de oude database. De front-end applicatie is een website gebouwd met het Angular framework.

Idealiter zou het gehele CrashFix systeem vervangen worden, daarom is het aan te bevelen om uiteindelijk een nieuwe applicatie te maken of een bestaande uit te breiden voor het verzamelen en verwerken van de inkomende crashes. Dit zou zowel het laatste component van het CrashFix systeem vervangen, als het gedeelte van het nieuwe systeem dat de databases synchroniseert. Hierdoor kan het bedrijf het systeem zelf onderhouden, uitbreiden en zijn ze hierbij onafhankelijk van externe partijen.

Glossary

CrashReport	When a TrueSort program crashes, the program sends a report to the CrashFix system. We call this report a Crash Report. A crash report contains information regarding the crash – examples of this are: where in the code has the crash taken place, what time has it taken place etc.
Collection	A Collection is a bundle of crash reports with the same crash.

Acronyms and abbreviations

Abbreviation	**Meaning**
TFS	Team Foundation Server
TDD	Test Driven Development
POC	Proof of Concept
ERM	Entity-Relationship Model
API	Application Programming Interface
HTTP	HyperText Transfer Protocol
MVC	Model-View-Controller
OData	Open Data
LDAP	Lightweight Directory Access Protocol
UI	User Interface
GUI	Graphical User Interface
EF	Entity Framework
HTML	Hypertext Markup Language
CSS	Cascading Style Sheets
JSON	JavaScript Object Notation
TU	Technical University
USA	United States of America
PC	Personal Computer

Introduction 1

Ellips develops optical grading & sorting technologies for the fruit and vegetable market. Their product allows the customer to grade fruit and vegetables and runs on any sorting machine, of any brand. The company currently uses an open source application called CrashFix to collect and analyze crashes that take place in their software located at the end-users. Unfortunately, the CrashFix application is no longer being maintained by its owner. The lack of maintenance has led the application to become unstable, slow and lacking in new functionalities. The front-end shows frequent and useless error pages that miss navigation to get back to the useful web pages. The poor quality of the system results in the company's developers spending much time in analyzing and fixing crashes from their end-users and therefore being a large expense.

The assignment is to fix issues of the current system by replacing parts of the front-end and backend of the system. The newly created applications were to be expandable and easily maintainable by Ellips development teams. Therefore, research was performed to define what parts of the CrashFix system needed replacement in order to realize this goal. The main research question became: "*How can the CrashFix system be adapted to fix current issues, prevent future issues, extend the system with additional required functionality, and have it maintained by Ellips software development teams?*". Also included in the scope of this project is the realization of the parts in the CrashFix system that have to be replaced.

Ellips wishes to develop applications following the Scrum methodology. This means the project was executed in iterative periods of time, called sprints. In this case, the sprints are blocks of 2 weeks in which tasks are performed. At the end of every sprint, a retrospective was held to look back on the performed tasks and to provide the company supervisor with insight on how the project progresses. The development of applications was, where applicable, executed Test-Driven. Therefore, tests were written before implementing features.

This report covers the subjects mentioned earlier in the following order: Chapter 2 gives more information about Ellips and their product. In Chapter 3, all details about the assignment can be found. Chapter 4 presents the most important research within this project and gives the answer to the main research question. Chapter 5 gives the implementation of the newly created system and the reason behind this implementation. Chapter 6 presents the conclusions made in this internship and gives recommendations for Ellips.

Ellips

2

The supervising company for this internship is Ellips. In this chapter, Ellips is introduced. A brief description of the company's history is given, as well as a description of the TrueSort product.

2.1 About Ellips

Ellips is one of the leading company's worldwide on the development of optical grading & sorting technologies for the fruit and vegetable market. Ellips provides its software and hardware under the name TrueSort. The product allows the customer to grade fruit and vegetables and runs on any sorting machine, of any brand.

Besides the product, Ellips provides training for both the dealers (machine builders) and the end-users. Ellips also offers customer support to contact whenever a customer has questions about the software.

Ellips has 2 different branches. The main branch is currently located at Esp 300 in Eindhoven in The Netherlands. The second branch is located in Wenatchee, USA. Ellips currently employs approximately 50 employees, of which 5 in the USA.

Erwin Bakker is the Managing Director of Ellips. He is part of the management department and deals with customer relations. The management department looks over the development and the support departments. Support is separated into 2 departments, Support Ellips HQ at the branch in the Netherlands and Support Ellips USA at the branch in the USA. Development is separated in scrum teams. There are three scrum teams within Ellips. Team Wombat and Team Delta are feature teams working on the new TrueSort 2.0 product and maintaining the current TrueSort software. The intern is part of Team Wombat. Team Vision works on all the Vision algorithms for grading the crops. The organization described above is visualized in an organigram in figure 2.1.

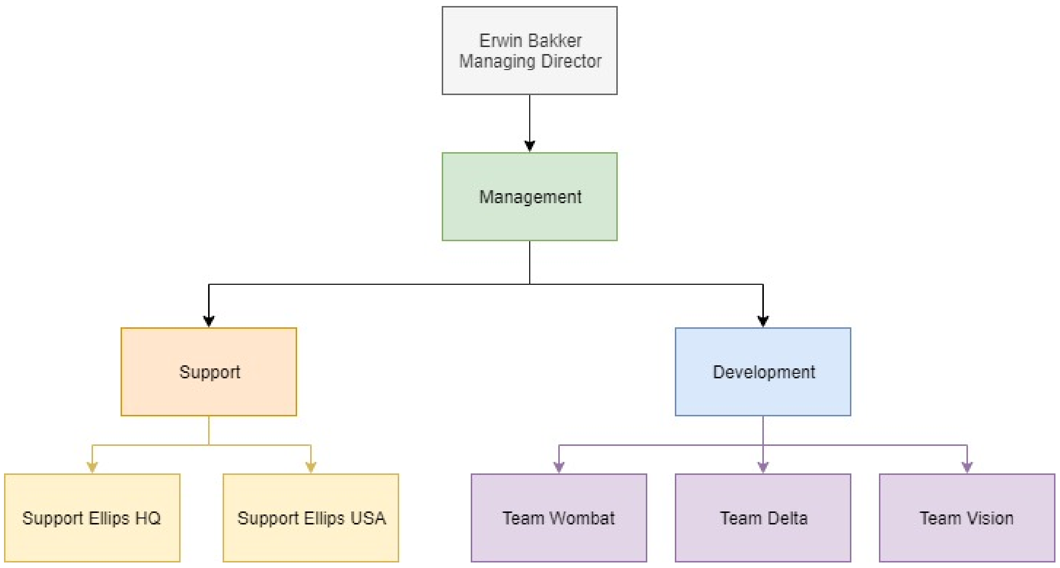


Fig. 2.1: Organigram visualizing my location in Ellips.

2.2 History

Ellips was founded in 1989 by Erwin Bakker and a partner. At that time, they were located on the grounds of the Technical University Eindhoven. Ellips first started sorting fruits in 1990 and sold its first 6 systems within The Netherlands. In 2004 Ellips created their own real-time operating system called Heros. This operating system is used to run the TrueSort software and is connected to the sorting machine's cameras.

While continuing to provide software solutions in this industry, eventually in 2008 Ellips introduced the current grading platform, TrueSort. Ellips is currently working on creating a successor of TrueSort, called TrueSort 2.0. This successor's main goal is to create a more user-friendly user interface.

In its history, the company has moved its location in Eindhoven 3 times. It was 2011 that the company moved to its current location in Eindhoven, Esp 300. In 2014 Ellips founded Ellips USA, a new establishment providing customer support, with as goal to increase the number of hours per day for which customer support is available. Ellips' first and last locations are displayed in figure 2.2.

Fig. 2.2: On the left of the image, you see the first location of Ellips at the TU Eindhoven. On the right of the image, you see Ellips current location in Eindhoven, Esp 300.

2.3 TrueSort

The TrueSort software creates a large number of images of the crops going through the sorting machine. Using these images, features like size, color and quality can be determined. Besides those features, the internal quality of a product can be accessed with a projector designed by Ellips. Using this projector, deviations like rot, brix, dry matter, acidity, and ripeness can be determined as well.

When delivering the software, the hardware needed to run the sorting software is also delivered. This hardware exists of:

- A Windows PC running the TrueSort GUI software.
- One or more Heros PC(s). These computers are connected to the cameras and run the sorting software.
- The cameras used to create images of the crops.
- An I/O controller specially designed to control the exists of sorting machines.

The TrueSort software is designed modularly. The core measurement software has 5 modules, it can grade crops based on weight, size, color, external and internal quality. In addition to the core modules, Ellips offers add-ons for pre- and post-measurement, offering extra functionality. Examples of these add-ons are the Application Programming Interface (API) for exchanging information with external software and the product labeling module, used for operating labelers. Figure 2.3 displays a sorting machine with its operating system.

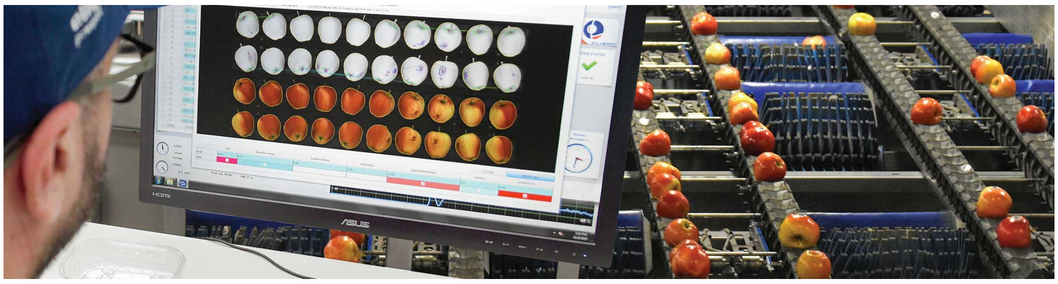


Fig. 2.3: Image of a sorting machine running on the TrueSort software [4].

The assignment 3

In this chapter, the reasons for this internship are introduced in the problem description. The goal of the project is given, as well as an extensive description of the assignment. Last, the scope of the research during this assignment is given.

3.1 Context

To help maintain the stability of their software, the company currently uses an open source application called CrashFix. This application collects and processes crashes in the software located at the end-users. The CrashFix application visualizes the information of the crashes in a web application. The application is used to analyze crashes and detect bugs, with the main goal to resolve those bugs.

In Figure 3.1 you can see how the current CrashFix system functions visualized in a diagram. The CrashFix system consists of 3 main components. These components are a crash reporting library, a backend service and a web application (front end).

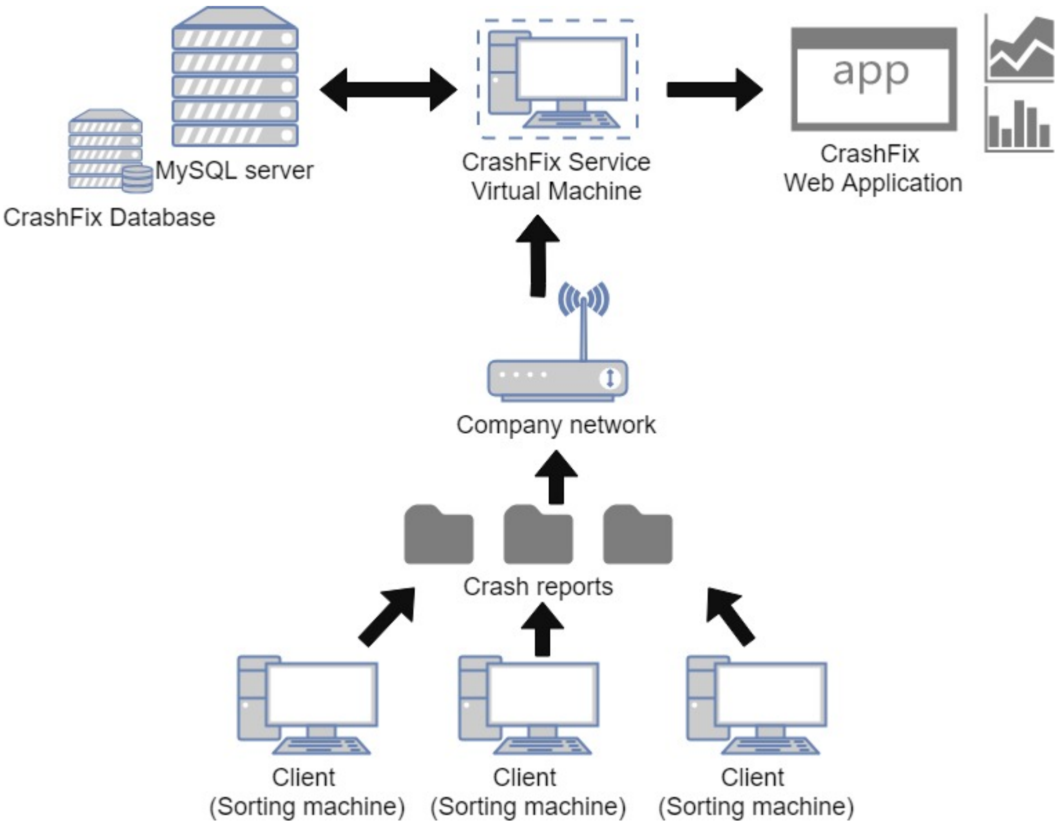


Fig. 3.1: The working of the current CrashFix system visualized in a diagram.

The crash reporting library is a client-side component, it is shipped with their installed software. When a crash takes place in one of their distributed systems, the crash reporting library bundles the files corresponding to that crash in a zip file. This zip file is then sent to the CrashFix service.

The back-end service is a server-side component. The service collects the incoming zip files and saves them to disk. The service then analyses the files and creates a report from the most critical information, this is called a CrashReport. This CrashReport is then saved in a database. This service also places these CrashReports in groups based on its crash. Such a group is called a Collection.

The last component, the web application, is also a server-side component. The web application retrieves the CrashReports and Collections from the service and displays it on a web page. The application also enables you to track bugs by creating a bug object from CrashReports and Collections. All the lists of CrashReports, Collections, and bugs are filtered by a project (applications in which a crash can take place) and can be filtered by a project's version. This disables you to for example view all CrashReports from all projects in one list. The application's home page shows statistics of the currently selected project.

Both server-side components are hosted at Ellips in Eindhoven and are approachable only from within the company's private network.

3.2 Problem description

The problem lies with the CrashFix application mentioned earlier. Unfortunately, the open source application is no longer being maintained by its owner. As a result of this, the application does not get any new functionalities or updates. The lack of maintenance on the application has also led to the application becoming unstable. A complete list of issues with the CrashFix application is listed below:

- The application frequently redirects to an error page. This error page misses the navigation bar, resulting in navigation problems.
- Deleting bugs/CrashReports/Collections can result in error pages due to missing objects.
- The application makes sub-optimal use of your screen size.
- The application automatically logs out users that are inactive for over 5 minutes.
- The statistics on the home page generally give little extra information. This is partly because this page is very fragile and can break when items are deleted.

Besides fixing the problems with the current system, the company would also like to see the following extensions if there is enough time:

- The ability to connect Bugs in the system to the company's Team Foundation Server scrum board.
- An overview page showing useful information in the form of graphs and lists. Examples of these are:
  - A bar chart showing the number of CrashReports per day of the current week.
  - A list of the Collections with the most CrashReports.
  - A notification that CrashReports are being received in Collections connected to a bug that is marked as solved.

The problems and the lack of functionality giving a good overview can sometimes result in spending a lot of time analyzing CrashReports. Time of which could be spent elsewhere, for example, on the development of their upcoming TrueSort 2.0 product.

3.3 Internship goal

The goal of this project is to make it easier for the Ellips' developers to get an insight on what bugs are in their products by providing a more stable web application. The new web application should make it easier for the developers to keep track of their bugs, and more importantly, solve them.

3.4 Assignment description

The assignment is improving the user interface of the outdated CrashFix application. Initially, the idea was to create a new front-end web application that communicates with the CrashFix backend, as a replacement of the CrashFix website. In addition to replacing the current functionality of the CrashFix application, the company wanted new functionalities. Examples of these functionalities are a global search function or the ability to connect bugs of the CrashFix application to the companies Team Foundation Server, to keep track of bugs in their online Scrum backlog.

In the first week of the internship, research showed that the CrashFix backend is part of the problem with the CrashFix application. Later during the internship, therefore not mentioned in the Project Plan document, research on the current database concluded that its way to complex and that it was better to realize a new database. Because of this, the company's priorities changed.

The main goal is to create an expandable and easily maintainable web application. To realize this, we decided it was best to create a new backend for the new application, in order to replace most of the old CrashFix system. Because this made the project's scope much larger, the need for new functionalities gained less priority. This resulted in the new application existing of the following components:

1. A new front-end web application.
2. A new database in order to simplify the current data structure.
3. A new, well documented, Web Application Programming Interface (API) for the web application to communicate with the database.
4. A service to synchronize the old database with the new database.

The following CrashFix components would fall outside the project's scope:

- The crash reporting library.
- The back end service for collecting incoming zip files and processing them.
- Authentication. Authentication for the application does not have to be implemented, since it will only be approachable from the company's network and is therefore unnecessary.

This project was executed following the Scrum methodology. The development of the application was executed test-driven, following the principles of Test-Driven Development.

Due to the certain technologies being used within Ellips, to keep the new application maintainable by their employees, there were some technology boundaries for building the new application's components. These boundaries are listed below:

- The new database is to be developed in MySQL.
- The new Web API should be developed in either Microsoft's ASP.NET or ASP.NET Core. ASP.NET and ASP.NET core are programmed in C#.
- The service for synchronizing the databases is also to be made in C#, possibly as a background task of the Web API.

Ellips doesn't currently develop front-end web applications and therefore has little experience with these frameworks. Due to this, there was no technological boundary for building the front-end of the new application. Therefore, it is for the intern to decide what technology to use by researching existing frameworks, to see what framework was most suitable for the application and for the company's needs.

3.5 Research subjects

Based on the goal of the project, one main research question can be created. This research question is:

- "*How can the CrashFix system be adapted to fix current issues, prevent future issues, extend the system with additional required functionality and have it maintained by Ellips software development teams?*"

To answer the main research question, research was done on several subjects. At first, the current CrashFix system was analyzed in order to find what parts of the system needed replacement. After analyzing the old system, research was done on technologies for creating the new system. The full list of research subjects are listed below:

- The current CrashFix system's structure.
- The issues with the current CrashFix application.
- The requirements for the new application.
- The most suitable front-end framework for the project.
- The best way to document the new Web API.
- The principles of Test-Driven Development.
- The current CrashFix database structure.
- The possibility of using Microsoft's Entity Framework Core (EF) for creating and communicating with the new database.
- The use of the Open Data Protocol (OData) for Restful API's.

Many of the results of these research questions can be found in the next chapters. For the full documentation of the research performed in this project, see Appendix B: Research document.

Initial phase 4

This chapter will explain the most important research that led to the answer to the main research question of this internship. Doing so, the research that led to the current assignment description is given. After this, the Test-Driven Development process is discussed. The chosen front-end framework for building the new application, as well as the reason for that choice is given. Also discussed is the use of the Entity Framework Core and the Open Data Protocol. The research regarding the subjects not mentioned here can be found in Appendix B: Research document.

4.1 Defining the assignment

At the start of the project, the first thing that needs to be done is to properly define the assignment. In order to do this, the first thing to do is to analyze the old system to find the biggest problems with the system and see what components needed replacement. After doing this, the requirements for the new system were set by looking at the most important functionalities of the current system. This process eventually resulted in the assignment described in 3.4: Assignment description.

4.1.1 Analyzing the old system

The problems with the current CrashFix system were defined by asking several developers of the company what problems they encountered and by playing with the application itself. The most important problems for the rest of this chapter were that the application frequently returns an error page, which misses navigation functionality, resulting in navigation problems. The complete list of problems was described in 3.2 Problem description.

The analysis of the CrashFix system's components and deployment was performed by asking the employees that first installed the application about how it was deployed, as well as looking at its official (but very limited) online documentation. The analysis of the CrashFix system's components and its deployment was described in 3.1 Assignment context.

Initially, the new front-end web application was supposed to communicate with the old CrashFix backend. During the research corresponding the CrashFix system's structure, it was found that the CrashFix backend isn't callable from outside its front-end application. It seemed the better option was to create a new Web API, because the instability of the current system could just as well be caused by its current backend.

The analysis of the old system's database was performed by accessing the database and looking at its structure and the data it contained. The CrashFix database is built in MySQL. Therefore, analyzing the database was performed using MySQL Workbench. The database has 30 tables, of which around a third are not being used and are therefore completely unnecessary. Authentication would not be implemented in the new application, leaving some tables out as well. The CrashFix system currently kept a history of changes made to bugs, which the company didn't require for the new system and could be simplified into fewer tables. Due to all this, the decision was made to create a new database for the new system, replacing the old complex database. The full research about the CrashFix database, including a generated Entity-Relationship Model, can be found in chapter 9 of Appendix B: Research document.

4.1.2 Requirements for the new system

The defining of the requirements was performed by the intern in cooperation with the company supervisor and one of his colleagues. The most important requirements are the base functionalities of the old CrashFix system. These requirements are listed below:

- The ability to list CrashReports, Collections, and Bugs.
  - The ability to filter these lists by projects and versions.
  - The ability to sort lists of CrashReports, Collections, and Bugs.
- The ability to view the details of a CrashReport, Collection, and Bug.
- The ability to create and alter Bugs.
- The ability to add Collections to a Bug.
- The ability to delete CrashReports, Collections, and Bugs.

Aside from the base requirements are the additional requirements. These requirements are new/less important functionalities that the company would like to see but can fall out of the project's scope if there is a lack of time. These additional requirements are listed below:

- The ability to connect Bugs in the system to the company's Team Foundation Server scrum board.
- The ability to search for CrashReports, Collections, and Bugs.
- An overview page showing useful information in the form of graphs and lists. Examples of these are:
  - A bar chart showing the number of CrashReports per day of the current week.
  - A list of the Collections with the most CrashReports.
  - A notification that CrashReports are being received in Collections connected to a bug that is marked as solved.

All given requirements were transformed into user stories. Some of these requirements were split into multiple user stories. These user stories are listed and prioritised in chapter 5 of the Research Document in Appendix B: Research document.

4.2 Test-Driven Development

Test-Driven Development is a software development approach in which 3 steps are repeated. First, a particular functionality or method is chosen, and a test case is written for that functionality. This test case will then fail because there is no code to make the test pass. After the failing tests are written, the code is written to make these tests pass. When the tests pass, the developer looks back on his code and tries to clean and improve it. After refactoring the code, the cycle gets back to writing new tests. This process is called the Test-Driven Development cycle. This cycle is visualized in Figure 4.1. This image was retrieved from a blog and its source can be found in the bibliography [20].

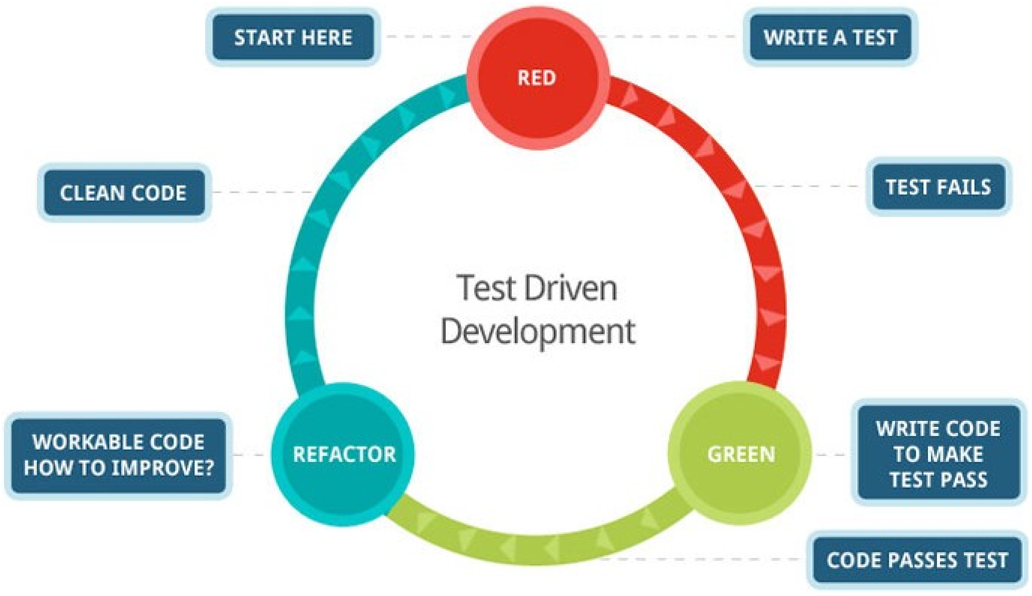


Fig. 4.1: The Test-Driven Development cycle visualized in an image.

The goal of TTD is to reduce the number of bugs and flaws in an application. This is done by having a good set of tests, these tests should find these flaws in the application before releasing the software, therefore reducing the number of problems the end-users encounter. Solving these problems before releasing the software can save a lot of time and money. Because the application had to be well-tested, and it is a good learning objective, the TTD approach has been taken to develop the new applications for the company.

4.3 Finding the right front-end framework

At the start of this project, research was performed to choose what front-end technology to use for developing the new application. The chosen framework was Angular. The research resulting in this decision is described below.

When it comes to front-end frameworks, there is a lot to choose from. To decide what frameworks to look further into, the intern searched the internet for articles comparing different frameworks and the websites of frameworks themselves.

In result of this, the first thing that came to mind was using Microsoft's ASP.NET MVC with Razor for developing the front-end, because the Web API was going to be built in ASP.NET as well. The disadvantage of Razor is that Razor itself uses server-side rendering. When using server-side rendering, every time you want to view a new web page, that specific web page has to be requested from the server, damaging the responsiveness of the website. The other option is client-side rendering. When using client-side rendering, the entire website is requested once, and then new web pages can be loaded without calling the server, resulting in a more dynamic and responsive website. Because the new user interface has to be dynamic and responsive, while handling a large amount of data, the company wants a client-side rendered framework to be used. The usage of Razor in addition to such a framework would only increase the complexity of the application and was therefore discarded.

Besides looking at ASP.NET with Razor, the most interesting frameworks found in this research were Angular and React (ReactJS to be more specific). The reason for this is that both are popular frameworks with a large user base, both were supposedly easy to learn, both are component-based and both use client-side rendering. When a front-end framework is component based, you can split your web application into small components, these components can be easily reused to prevent duplicate code. These features are why these frameworks were looked further into. For both frameworks, a Proof of Concept (POC) was made following a tutorial. For Angular, I followed their Tour of Heroes tutorial using Visual Studio Code, for React I created a test project using Microsoft Visual Studio 2017. While making the POC's, the most important things that were examined are:

- The project structure. A clear project structure is needed for the developers that have to maintain the application, so they can easily navigate through the project.
- The separation between UI and logic. UI and logic should be nicely separated to keep the code readable.
- Testing within the framework. The logic should be testable using unit tests.

The package structure of a project is something that you can define for yourself, but when using a component-based framework, it's the most logical to group the components in a folder. Despite the package structure being similar, there are some differences in the project's file structure between Angular and React. For one, React is mainly written in JavaScript, while Angular is mainly written in typescript. Also, components in React consist of just one JavaScript file, which injects HTML in the base HTML page. While Angular's components consist of their own CSS and HTML file, connected with a typescript file.

As mentioned above, the Angular component nicely separates its UI and logic by having their HTML and typescript code in separate file. In React, on the other hand, components consist of a single JavaScript file which injects its HTML code into the base HTML page. This makes the code hard to read and understand.

Regarding testing in these frameworks, both have many different test frameworks to choose from. When testing Angular, the jasmine testing framework was examined and used because it's recommended in the official Angular docs. With jasmine, you can test the logic in your components, but also other code, like services for making API calls to the Web API. An attempt was made to try and test the React POC using jasmine as well, but it was hard to get this configured in Visual Studio 2017. At this point, the choice of what framework to use was already clear, and therefore it was unnecessary to spend much time on testing in React.

The result eventually was to choose Angular as a framework for developing the front-end of the web application. This choice was made due to several reasons:

- Angular's project structure is easily navigatable and readable.
- Angular has its UI and logic nicely separated in different files.

- Angular is very well documented. This is needed for the company's developers that will have to maintain the application.
- Angular's logic code can be tested relatively well.
- Angular has a very fast compilation. When you save your project, the application is automatically recompiled and the browser is refreshed. This is nice to have for developing the interface.

4.4 Usage of the Entity Framework Core and the Open Data Protocol

When building a Web API with ASP.NET Core, using the Entity Framework (Core) should be considered. The Entity Framework, from now on referred to as EF, can take care of most of the data-access code usually needed for accessing the database. Since a new database was needed, the EF is investigated for creating and managing the new database.

Also investigated is the Open Data Protocol for Restful Web API's, also known as the OData protocol. During this research, a library for ASP.NET Core was found. This library can be used to apply the OData practices when building a Web API.

4.4.1 The Entity Framework as database

The official Microsoft documentation states that: "With EF Core, data access is performed using a model. A model is made up of entity classes and a derived context that represents a session with the database, allowing you to query and save data." [23].

Using the Entity Framework, your database tables are created from your entity classes. This gives you the possibility to create a database simply by writing C# entity classes. Another possibility is to generate a model from an existing database. This means you create a database first, and let the EF generate a model with entity classes based on your database.

Being able to query and save data to the database, without having to write your own SQL queries, can simplify the project a lot. This also reduces the possibility of syntax errors when querying to the database. Due to this, the decision was made to use the EF for building the new database and the communication with the Web API. The EF has also been used to generate data models for the old database and communication between the old database for synchronizing the databases.

4.4.2 The Open Data Protocol for Restful API's

As the OData website states: "OData (Open Data Protocol) is a standard that defines the best practice for building and consuming RESTful APIs." [25]. The protocol itself is very big and has many rules and models. For this project, the most relevant are the URI conventions and query options. With the querying in the OData protocol, you can select specific data simply by adding query strings to the API URL. In case the reader is interested in the exact details of the OData protocol, it can be found on their website [25].

While looking into OData, a library for ASP.NET Core by OData was found. This library can be used to support the OData query syntax in your Web API. The biggest advantage of this is the possibility to query your API methods. Querying can be used to filter lists, select specific data fields, expand with related entities etc. Some simple examples of these queries are:

- api/crashreport?$filter= projectid le 5
  - This example returns all CrashReports where the connected project's Id is lower or equal to 5.
- api/crashreport?$select= id, projectid
  - This example returns only the Id and project Id of all CrashReports.

Besides its benefits, the OData library had its downsides as well. For one, the OData routing is different from MVC. This routing isn't automatically compatible with the Swagger documentation tool. To use Swagger, a workaround is required. When using the workaround for Swagger, the API isn't fully documented, since Swagger doesn't document all possible query options.

Although the usage of the OData library has its downsides, the ability to query our data compensates that for us. Therefore the OData library is used for building the Web API.

4.5 Answering the main research question

The main research question is: "*How can the CrashFix system be adapted to fix current issues, prevent future issues, extend the system with additional required functionality and have it maintained by Ellips software development teams?*"

As found when defining the assignment, the current unstable state of the CrashFix system is caused by both an overly-complex and outdated backend, as well as a front-end with useless and broken error pages. The best solution to this problem is creating both a new front-end web application, as well as creating a new backend. The backend would need a new database, a service to synchronize the old database with the new database, as well as a new Web API for communicating with the backend.

If the newly created application would be well-tested and stable, this should make it possible to fix the issues with the current system. The decision was made to put the focus on creating a well-tested, and therefore stable system resulting in less/no errors. This system should be expandable and maintainable by the company, so that the company can expand the system with new functionalities in the future.

Because of a large number of components in the CrashFix system in need of replacement, the project was becoming too large for an internship. Therefore the additional requirements for new functionalities were given a lower priority, as they would fall outside of the scope of the project.

Implementation 5

This chapter will first present the main architecture of the new system and its components. Besides that, the structure of the newly realized database will be given, as well as the implementation of the synchronization between the new database and the old CrashFix database. Also, the implementation of the Web API will be given. Also discussed is the design and structure of the front-end web application.

5.1 Architecture

The architecture of the new system is visualized in an architecture diagram in Figure 5.1. As seen in the architecture diagram, the system consists of 4 main components. These components are: the old database, the new database, the ASP.NET Core Web Application, and the Angular Web Application (The User Interface).

The ASP.NET Core application consists of 3 sub-components, the database synchronizer, the Web API and the Swagger API Documentation. The database synchronizer is a timed service that frequently checks for new data in the old database, converts the old data to fit the data structure of the new database, and then inserts this data into the new database. The Web API is a web-based service that handles HTTP requests and, if the request is valid, fetches the requested data from the database and returns this data in JSON format. The Swagger documentation for the Web API is a web page automatically generated by the Swagger package. This web page shows what requests the Web API handles, what those requests return, and also gives the possibility to try out those requests. Both the Web API and the Swagger documentation are approachable only from within the company's private network.

The last component, the Angular Web Application, is a web application built to replace the CrashFix website. This application has a fresh, modern, and basic user interface. It currently has the base functionalities of the old CrashFix website and can be expanded in the future to add extra functionality.

The clients, in this case, the developers of Ellips, can approach both the Angular Web Application, the Web API and the Swagger API documentation using their browser.

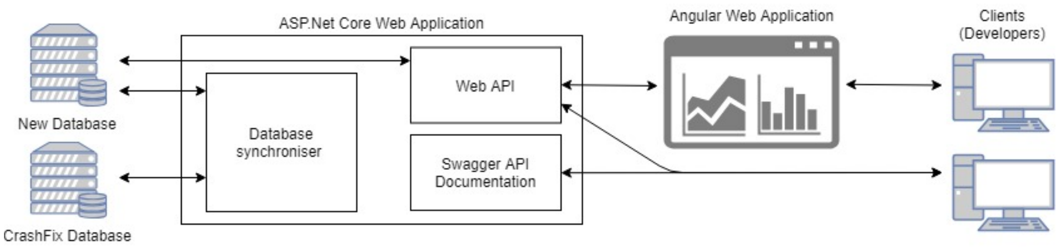


Fig. 5.1: The architecture of the new system visualized in an architecture diagram.

5.2 The database

After analyzing the old database, due to many unused tables, the decision was made to develop a new and, most importantly, less complicated database for the new system. This database includes only the tables necessary for the new application to work. Doing so brought extra work with it as well, the CrashReports are still sent to the old system and processed into the old database. To keep the new database updated, the databases has to be regularly synchronized. The implementation of the new database and the choices for that implementation can be found in the next section, as well as the chosen solution for synchronizing the databases.

5.2.1 Database implementation

The new database has been built using Microsoft's Entity Framework Core, mentioned earlier in section 4.3, and is based on the same information as the old database. The difference between the databases is that the new database only contains the data the new system needs, leaving out unused and unnecessary tables and fields from the old database. This results in a much smaller, and more important, a less complicated data structure. This leaves the new database with only 8 tables, one of which, used only for checking when the last synchronization between databases has taken place.

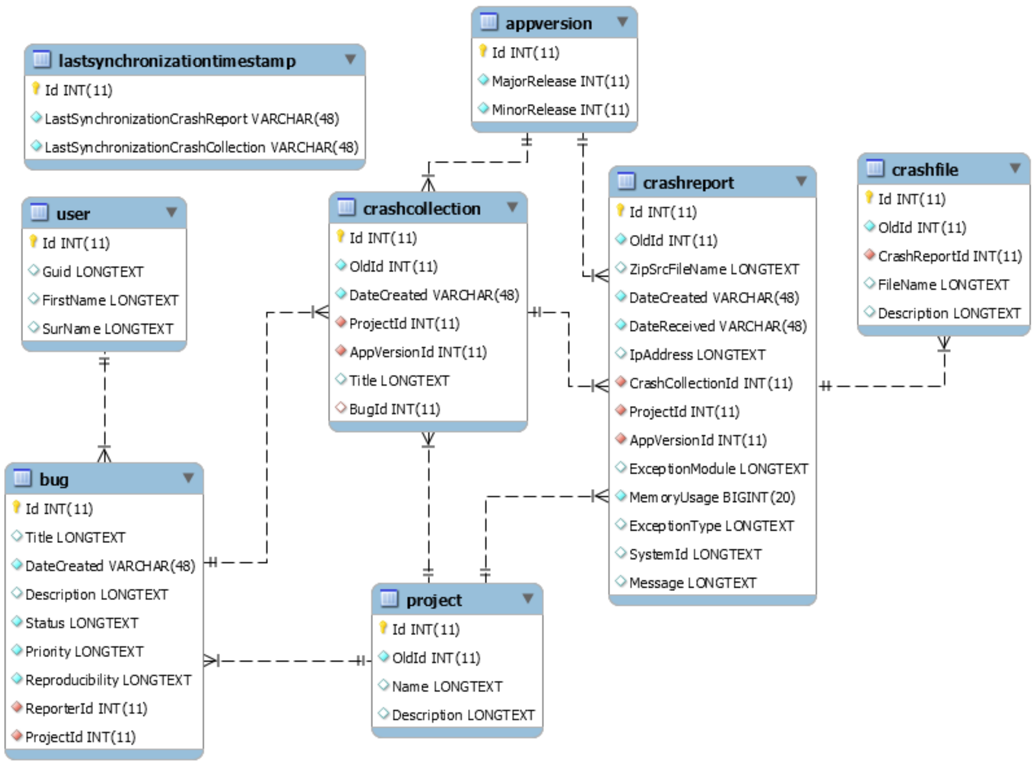


Fig. 5.2: The database structure of the new database visualized in an Entity-Relationship Model (ERM).

To clearly visualize the structure of a database and the information it contains, a model is created showing the entities, their data, and the relationship between those entities. Such representation is called an Entity-Relationship Model. The Entity-Relationship Model for the new database is given in Figure 5.2.

In this Entity-Relationship Model, you can see that the database is built around the 3 main entities: Bug, CrashCollection, and CrashReport. These entities are all related to a single Project and, except for bugs, an AppVersion. A Project, in this case, is an application in which the error has taken place. An AppVersion is a version of the project, for example: 1.1, 1.2, 2.1 etc. These AppVersions were related to Projects in the old database, but due to multiple projects having the same version numbers, that relationship was not implemented in the new database to prevent duplicate entries. The CrashFile entity is used to keep track of what files are in a CrashReport's zip file and, therefore, cannot exist without a related CrashReport. The User entity used to track the current Project and AppVersion the user was searching for and gave the ability to restrict access to selected projects. Due to this feature not being used, and the decision not to include authentication in the new system, the purpose of the User entity in the new database is solely to assign developers to bugs.

5.2.2 Database synchronization

The data from the old database that needs to be carried over to the new database is limited to: CrashReports with their related CrashFiles, CrashCollections, Projects, and AppVersions. Because the new system wasn't going to track changes made to bugs, the complexity of the old bug tracking system, and a large amount of old, unused bugs, the decision was made not to synchronize the existing bug data. Without synchronizing the bug data, there was no need to synchronize the user data either. Instead, in the new system, the users are fetched from developers' group in the active directory of Ellips domain and inserted to the database upon being connected to a bug.

To synchronize this data, the data in the old database must be compared with the data in the new database. Then, if there is new data in the old database, this data should be converted to fit the data-structure of the new database and the converted data should be inserted in the new database. The part of the old system processing incoming CrashReports will stay intact, therefore, new data will still be processed into the old database first. Due to this, the data synchronization will have to be executed frequently to keep the new database up-to-date.

The new database was built code-first using the EF. From the research in chapter 4.3.1, we discovered that one can generate a database model complete with entity classes based on an existing database. This seemed the best technology to built the database synchronizer since database access is relatively easy, less work and less error-sensitive using the EF. Because of the benefits of the EF, a database model was generated from the old database, generating entity classes for only the necessary tables to keep the program as clean as possible.

The easiest way to implement the database synchronization is to build it in the same application as the Web API, as a background task performed on a timely basis. Another option is to create a separate application for keeping both databases synchronized. The first option was used because this was easier and faster.

Creating a background task to run parallel to the Web API was a relatively simple task. In ASP.NET Core, background tasks can be implemented using the Hosted Service interface. After implementing the Hosted Service interface, the service can be registered to the Web API. Doing so, ASP.NET automatically starts the service on a separate thread when starting the application and the service will be shut down on application shut down.

The database synchronizer is built of several parts. For some entities, converting the data structure to fit the new database took relevantly much code. Therefore, the database synchronization contains separate converter classes for converting these entities from the old database to fit the data structure of the new database. The structure also contains a synchronizer class for each of the entities that will be synchronized. If new data is found when synchronizing, these synchronizers use the converters to convert the new data and then insert this converted data into the new database. The main part, which is also registered as a service on starting the application, is the PeriodicDatabaseSynchronizer. This service gets an implementation of the IDatabaseSynchronizer through dependency injection. The service itself implements a timer which calls synchronize on the IDatabaseSynchronizer every 10 minutes. The DatabaseSynchronizer implements the IDatabaseSynchronizer interface. On calling Synchronize, the DatabaseSynchronizer calls synchronize on each of the entity-specific synchronizers. As shown in Figure 5.3, the data converters can be found the Converters folder and the entity-specific synchronizers can be found in the Synchronizers folder.

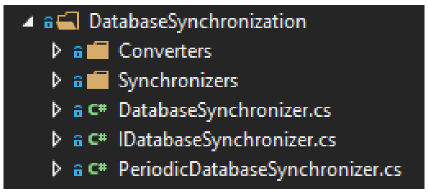

```
DatabaseSynchronization
    Converters
    Synchronizers
    DatabaseSynchronizer.cs
    IDatabaseSynchronizer.cs
    PeriodicDatabaseSynchronizer.cs
```

Fig. 5.3: The project structure of the database synchronization.

5.2.3 Database synchronization - Testing

The database synchronization is a fully unit tested component. To test the database synchronization, both databases were needed. In order to test this without using the actual databases, the EF was used to create a temporary in-memory SQLite database for each of the databases. These in-memory databases are created based on the database models and their entity classes. These databases are created and filled with fake data when a test needs them and torn down when the test is finished. The reason to recreate both databases after each test is to keep the databases clean and independent of other tests.

The converter classes converting the data from the old database to fit the new data structure were tested differently. These classes are independent of where the data comes from, and can therefore just use a list of the specific entities it converts.

5.3 The Web API

The Web API, as stated earlier, is built using Microsoft's ASP.NET Core. The Web API can be approached by applications within the Ellips network, such as the Angular Web Application or just a regular browser like Google Chrome. The Web API follows the standard HTTP protocol. Through the Web API, one can make several kinds of HTTP requests. One can make a GET request to retrieve data from the database, a POST request to create objects in the database, a PUT request to update complete entities in the database, a PATCH request to update parts of entities or a DELETE request to delete objects in the database. The Web API response contains an HTTP Status Code and, if applicable, the requested data in JSON format. Basic examples of these status codes are 200 OK, 201 Created, 400 BadRequest, 500 InternalServer-Error etcetera. Note that not every request is available for each entity, for Project, for example, only GET requests are possible.

The structure of the ASP.NET Core application is given in Figure 5.4. The structure is separated into Controllers, Data and Utils. The Controllers folder contains the controllers of the Web API. These controllers handle the HTTP requests sent to the Web API.

The Data folder contains a folder CrashFixDatabase containing the database model complete with the used entities generated using the EF reverse engineering option on the old CrashFix database. The DatabaseSynchronization folder contains all classes regarding the database synchronization elaborated in the previous chapter. The HiveDatabase folder contains the database model and its entity classes of the new database. The LDAP folder contains the classes for fetching the domain users from the developers' user group in the company's private domain using Microsoft's DirectoryServices package. The Utils folder contains all utilities of the application such as global constants or specific property converters.

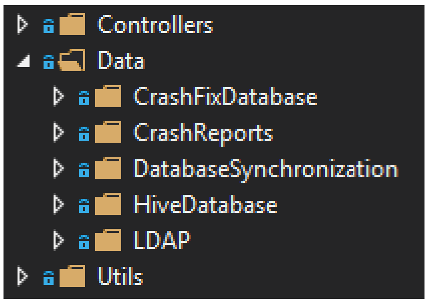


Fig. 5.4: The project structure used in the ASP.NET Core application.

In the next sections, the API controllers will be further discussed, including the implementation of the ODataControllers and the separation between the controllers and the database access. Also, the implementation of the Swagger API documentation will be given. The last section will elaborate on how the Web API was tested.

5.3.1 Controllers/Data Access

In the Web API, each entity has its own controller, excluding the LastSynchronizationTimeStamp and CrashFile Entities. The LastSynchronizationTimeStamp has no use for a controller because it is only used for the database synchronisation and therefore never has to be accessed from outside the Web API. The only situation where a CrashFile is requested is when viewing a CrashReport. In this case, when requesting the specific CrashReport, the related CrashFiles can be expanded to the request as children objects. Therefore, CrashFile doesn't need a Controller.

All the API's controllers inherit from the ODataController base class of the OData package. Using the ODataController, an API method can be made queryable using the EnableQuery attribute. When a method is queryable, one can apply query string options to their HTTP request. Note that this is only applicable to GET requests. The available query options are listed below:

- OrderBy ($orderby): The OrderBy query option can be used to order the entry set by specific properties. The order is ascending by default but can be set to descending.
- Top ($top): The Top query option can be used to select the top entries of the entry set. Example: "*/crashreport?$top=5*" returns the first 5 CrashReport entries from the database.
- Skip ($skip): The Skip query option can be used to skip the first entries of the entry set. Example: "*/crashreport?$skip=5*" returns the complete set of CrashReport entries starting by the sixth CrashReport entry.

- Filter ($filter): The Filter query option can be used to filter the given entry set by specific properties.
- Expand ($expand): The Expand query option can be used to expand the requested entities with related entities. Example: "*/crashreport?$expand=crashfiles*" returns the complete CrashReport entry set. The CrashReports in the entry set will have their related CrashFiles in a list as additional property in the response data.
- Select ($select): The Select query option can be used to select specific properties of requested entities. Example: "*/crashreport?$select=id,datecreated*" returns only the Id and DateCreated properties of all CrashReport entries.

Multiple query string options can be combined in a single HTTP request to query more specific data. An example of how to apply this is:

- */crashreport?$filter=id le 50&$expand=project,appversion&$orderby=id desc*

The request in the example above will return all CrashReports with an Id Lower or Equal to 50 (*$filter=id le 50*), expanded with the related Projects and AppVersions (*$expand=project,appversion*), ordered by their Id in an descending order (*$orderby=id desc*). For the exact documentation on the possible query options, one can visit the OData 2.0 URI Conventions documentation on the following web page: https://www.odata.org/documentation/odata-version-2-0/uri-conventions/.

The bare */[controller]* HTTP GET request returns the complete *DbSet* of the entity directly from the database through the EF. A different and often used implementation, which was also considered here, is the use of the Repository Pattern. If implemented, this would mean that we'd create a layer of repositories between the Web API controllers and the EF classes to help insulate our application from changes in our data structure. This was not implemented for 2 reasons. For one, the way it's implemented now, the OData query options can access the database queries itself. Doing so, the OData controller applies these query options when the EF queries the database, rather than on an already fetched set of data stored in the application's memory. This is more efficient in terms of memory usage. The second reason is that this is extra work and simply wasn't needed for our application, because the new data structure is very simple.

5.3.2 Swagger API documentation

The Web API is documented using a package called Swashbuckle.ASPNetCore in the Web API application. This package enables the application to automatically generate Swagger API documentation based on the Web API's routes, controllers and models. This package comes with 2 possible user interfaces that were both considered for this Web API. These interfaces are called SwaggerUI and Redoc. SwaggerUI is the more traditional Swagger user interface. Redoc is a newer interface. The Redoc interface is supposed to be easier navigatable and should give more information in a more structured manner. Unfortunately, the Redoc interface presented by the package is not fully developed yet and misses the key functionality of being able to try out HTTP requests and seeing their responses. Due to this, the SwaggerUI interface is used for this Web API. For full details on the research about the API documentation and the difference between the two interfaces, see Chapter 7 of Appendix B: Research document.

5.3.3 Testing

The Web API is mainly tested by integration tests, and for some methods, unit tests. To realize this, a fake data source was needed to replace the connected database. This was performed using 2 separate implementations. Most of the API methods only fetched data from a single entity's data set. To test these methods, that specific data set was mocked using the Moq library. API methods accessing multiple data sets in its process were becoming complicated to test using mocking. For those tests, we used the EF to create an in-memory SQLite database based on our database model and its entity classes and filled this database with mock data. To keep that fake database clean and independent of the other tests using it, the database is created before each of those tests, and broken down after every test.

The query options of the Web API have not been tested with integration or unit tests. This is partly because there are simply too many situations to test for this. The other reason is that this functionality comes from an official library and should be tested by the creators of the library.

5.4 The front-end web application

The new front-end web application is built in Angular. This website is built to become a replacement for the old CrashFix system's website. The application was initially created in Visual Studio Code. Developing an Angular application in Visual Studio Code, many actions are performed using commands in the built-in terminal and debugging TypeScript code is limited and requires a learning curve. Due to the company's developers not being familiar with this development environment and thus preferring the application running in their familiar Microsoft's Visual Studio 2017 working environment, this project was converted to an ASP.NET Core application using the ASP.NET Core Angular template. An advantage of this is that it allows the user to run the application using a simple button instead of using commands in the terminal. Unfortunately, the TypeScript unit tests cannot be run using Visual Studio's Test Explorer and still have to be run using commands in the IDE's terminal.

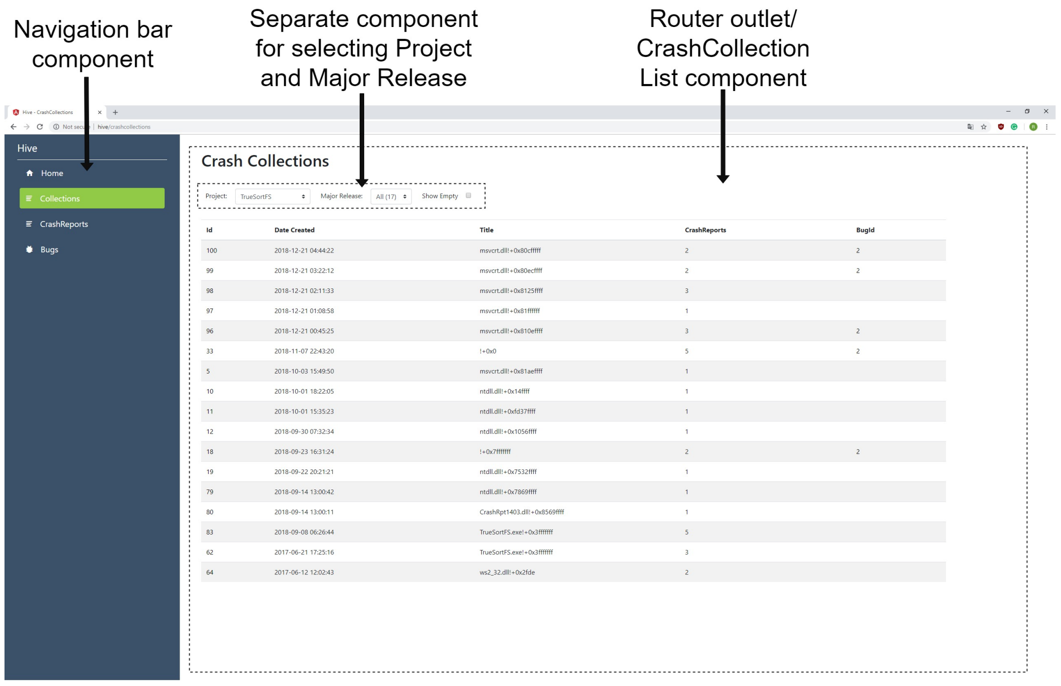
In this section, we will go further into the structure of the Angular application, including the project structure and the use of components. Besides that, the design of the User Interface will be further elaborated. Also mentioned, is how the Angular application was tested.

5.4.1 Structure

The Angular application is mainly built from components. In this case, the application is set to display at least 2 components at once. These components are the Navigation Bar and the Router Outlet. The navigation bar makes it possible to navigate to the pages of the application. The Router Outlet has defined a set of routes. These routes are connected to certain components, the */crashreports* route, for example, is connected to the CrashReportList component. Navigating to that specific route will display the CrashReportList component in the place of the Router Outlet.

Within a component, it is possible to view another component. This means that, instead of building the same mechanism multiple times because it is used in multiple components, you can build a separate component for the mechanism and reference it in the components that need it. In our case, this was applicable to the selection of

Project and Version when filtering the lists of Bugs, CrashReports, and Collections. There are 3 components viewing those lists, one for every entity. Instead of building the mechanism for selecting the Project and Version 3 times, this mechanism has its own component and is used in each of the list components. In this case, an extra service class was created to distribute the information of what Project and Version were selected. This was necessary to get the information from the selection component to the list components. To give a more visual perspective on this subject, one can view the edited screenshot given in Figure 5.5.



Navigation bar component
Separate component for selecting Project and Major Release
Router outlet/ CrashCollection List component
Hive
Home
Collections
CrashReports
Bugs

Crash Collections

Project: TrueSortFS Major Release: All (17) Show Empty

Id	Date Created	Title	CrashReports	BugId
100	2018-12-21 04:44:22	msvcrt.dll!+0x80cfffff	2	2
99	2018-12-21 03:22:12	msvcrt.dll!+0x80ecffff	2	2
98	2018-12-21 02:11:33	msvcrt.dll!+0x8125ffff	3	
97	2018-12-21 01:08:58	msvcrt.dll!+0x81ffffff	1	
96	2018-12-21 00:45:25	msvcrt.dll!+0x810effff	3	2
33	2018-11-07 22:43:20	!+0x0	5	2
5	2018-10-03 15:49:50	msvcrt.dll!+0x81aeffff	1	
10	2018-10-01 18:22:05	ntdll.dll!+0x14ffff	1	
11	2018-10-01 15:35:23	ntdll.dll!+0xfd37ffff	1	
12	2018-09-30 07:32:34	ntdll.dll!+0x1056ffff	1	
18	2018-09-23 16:31:24	!+0x7fffffff	2	2
19	2018-09-22 20:21:21	ntdll.dll!+0x7532ffff	1	
79	2018-09-14 13:00:42	ntdll.dll!+0x7869ffff	1	
80	2018-09-14 13:00:11	CrashRpt1403.dll!+0x8569ffff	1	
83	2018-09-08 06:26:44	TrueSortFS.exe!+0x3fffffff	5	
62	2017-06-21 17:25:16	TrueSortFS.exe!+0x3fffffff	3	
64	2017-06-12 12:02:43	ws2_32.dll!+0x2fde	2	

Fig. 5.5: A screenshot of the Angular application's CrashCollections list page.

There are a large number of components and services in the application. To keep the project structure easy to read and navigate, the components and services of the application have their own folder. Besides the project and version selection service, the services in this application are classes that contain methods to make HTTP requests to the Web API. These methods are used by the components to fill the user interface with information and to perform other administrative tasks, such as deleting or updating entities. The Models folder contains all data models used in the application. The Routing folder contains a module that handles all routing within the application. The Utils folder contains utilities globally used in the application. The only thing the app component files in the app folder do is define the base structure of the website, dividing each page of the website into the navigation bar and the router outlet as explained in the previous section. The app module file

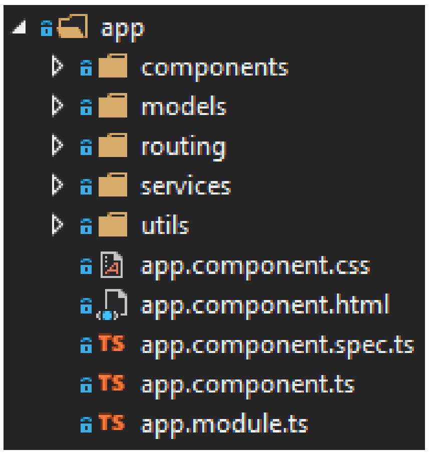


Fig. 5.6: The project structure of the Angular web application.

registers all of the components and services within the application. This module also imports all used external modules. The project structure can be found in Figure 5.6.

5.4.2 User Interface design

At first, one of the more important parts of this internship seemed to be designing a new user interface for the CrashFix system. This would have included design sessions with employees of the company, in order to improve the quality of the designed UI. However, due to the project becoming bigger, the design of the user interface gained less priority, giving more priority to developing a reliable and well-tested application. Because of this, no design sessions were performed, and the user interface has been kept to a simple interface.

In Figure 5.5 an image of the page viewing the list of CrashCollections was given. As explained earlier, the interface is rather simple. The left of part of the page contains the navigation bar. The navigation bar contains the links to the home page and the pages viewing the lists of entities. The right part of the page contains the component which is linked in the current route, in this case, the CrashCollection List component (*/crashcollections* route).

5.4.3 Testing

As stated in my research about Angular, the Jasmine testing framework was used to test the Angular application. The framework was used to test both the components and the services in the Angular application. In the components tests, things that are tested are: calls being made to a service, the handling of the response data of these services, data bindings etc. When testing the HTTP services, we test if a specific request is made when calling a method. All these tests can be run using the command prompt. When doing so, Jasmine opens a Google Chrome browser displaying the results of the tests found within the project. While the browser is kept open, saving changes to the application's code will make jasmine re-run all unit tests. This feature has been useful for the Test-Driven approach that was used when building the application.

6 Conclusions and recommendations

The main research question was partly answered when defining the problems with the CrashFix system and the requirements for the new system. The current unstable state of the CrashFix system is caused by both an overly-complex and bugged backend, as well as a front-end with useless and broken error pages. The best solution to this problem is creating both a new front-end web application, as well as a new backend consisting of a Web API, new database, and a service to frequently synchronize the new database with the data from the old database. This left only one part of the old system being used, the processing of the incoming crashes. To prevent future issues in the new system and keep it expandable and maintainable by the Ellips software development teams, the new application had to be well-tested and easy to read.

Due to the large number of components in need of replacement, the project's scope was becoming too large for the available time. Therefore, the focus was set on creating a solid, well-tested backend and front-end, setting less focus on the development of new functionalities and the UX design of the web application.

The new backend application has ultimately been created using Microsoft's ASP.NET Core framework. This application uses the Entity Framework Core to create the new MySQL database, based on a database model with entity classes. Also located within this application, is a service for synchronizing the new database with the incoming data of the old database. Besides the database structure and synchronization in this application, the application functions as a Web API, handling HTTP requests from the front-end. The new front-end application has been developed in a separate application and is built using the Angular framework.

The optimal situation would be to completely replace the old CrashFix system. Therefore, the only recommendation here is to eventually create a new application, or expand one of the new applications, to collect and process the incoming crashes directly into the new database. Doing so would replace the last remaining part of the old CrashFix system, as well as the database synchronisation service located in the new system.

Evaluation

This project has taught me many interesting things. On the technical side, I learned how to create a Web API with ASP.NET Core. Besides that, I learned how to use the Entity Framework, which in turn became of great help for creating the new database and communicating between both databases. I also got more familiar working with the Angular framework and learned how to create unit tests within this framework.

I think one of the most interesting, but at the same time, also the most unexpected things I learned was about the OData protocol and the existence of the OData package for ASP.Net Core. The ability of this package, in combination with the Entity Framework, to turn a simple HTTP GET method into an API method that can be queried to enhance the request, was very powerful.

Another important learning moment, for me, was when I heard that the company wanted to have the Angular application run in the Visual Studio 2017 environment. At the time, I had already built a relatively large part of the Angular application in Visual Studio Code. Due to this, converting the Angular project to fit the Angular template in ASP.NET Core took me relatively much time to get working. I could have prevented this from happening by better checking the company's interests and by better communicating with my company supervisor.

Regarding working methods, I learned about Test-Driven Development. Working test-driven was often hard, and I could not always apply this as well as I wanted. But when working test-driven, I looked at my applications from a different perspective, by first figuring out how to test something, instead of first building what I needed to test.

I have also learned much about writing a report. Since this is the first time I have written an internship report, I didn't have much knowledge about writing one. During the writing of this report, I have gained a lot of helpful feedback from my supervisors, my friends, and my family.

Personal learning objectives

Before the start of the internship, I had to define personal learning objectives on which I wanted to spend extra attention. These learning objectives are my research skills, my knowledge of software architecture and my communication skills.

Since a large amount of research was performed during this internship, it can be stated that extra attention has been spent on research, and, therefore, improving my research skills.

During the research on the CrashFix application, I have looked into the architectural structure of the system. Besides this, I have spent time creating the architecture of the new applications.

Regarding communication, I have communicated with my colleagues at Ellips to create a list of issues with the CrashFix system, find out what the most important functionalities of the system are, and what new functionalities the company would like to see.

Bibliography

[1] Kent Beck. *Test Driven Development By Example*. Pearson Education Inc, 2003.

[2] Ellips B.V. *Ellips official website*. URL: `https://ellips.com/nl/`.

[3] Ellips B.V. *Ellips official website - history*. URL: `http://ellips.com/history/`.

[4] Ellips B.V. *Ellips image of TrueSort program and sorting machine*. URL: `https://ellips.com/wp-content/uploads/2017/12/grading-ellips-next-level-grading.jpg`.

[5] SourceForge.net - zexspectrum. *CrashFix official website - architecture overview*. URL: `http://crashfix.sourceforge.net/doc/html/architecture_overview.html`.

[6] Medium.freeCodeCamp - Jacek Schae. *A comparison of front-end frameworks with benchmarks*. URL: `https://medium.freecodecamp.org/a-real-world-comparison-of-front-end-frameworks-with-benchmarks-2018-update-e5760fb4a962`.

[7] Codeburst - Natalia Kharchenko. *Article comparing the Angular and React web development frameworks*. URL: `https://codeburst.io/react-vs-angular-who-gets-the-final-say-c3d3ef8c3dd0`.

[8] Medium - TechMagic. *Article comparing Reactjs, Angular 5 and Vuejs*. URL: `https://medium.com/@TechMagic/reactjs-vs-angular5-vs-vue-js-what-to-choose-in-2018-b91e028fa91d`.

[9] Google. *Angular's Tour of Heroes tutorial*. URL: `https://angular.io/tutorial`.

[10] Google. *Angular documentation on testing Angular applications*. URL: `https://angular.io/guide/testing`.

[11] Google. *Angular documentation on using the HttpClients for making HTTP requests*. URL: `https://angular.io/guide/http`.

[12] Google. *Official Angular documentation website, many pages have been used for creating the new web application*. URL: `https://angular.io/docs`.

[13] Facebook. *Reactjs's official tutorial on getting started with Reactjs presenting recommended toolchains*. URL: `https://reactjs.org/docs/create-a-new-react-app.html`.

[14] DZone - Ankit Sharma. *Tutorial on creating an ASP.NET Core Web API application using the standard Reactjs template in Visual Studio 2017*. URL: `https://dzone.com/articles/aspnet-core-crud-with-reactjs-and-entity-framework`.

[15] Microsoft. *Tutorial on unit testing javascrip in Visual Studio 2017*. URL: `https://docs.microsoft.com/en-us/visualstudio/javascript/unit-testing-javascript-with-visual-studio?view=vs-2017`.

[16] Microsoft. *Tutorial on generating Swagger API documentation using the Swashbuckle package*. URL: https://docs.microsoft.com/en-us/aspnet/core/tutorials/getting-started-with-swashbuckle?view=aspnetcore-2.1&tabs=visual-studio.

[17] Rebilly. *Official documentation of the ReDoc user interface for Swagger API documentation*. URL: https://github.com/Rebilly/ReDoc.

[18] Swashbuckle. *Swashbuckle git repo, Documentation on the configuration of the package*. URL: https://github.com/domaindrivendev/Swashbuckle.AspNetCore.

[19] Vaibhav Dubey. *Article on Test-Driven Development*. URL: https://blog.usejournal.com/test-driven-development-understanding-the-business-better-9c4cae4cb990.

[20] Vaibhav Dubey. *Image used in Chapter 8 of the Research Document to clarify the Test-Driven Development cycle*. URL: https://cdn-images-1.medium.com/max/1600/1*tZSwCigaTaJdovyWlp5uBQ.jpeg.

[21] Microsoft. *Tutorial on creating a Web API with ASP.NET Core*. URL: https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-web-api?view=aspnetcore-2.1&tabs=visual-studio.

[22] Microsoft. *Tutorial on creating unit tests in ASP.NET Core using the NUnit package*. URL: https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-with-nunit.

[23] Microsoft. *Microsoft's official documentation on the usage of the Entity Framework Core (Multiple pages of the documentation have been used.)* URL: https://docs.microsoft.com/en-us/ef/core/.

[24] Microsoft. *Official documentation on using an SQLite in-memory database for unit testing*. URL: https://docs.microsoft.com/en-us/ef/core/miscellaneous/testing/sqlite.

[25] OData. *OData official website - home*. URL: https://www.odata.org/.

[26] OData. *Basic tutorial in understanding the OData protocol*. URL: https://www.odata.org/getting-started/understand-odata-in-6-steps/.

[27] OData. *OData v2.0 URI Conventions official documentation*. URL: https://www.odata.org/documentation/odata-version-2-0/uri-conventions/.

[28] Microsoft - blogs - Sam Xu. *Tutorial on the usage of the OData package for ASP.NET Core*. URL: https://blogs.msdn.microsoft.com/odatateam/2018/07/03/asp-net-core-odata-now-available/.

[29] Microsoft. *Microsoft documentation used for extracting files from .zip files in the Web API*. URL: https://docs.microsoft.com/en-us/dotnet/standard/io/how-to-compress-and-extract-files.

Appendix A: Project Plan

In this appendix, the Project Plan, can be found. The Project Plan document was was created at the start of this project.

Project Plan

CRASH REPORT SYSTEM

Esp 300, 5633 AE Eindhoven

Date:	
Version:	1.0
Status:	Waiting for feedback
Author:	Rik van Spreuwel

Version

Version	Date	Author(s)	Changes	Status
0.1		Rik van Spreuwel	Created document	In progress
0.2		Rik van Spreuwel	Document finished	Waiting for feedback
1.0		Rik van Spreuwel	Document finished	Waiting for feedback

Distribution

Version	Date	To
0.2		
1.0		

Glossary

Term	Definition
Crash Report	When a TrueSort program crashes, the program sends a report to the CrashFix system. We call this report a Crash Report. A crash report contains information regarding the crash – examples of this are: where in the code has the crash taken place, what time has it taken place etc.
Collection	A Collection is a bundle of crash reports that belong to the same bug/problem.
TFS	TFS stands for Team Foundation Server. This is a Microsoft software development tool.

Contents

1. The Assignment

1.1 Context

The client for this project is Ellips. The company was founded in 1989 and is focussed on the development of optical sorting technology for the vegetable- and fruit market. The software which the company sells and maintains is called TrueSort and works on any sorting machine, from any brand. For more information about the company and their product, visit: www.Ellips.com.

Within the company they currently work following the Scrum-methodology. This means they work with sprints of 2 weeks and every morning they have a daily stand-up at 10 am. The company uses Team Foundation Server (TFS) for their software development process. This system provides version control, Agile tools (Scrum board) and is used for continuous integration.

The company is currently using a web application called CrashFix for collecting and analysing crash reports coming from their TrueSort product. This is an open source application which the company's developers use for tracking and solving bugs. After some time, the open source application wasn't being maintained anymore. As a result of this, the application doesn't get any new functionalities or updates. This caused it to become less stable, which causes frustration among the developers.
As an attempt of solving this issue, they set up a different open source application called Exceptionless for future applications. Despite the fresher interface, the new application still missed features the company would like to have. For example, they missed the possibility to sort and filter crash reports and collections. Also they would like to be able to make a "bug" object from a collection, which is put in the backlog of the TFS system's scrum board. These needs resulted in this internship assignment (for the assignment description, see 1.2 Assignment Description).

The architecture of the current CrashFix system is visualized in a diagram in figure 1. The sorting machines (Clients) are located at the customers. When a crash takes place somewhere in the sorting machines' system, the data of that crash is bundled and send to the CrashFix service. This service collects the incoming crash reports and saves them in the database. The service can only be accessed by the web application, not by users directly. The web application is a PHP website that allows CrashFix users to perform administrative and daily usage tasks through a web browser application.

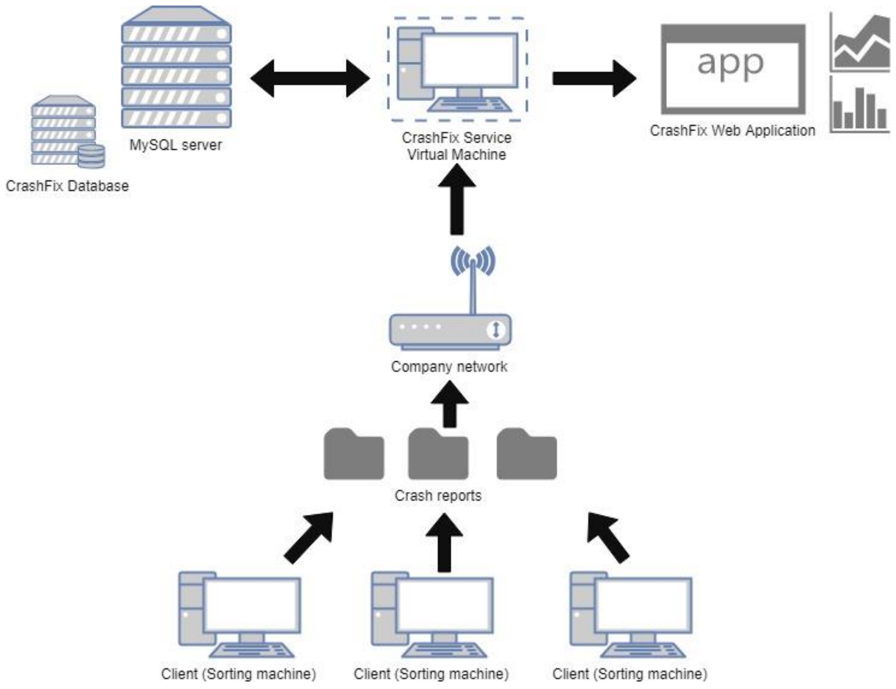


Figure 1: The current CrashFix system visualized in a diagram.

1.2 Assignment description

1.2.1 Initial assignment

The assignment is improving the user interface of the outdated CrashFix system. In this assignment the intern has to do research on how the current CrashFix system works and what the possibilities are for making a new user interface. A few proposed solutions are:

- Making a new front-end which can be added to Exceptionless as an extension in order to combine the systems.
- Making a completely new front-end using the old CrashFix system.

In the first week of the internship the intern quickly discovered that Exceptionless doesn't support such extensions, scrapping the first solution off the list. Also we found out that the CrashFix service is closely attached to the PHP web application, making it not very reusable. That also partly scraps the second solution off the list. Research showed that the CrashFix database is still fully functional and is approachable by other applications. We chose to reuse the CrashFix database, but build a new Web API that communicates with it. This resulted in the following solution: Making a new Web API and front-end for a new web application using the old CrashFix database. For more information about this research, see 1.6 research questions.

1.2.2 Final assignment

The research findings of week one resulted into the following assignment: Making a new Web API and front-end for the old CrashFix server. The front-end will fetch the data from the old CrashFix server through the new Web API. The new Web API will make a connection with the existing database and pass the data to the front-end through API calls. Because the company is familiar with C#, the Web API will be made in Microsoft's ASP.Net/ASP.Net Core. The most appropriate technology for building the front-end will be decided by the intern through further research.

The company values the quality, reliability and expandability of the system. That's why it's important that the intern tests the system well (see 4. Test approach and configuration management). It's also important that the intern makes clear documentation of the Web API. The realization of the project will be Test-Driven, to apply this, the intern will do a research on Test-Driven Development.

1.3 Goal of the project

The goal of this project is to give the company's developers better insight on what bugs are in their products. The application is supposed to make it easier for the developers to track and solve those bugs.

1.4 Boundary

Included in the project:	Not included in the project:
1 Development of the Web API	1 Development of a database (old CrashFix database will be used).
2 Development of the front-end	2 The collecting service of the CrashFix service.

Technology boundaries:

- Version control will be managed by company's Team Foundation Server.
- The Web API will be developed in ASP.Net or ASP.Net Core.

1.5 Strategy

This project will be executed following the scrum-methodology. We will work with sprints of 2 weeks and every morning at 10 am there will be a daily stand-up. This gives the company supervisor a clear idea on what the intern is doing and the intern will learn how scrum is applied within the company. At the end of every sprint there will be a retrospective to reflect on what was done in the sprint.

The intern will first do research on what can be tested when making the Web API and how we can test that. He will also look at what API documentation tool he will use for the new system. After that, he will do research on what technology to use when making the front-end and how to test the front-end when using that technology.

When the intern is done with his research, he will start building the application. He will work in a test-driven manner during this project, to do this, he will do a research on Test-Driven Development. Working Test-Driven, when building the application, the intern will build tests first and functionality second. Furthermore, he will work with user stories. When implementing an user story, he will have to finish the user story in order to continue to a next user story. A user story is finished when it meets the Definition of Done conditions, which are:

- The functionality works, both front-end and Web API.
- The build Web API methods are documented.
- All tests pass and there is a code coverage of more than 80%.

1.6 Research questions

The first research questions for this project were answered in week one, these are listed below:

1. How does the current CrashFix system work?
2. Can the new front-end be made as extension of Exceptionless?
3. What are the issues with the current CrashFix system?
4. What are the requirements for the new system?

The first research question has been answered by asking the company supervisor about how the system works (Field). Besides that the intern looked at the product's online documentation (Library).

The second research question has been answered by searching the internet for existing solutions and looking at the Exceptionless documentation (Library). It soon became clear that creating front-end extensions to the application wasn't supported.

For the third research question, the intern asked several colleagues what problems they encountered when using the program (Field). The intern also gained access to the system, so he could get familiar with the program and see the problems for himself (Lab).

The fourth research question has been answered by consulting the company supervisor and one of his colleagues to make a list of requirements (see 1.7 Requirements of the assignment). The requirements will may change during the project, due to changing needs of the company.

With the answers to the first four research questions, the assignment got a bit more concrete (see 1.2.2 Final assignment). Based on the additional information I drafted the following research questions:

5. What technology is most suitable for building the front-end?
    a. What technologies are most relevant for making a front-end web application?
    b. How can you test the logic and UI within these technologies?
    c. What technology is best suited for this project?

6. What is the best tool for documenting the Web API?
    a. How do I implement this within my project?
7. How do I apply Test-Driven Development within my project?
    a. What exactly is Test-Driven Development?
    b. How do I apply it to my project?
8. How can I link my application to TFS, in order to make a bug in the TFS system using my application?
    a. What problems does the company encounter not having this feature?
    b. What is the value of implementing this feature?
    c. Is it possible to make bugs in a TFS digital scrum board using a C# application?
    d. How can I implement it in my application?

The most suitable technology for building the front-end (question 5) will be found by studying articles and literature on the internet about front-end frameworks (Library). When the options have been narrowed down, the intern will make a few Proof of Concepts using the remaining frameworks to see what framework is the clearest, most structured and most suitable for our application (Lab). This research will determine what front-end framework will be used.

Regarding the API documentation tool (question 6), the company wants to use Swagger. Within Swagger you can choose between two interfaces. These interfaces are SwaggerUI and Redoc. The intern will experiment with both interfaces to see what interface is the clearest for the reader and will be used for the project. The result of this research is whether SwaggerUI or Redoc will be used for the API documentation, the final decision on this will be made in consultation with the company supervisor.

The research on Test-Driven Development (Question 7) will be performed by reading articles on the internet and by reading books on this subject, provided by the company (Library). Also the intern will try to apply the technique in the Proof of Concept projects (Lab).

How to link my application to TFS (Question 8) will, if the intern has the time, be examined later in the project. For this research, the intern will search on the internet if this has been already been achieved by other developers, and if so, how they have done it (Library). After that he will first make an proof of concept of that implementation. If that works, he will implement it into the new system (Lab).

1.7 Requirements of the assignment

In consultation with the company supervisor, the intern created a list of functionalities. These functionalities have been phrased in user stories. These user stories have been prioritised by High, Average and Low. Within this system there is only one actor, the users, which are company developers. The user stories are listed below:

1. As a user, I want to be able to view the details of a crash report, so that I can analyse it (High).
2. As a user, I want to view a list of all crash reports, so that I have a better overview of what crashes there are (High).
3. As a user, I want to view a list of collections, so that I have a better overview of what collections there are (High).
4. As a user, I want to be able to open a collection, so I can view all crash reports within the collection (High).
5. As a user, I want to be able to sort crash reports, so that I can get a better overview of all crash reports (Average).
6. As a user, I want to be able to filter crash reports, so that I can get a more specific list of crash reports (Average.)
7. As a user, I want to be able to sort collections, so that I can get a better overview of all collections (Average).
8. As a user, I want to be able to filter collections, so that I can get a more specific list of collections (Average).

9. As a user, I want to be able to create a bug in both the TFS-system and crash report where they are connected to each other (Low).

When developing user stories, there are quality requirements that need to be taken into account, these are the following:

- The API documentation must be kept updated.
- The functionalities must be tested, to keep the project bug free (tests first, functionality second).

The most important non-functional requirements are:

- Expandability, the system must be expandable for the company's changing requirements.
- Maintainability, the system must be easy to maintain.
- Reliability, the system may not crash and should not have any bugs.
- Usability, the system must be easy accessible for the company's developers and it must be easy to use.

The non-functional requirements must always be taken into account during the project. The code must be clean and readable, so colleague's that have to work with it in the future can understand and maintain it easily. The reliability will be maintained by testing all functionalities and logic in the code. To take the usability of the system into account, there will be design sessions with the company supervisor and some of his colleague's.

1.8 Final products

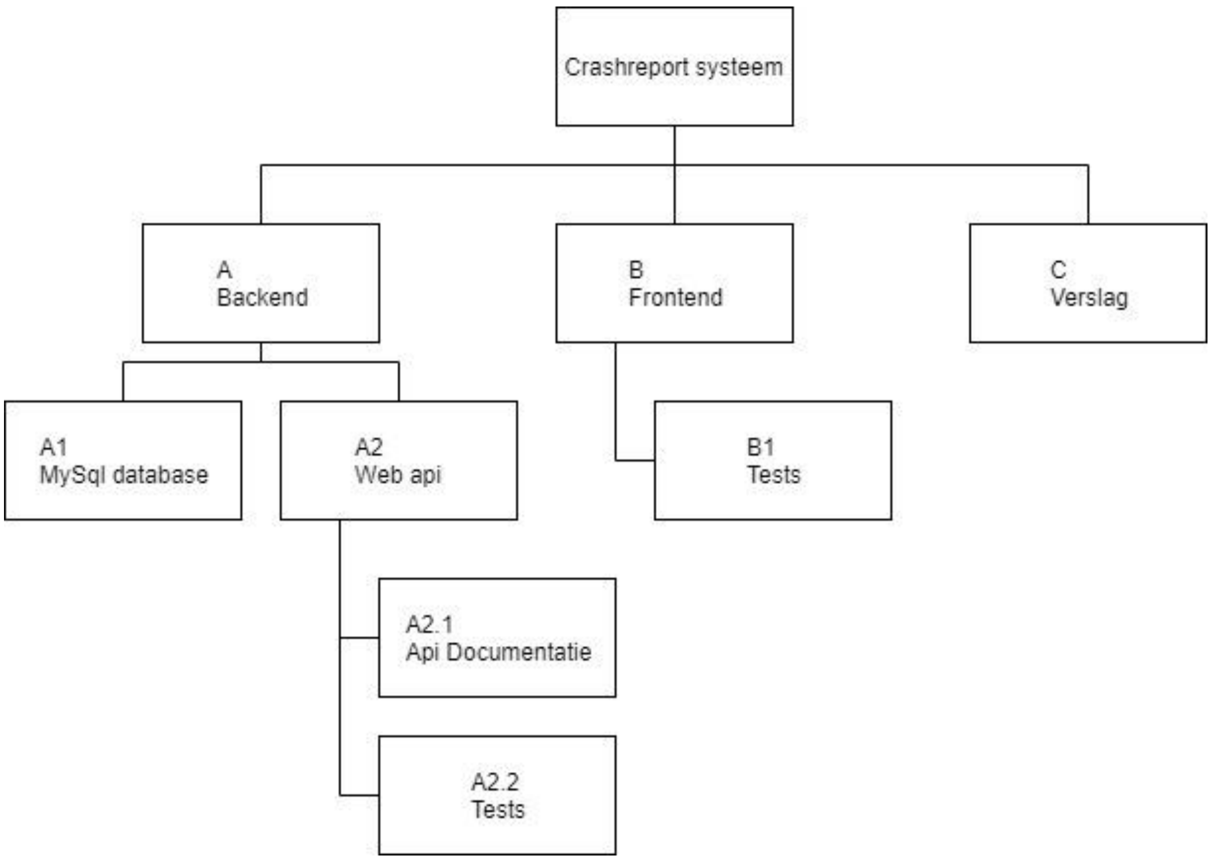


Figure 2: Project Breakdown Structure (PBS)

The final product will be a crash report system. This system will consist of a partly new backend (A) and a completely new front-end (B). The backend is divided into a database (A1) and a Web API (A2). The database will be the existing CrashFix system. The Web API has 2 sub products, API documentation (A2.1) and Tests (A2.2). The front-end will have Tests as a sub product (B1). The last final product will be the internship report (C). This structure is visualised in the Project Breakdown Structure (PBS) in figure 2.

2. Project organization

2.1 Team members

Name + phone number + e-mail	Role	Availability
	Intern.	5 day a week working on the project.
	Company supervisor.	5 days a week available for questions.
	Fontys internship supervisor.	Available for questions by email of phone.

2.2 Communication

Every day at 10 am we do a daily stand-up. This will give the company supervisor a good insight on what the intern has worked on and what he is planning to do that day. At the end of every sprint there is a retrospective. In this retrospective we look back on how the sprint went and if the target goals have been achieved. After that, we will set the goals for the next sprint.

The intern will keep a log in which he will write down what he did that day. At the end of each week, he will also write down what he plans to do the next week. This log will be send weekly to the Fontys supervisor.

3. Activities and planning

3.1 Approach

We work following the scrum-methodology, therefore the planning is split into sprints of 2 weeks. In total we have 20 weeks. The first 2 weeks were reserved for making the project plan, this gives us 9 sprints of 2 weeks left. In these sprints we have a variety of activities: Creating the project plan, doing research, implementing user stories and working on the internship report.

3.2 Overall time plan

3.2.1 Planning

Phasing	Activities	Days	Start	End
Sprint 0	Creating the project plan. Doing research.	10		
Sprint 1	Doing research.	10		
Sprint 2	Implementing user stories 1 and 2.	10		
Sprint 3	Implementing user stories 3 and 4. Documenting research findings.	10		
Sprint 4	Finishing user story 4 and implementing user story 5. Start working on the internship report.	10		
Sprint 5	Finishing user story 5 and implementing user story 6. Working on the internship report.	10		
Sprint 6	Finishing user story 6 and implementing user story 7. Implementing user stories 8 and 9. Working on the internship report.	10		
Sprint 7	Finishing user story 7 and implementing user story 8. Finish first version of the internship report and send this to supervisors for feedback.	10		
Sprint 8	Finish user story 8. If there is time, start implementing user story 9. Working on internship report. Process feedback if given.	10		
Sprint 9	Finishing project and internship report	10		

3.2.2 Explanation

The planning you see above is based on all currently formed user stories being realizable. Because we work following scrum, the company's needs can change, and if they do, the planning will change as well. At the beginning of each sprint we will plan what the intern will actually do that sprint.

This planning can be changed if there is time needed for other activities or in case of delay on currently planned activities. Also, some days in the sprints the student will not be available to work on the project due to for example national holidays or visits to Fontys.

4. Test approach and configuration management

4.1 Test approach

The company values the quality and reliability of their system, that's why the project will be executed following Test-Driven Development. This means that when working on a user story, I will first make tests, and then make the functionality that needs to conform to that test.

The Web API will be tested in 2 ways. These are unit tests for the internal methods and integration tests for testing the API calls. Testing the front-end is just as important, which will be done by testing the logic with unit tests and testing the user interface. How this will be tested will result from the research about front-end frameworks.

4.2 Test environment and necessities

The Web API and its unit tests will be made in Microsoft Visual Studio. The unit tests will use the Entity Framework to mock the data of the database. In addition to the unit tests, there will be integration tests testing the API calls. These integration tests will consist of an Postman script.

The development and test environment for the front-end will be dependent on what front-end technology will be used. They will be specified later in the project.

4.3 Configuration management

4.3.1 Deployment

The current CrashFix system is hosted on a virtual machine and is only accessible through the company network. The company wants to deploy the new system in the same way on the same virtual machine. However, this will not happen every sprint, but only when the project is finished.

4.3.2 Documentation

The documentation for this project will be saved in the version control system (Git). This ensures that when the documentation changes, it's available for everyone that has access to the repository. The documentation will exist of a Project plan, an internship report, API documentation and an research document.

4.3.3 Git flow

This project will use Git as version control, more specifically the TFS git system. Because there is only one person working on this project, there is no need for many repositories. As such there are only 2 branches, the master and the development branch. The develop branch will be used for making new functionalities. When these are finished (See Definition of Done) they will be pushed to the master branch.

Appendix B: Research document

In this second appendix, my research documentation can be found. This research documentation was created synchronically with the project when performing research.

Research document

CRASH REPORT SYSTEM

Esp 300, 5633 AE Eindhoven

Date:	
Version:	1.0
Status:	Waiting for feedback
Author:	Rik van Spreuwel

Version

Version	Date	Author(s)	Changes	Status
0.1		Rik van Spreuwel	Created document	In progress
1.0		Rik van Spreuwel	Finished first draft	Waiting for feedback

Distribution

Version	Date	To

Glossary

Term	Definition
Crash Report	When a TrueSort program crashes, the program sends a report to the CrashFix system. We call this report a Crash Report. A crash report contains information regarding the crash – examples of this are: where in the code has the crash taken place, what time has it taken place etc.
Collection	A Collection is a bundle of crash reports that belong to the same bug/problem.
TFS	TFS stands for Team Foundation Server. This is a Microsoft software development tool.

Table of Contents

1. Introduction

This document was created to include all relevant research which was conducted to answer the research questions for this project. These research questions were initially created in the Project Plan document. During the project, some new research questions/subjects came up and were added later to this document and are therefore not found in the Project Plan document.

2. How is the current CrashFix system organized?

The architecture of the current CrashFix system is visualized in a diagram in Figure 1. The sorting machines (Clients) are located at the customers. When a crash takes place somewhere in the sorting machines' system, the data of that crash is bundled in a zip file and sent to the CrashFix service.

This service processes the incoming zip files, then saves the zip files to disk and stores the processed data in the database. The service can only be accessed by the web application, not by users directly. The web application is a PHP website that allows CrashFix users to perform administrative and daily usage tasks through a web browser application.

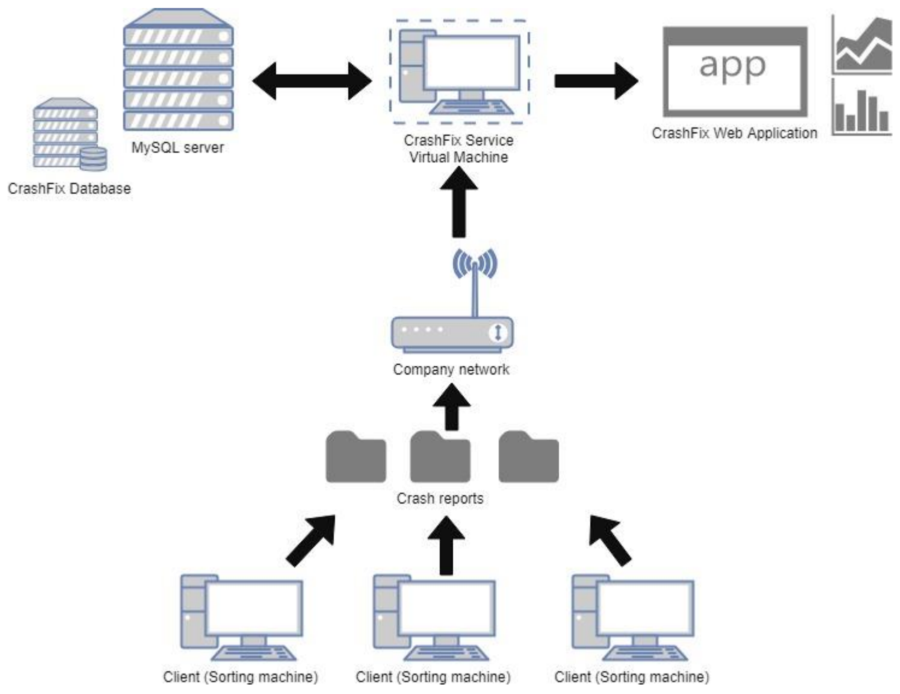


Figure 1: The current CrashFix system visualized in a diagram.

3. Can the new front-end be made as an extension of Exceptionless?

At first, we thought maybe we could make an extension for Exceptionless. To find out if this was possible, I searched the internet for possible solutions, but there wasn't much to find. The only thing I could find close to this was adding plugins to Exceptionless, but this did not include front-end plugins. I couldn't find any examples of this for this online, so it seemed to be a task outside the scope of an internship. Due to this, it made no sense to move forward on this subject.

4. What are the issues with the current CrashFix system?

To determine what the issues/missing functionalities with the current system are, I first asked several colleagues what problems they encountered with the system and what functionalities they were missing. This resulted in a list of issues and functionalities. To get a better view on this point, the company gave me access to the system, in order for me to explore the system and find the issues myself. This did give me a better viewpoint on the issues with the system, resulting in the following list of issues and missing functionalities:

- After logging in to the application, if you are inactive for over 5 minutes, you will be automatically logged out. This is really frustrating to work with.
- The search option is limited to searching for titles/IP addresses and email addresses. The search option is also separated per category (CrashReport/Collection/Bug), the company would like to see a global search option.
- You can't connect a bug of the CrashFix system to the TFS scrum board, making it hard to keep track of bugs in both systems at the same time.
- You can delete bugs/crash reports/collections, but this can break the bugs system, resulting in error pages.
- The application's design is old and doesn't look good.
- The application Makes suboptimal use of the size of the screen.

5. What are the requirements for the new CrashFix system?

To create a list of requirements, I consulted the company supervisor and one of his colleagues. This resulted in three lists of requirements: functional requirements, quality requirements and non-functional requirements. Within this system, there is only one actor, the user. A user in this system is a developer in the company.

The functional requirements are phrased in a list of user stories, these user stories have been prioritised by High, Average, and Low. The user stories/functional requirements are listed below (Note: this list has been updated throughout the project):

1. As a user, I want to view a list of all crash reports, so that I have a better overview of what crashes there are (High).
2. As a user, I want to be able to view the details of a crash report, so that I can analyse it (High).
3. As a user, I want to view a list of collections, so that I have a better overview of what collections there are (High).
4. As a user, I want to be able to open a collection, so I can view all crash reports within the collection (High).
5. As a user, I want to be able to create a bug on collections, so I can track what bugs my project has (High).
6. As a user, I want to view a list of all bugs, so that I have a better overview of what bugs there are (High).
7. As a user, I want to be able to view the details of a bug, containing the connected collections, so I can get a better view of what the bug is (High).
8. As a user, I want to be able to alter an existing bug, so that I can change its values (High).
9. As a user, I want to be able to add an collection to an existing bug, so I can group similar collections in one bug (High).
10. As a user, I want to be able to remove an collection from an existing bug, in case I misplaced the collection to the wrong bug (High).
11. As a user, I want to be able to download crash report files, so that I can analyse these files myself (High).
12. As a user, I want to be able to delete crash reports, so I can keep my system from becoming over-crowded (High).
13. As a user, I want to be able to delete collections, so I can keep my system from becoming over-crowded (High).
14. As a user, I want to be able to delete bugs, so I can keep my system from becoming over-crowded (High).
15. As a user, I want to be able to view all domain users and select what users to include in my application, so that I can assign users to bugs (High).
16. As a user, I want to be able to sort crash reports, so that I can get a better overview of all crash reports (Average).
17. As a user, I want to be able to filter crash reports, so that I can get a more specific list of crash reports (Average.)
18. As a user, I want to be able to sort collections, so that I can get a better overview of all collections (Average).
19. As a user, I want to be able to filter collections, so that I can get a more specific list of collections (Average).
20. As a user, I want to be able to search crash reports/collections by certain properties, so I can easily find what I need (Average).

21. As a user, I want to be able to create a bug in both the TFS-system and crash report where they are connected to each other (Low).

The quality requirements are to be implemented for each user story during the development of that user story. The quality requirements are listed below:

- The API documentation must be kept updated.
- The functionalities must be tested, to keep the project bug free (tests first, functionality second).

The non-functional requirements are to always be taken into account during the project. The non-functional requirements are listed below:

- Expandability, the system must be expandable for the company's changing requirements.
- Maintainability, the system must be easy to maintain.
- Reliability, the system may not crash and should not have any bugs.
- Usability, the system must be easily accessible for the company's developers and it must be easy to use.

6. What technology is the most suitable for building the Web Application?

6.1 What are the most relevant technologies for building a web application?

To answer this question, I’ve searched the internet on the most relevant and popular front-end technologies in 2018. I found some websites comparing frameworks and listing top web development frameworks. All of these websites were talking about different frameworks, which is not that strange because there is simply a lot to choose from.
I found that the most frequently shown and, to me, the most interesting frameworks were Angular and React. I thought these frameworks were interesting because they have a large user base, easy to learn, both use client-side rendering, and both are component-based, which is very reusable.

I chose to also take a quick look at using ASP.Net MVC with Razor as front-end framework as well because the back end is going to be made in ASP.Net Core, which would make it a single project. It would be nice to have just one application to maintain and develop, but there are also negative sides to this. For one, using this, the front-end and Web API would be one big component, instead of two separate components. We want to keep the UI and Web API separated so that if one of the components may change, this doesn’t affect the other component. Besides this, Razor itself uses server-side rendering. When using server-side rendering, every time you want to view a new web page, that specific web page has to be requested from the server, damaging the responsiveness of the website. The other option is client-side rendering. When using client-side rendering, the entire website is requested once, and then new web pages can be loaded without calling the server, resulting in a more dynamic and responsive website. Because the new user interface has to be dynamic and responsive, while handling a large amount of data, the company wants a client-side rendered framework to be used. The usage of Razor in addition to such a framework would only increase the complexity of the application and was therefore discarded.

The next part of this research is about Angular and React. For both frameworks we will examine the following:

- The project structures
- How to test within the framework
- The separation between UI and logic

6.2 Angular framework

The first framework I chose to look into was Angular. I had worked with angular before, but this was quite a while ago and we didn’t unit test any Angular code at all. That’s why I decided to follow Angular’s Tour of Heroes tutorial again. The tutorial is great for beginners, if you follow the steps there isn’t much that can go wrong. During this tutorial, I looked at the project structure and the separation between UI and logic. Unit testing within Angular was not included in the tutorial and was there for looked at afterwards.

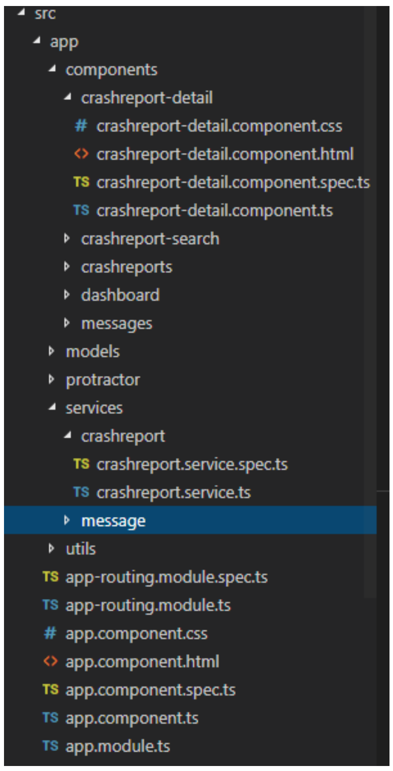
6.2.1 Angular project structure

The most of an Angular project structure consists of components and services. As you can see in Figure 3, on top of the hierarchy, you have the *"src/app"* folders. Within the app folder, I have my most important classes, the base component class (*app.component*), the routing class and the app module. The rest I've divided into components, models, protractor (for visual UI testing), services and utilities.

In the app module, you:
- Declare all the components within your project.
- Import all (libraries) modules you want to use within your project.

Components are (parts of) web pages within an angular project. These components are easily reusable inside other components, which can prevent duplicate code. Within a component, you have an HTML file for the webpage, a CSS file for styling that HTML page, a typescript file for writing the logic of that page. The *spec.ts* file is the unit test file of the component. In this test file you test all the logic in the component and (parts) of the interface.

A service is a broad term. Services can be for passing values, fetching data through HTTP calls, functions or features of an app. A service is a class with a well-defined purpose. The services needed for this application would mostly be for fetching and sending data.



```
src
  app
    components
      crashreport-detail
        crashreport-detail.component.css
        crashreport-detail.component.html
        crashreport-detail.component.spec.ts
        crashreport-detail.component.ts
      crashreport-search
      crashreports
      dashboard
      messages
    models
    protractor
    services
      crashreport
        crashreport.service.spec.ts
        crashreport.service.ts
      message
    utils
    app-routing.module.spec.ts
    app-routing.module.ts
    app.component.css
    app.component.html
    app.component.spec.ts
    app.component.ts
    app.module.ts
```

Figure 3: The image of the project structure in my Angular test project.

6.2.2 Separation between UI and logic

In Angular, the UI and logic are nicely separated. The HTML file takes care of the structure of the component and the CSS file enhances the appearance of the page. The typescript file takes care of the component's logic, this can be anything from fetching data to processing user input. Although the logic is separated from the UI, the typescript file is bound to the HTML file in order to easily bind, present and update data in the component.

Angular works with a two-way data binding. In your typescript code you can declare an object. This object can be linked to in your HTML code, this means that when you change the object in either the typescript code or the HTML code, it changes in both places. This minimizes the risk of errors because the object has the same reference in both places.

6.2.3 Testing in angular

Within Angular you can test both the user interface and the logic. I chose to use the Jasmine testing framework for low-level logic and user input, and look into Protractor to test the higher-level user interface.

With the jasmine tests, you can test API calls by mocking your Services or HTTP clients, telling them to return a specific value when a call is made. You can also test the user input by getting the HTML elements from the front-end using a ComponentFixture.
When you run the tests, the app launches the Karma test runner to execute the tests. The console output shows how many of your tests passed/failed. Karma also opens a Chrome browser that displays the test output on a webpage, making it easier for debugging.
The best thing about testing with the karma test runner in Angular is that whenever you save a file, it runs your unit tests. This is very useful for a Test-Driven approach because you see your tests fail whenever you save code that breaks any of them.

Protractor is useful for testing the high-level user interface. It runs visual in a browser and interacts with the application as a user would. When running the protractor tests, you can see the tests interacting with the application (although this goes rather fast). After the tests are finished you get a console output showing how many tests passed, how many assertions there were and how many failed assertions there were.

6.3 React framework

The second framework to look into was React. The first thing I had to choose was between React and React-native when looking into it, it was quickly clear that React-Native is meant for mobile platforms. That's why I chose to look further into ReactJS. Looking into ReactJS was generally overwhelming, there are a lot of ReactJS toolchains to choose from and there isn't one clearly better than the other.

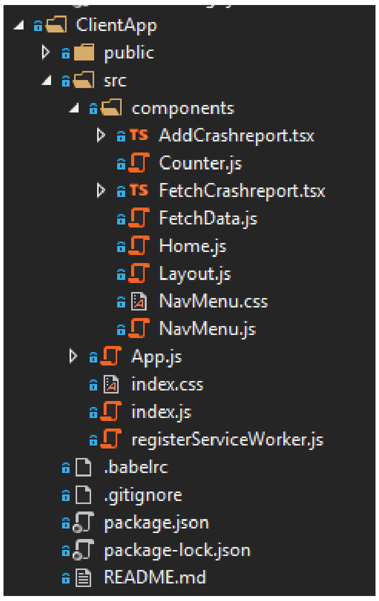
I was completely new to ReactJS and therefore I didn't know what to choose. After a while looking at all the ReactJS toolchains, I stumbled upon the option of using ReactJS in an ASP.Net application in Microsoft Visual Studio. I chose to use this option because when creating a project in Microsoft Visual Studio, you have a standard template which would be a nice example for me to learn from. With this example, I could examine the structure of the ReactJS application and try to make a proof of concept for myself without completely learning ReactJS to do so. More information about these subjects can be found in the next chapters.

6.3.1 ReactJS project structure

In Figure 4 you can see the package structure of my ReactJS test project. As one can see I have a ReactJS project *"ClientApp"*, this project has 2 directories, *"public"* and *"src".* The package and git related files can also be found within the *"ClientApp"* folder (*package.json*, *.gitignore* etc.). In the *"public"* folder you keep public files such as the base *index.html*, which is the base structure of your web pages.

The index.js file that can be found in the *"src"* folder is the base class of your application, it builds your app and renders the components. The App.js declares the available routes in your app, these are often related to your components (*FetchData.js* -> */fetchdata*).

The components can be found in the *"components"* folder within the *"src"* folder. The components are the smaller parts of your application. These components add content to your basic HTML page. You can have more than one component visible at one time, for example this project has the NavMenu visible at any time for navigation and another component is visible next to it.

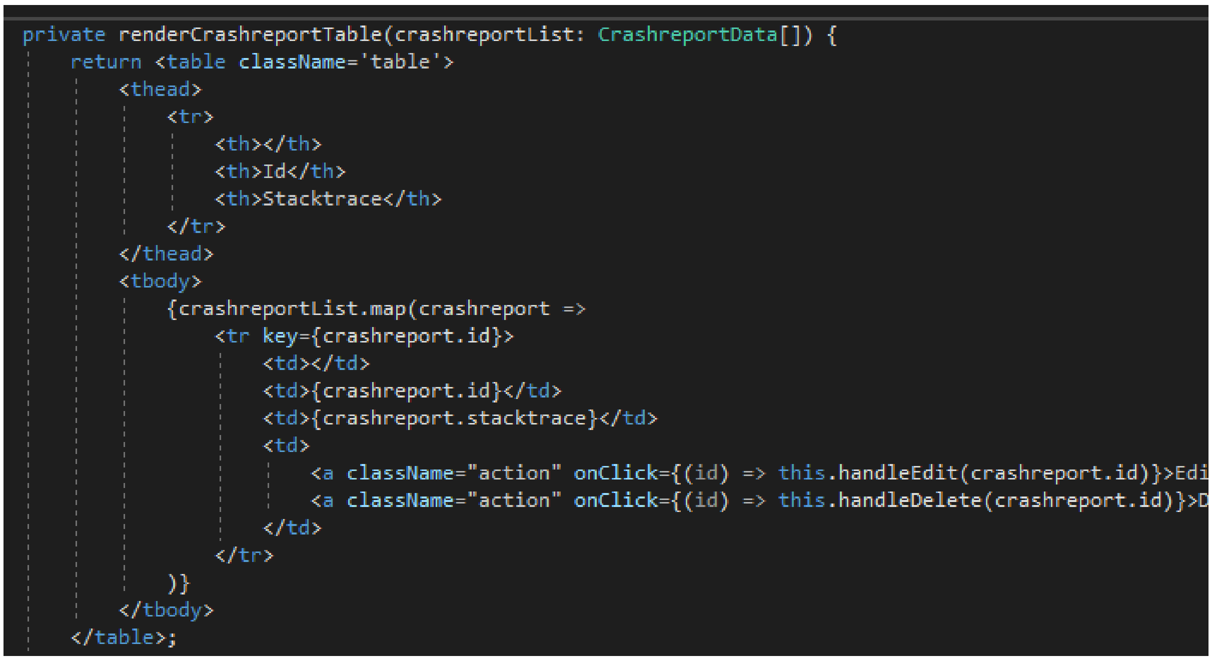


```
ClientApp
  public
  src
    components
      AddCrashreport.tsx
      Counter.js
      FetchCrashreport.tsx
      FetchData.js
      Home.js
      Layout.js
      NavMenu.css
      NavMenu.js
    App.js
    index.css
    index.js
    registerServiceWorker.js
  .babelrc
  .gitignore
  package.json
  package-lock.json
  README.md
```

Figure 4: ReactJS (test) project structure

6.3.2 Separation between UI and Logic

A ReactJS webpage consists of components, these components are JavaScript files. In these JavaScript files, you have both the logic and user interface. As one can see in Figure 5, in a component you have all the JavaScript/typescript code (logic) and in this code, you inject HTML (User Interface) code into your base HTML page. With this, you have both the logic and UI in the same class, giving no separation at all and making the code chaotic.



```
private renderCrashreportTable(crashreportList: CrashreportData[]) {
    return <table className='table'>
        <thead>
            <tr>
                <th></th>
                <th>Id</th>
                <th>Stacktrace</th>
            </tr>
        </thead>
        <tbody>
            {crashreportList.map(crashreport =>
                <tr key={crashreport.id}>
                    <td></td>
                    <td>{crashreport.id}</td>
                    <td>{crashreport.stacktrace}</td>
                    <td>
                        <a className="action" onClick={(id) => this.handleEdit(crashreport.id)}>Edi
                        <a className="action" onClick={(id) => this.handleDelete(crashreport.id)}>D
                    </td>
                </tr>
            )}
        </tbody>
    </table>;
```

Figure 5: Part of the code from the FetchCrashreport class.

6.3.3 Testing in ReactJS

Regarding testing in ReactJS, I expected that it would be very hard and chaotic since the component code is mixed with HTML code. I was working with Visual Studio, so I had to look at what Visual Studio supported. The official Microsoft website states that they support 4 testing frameworks, one being Jasmine (which I used in Angular). I chose to try and get Jasmine tests working with ReactJS, but I couldn't get it working because jasmine could not find any tests.
Earlier parts of this research already showed that ReactJS didn't fit the company's needs, with the UI and logic not being separated and the code being chaotic. That's why, after some attempts to get it working, I chose to stop trying to run unit tests in ReactJS.

6.4 Conclusion

There are a lot of front-end frameworks. The most relevant frameworks for my project were Angular and React, which I looked further into. I did research on both frameworks and tried them out with a Proof of Concept, but there was a clear winner.

I chose, in consultation with the company supervisor, to use Angular for my application. My reasons for choosing Angular were:

- Angular is component based. Because of this you can easily separate different functionalities and reuse them where needed.
- Angular is very flexible and responsive.
- Angular has UI and Logic nicely separated (in React it isn't, making it very chaotic).
- Angular is very well documented. There is much documentation to find which is good for development and maintainability (The ReactJS documentation is very chaotic).
- Very fast compilation, every time you save the project recompiles and runs the tests again. Using this, you (almost) instantly see the visual changes and you notice if your changes break any tests.

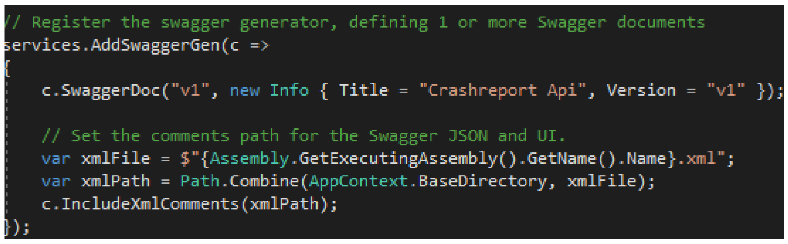
7. What is the best tool for documenting the Web API?

Regarding the Web API documentation tool, the technology we will use is a Swagger dependency for ASP.Net. This dependency will generate a JSON file, which the Swagger interfaces can decipher into a clear documentation page. When using swagger in ASP.Net Core, there are 2 different interfaces you can use: SwaggerUI and Redoc. These interfaces are quite different from each other and both work intuitive. In this research, we will compare both interfaces and determine what is the best choice regarding this project.

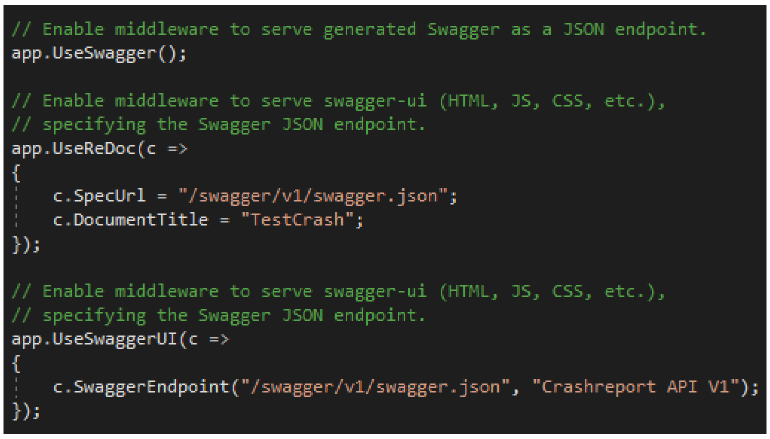
While studying the documentation for both interfaces, we discovered that the implementation of the interfaces is nearly identical. We also discovered that you can use both interfaces at the same time. Because of this, we chose to start by using both interfaces and just pick one at a later term.

The implementation of the interfaces (code-wise) can be seen in Figures 6 and 7. Figure 6 shows the *addSwaggerGenerator* method added to the *ConfigureServices* method. In this method, you apply the path of the XML file created for the comments in the Web API. This is provided to customize the generated interfaces.

In Figure 7 you see the code for the Configure method in the *startup.cs* class. In this method, we add *app.UseSwagger* to serve the generated Swagger as a JSON endpoint. The methods *UseRedoc* and *UseSwaggerUI* enable the Redoc and SwaggerUI interfaces. The SwaggerUI interface is visible at the */swagger* route and the Redoc interface is visible at the /API-docs route.



```
// Register the swagger generator, defining 1 or more Swagger documents
services.AddSwaggerGen(c =>
{
    c.SwaggerDoc("v1", new Info { Title = "Crashreport Api", Version = "v1" });

    // Set the comments path for the Swagger JSON and UI.
    var xmlFile = $"{Assembly.GetExecutingAssembly().GetName().Name}.xml";
    var xmlPath = Path.Combine(AppContext.BaseDirectory, xmlFile);
    c.IncludeXmlComments(xmlPath);
});
```

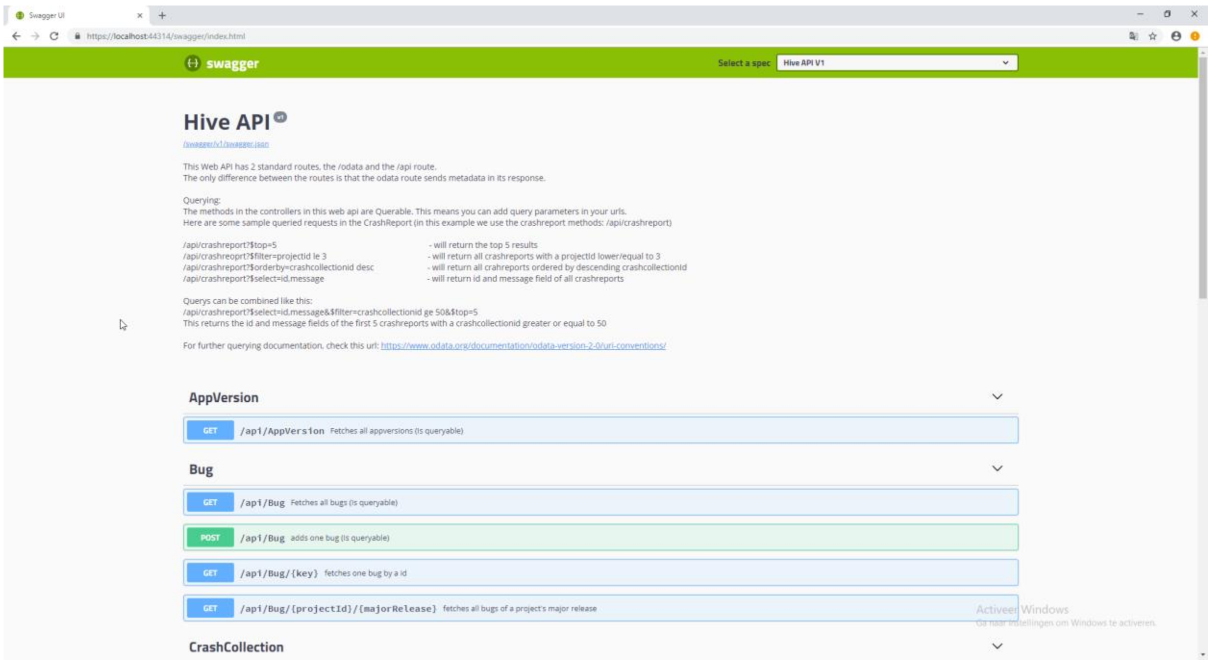


```
// Enable middleware to serve generated Swagger as a JSON endpoint.
app.UseSwagger();

// Enable middleware to serve swagger-ui (HTML, JS, CSS, etc.),
// specifying the Swagger JSON endpoint.
app.UseReDoc(c =>
{
    c.SpecUrl = "/swagger/v1/swagger.json";
    c.DocumentTitle = "TestCrash";
});

// Enable middleware to serve swagger-ui (HTML, JS, CSS, etc.),
// specifying the Swagger JSON endpoint.
app.UseSwaggerUI(c =>
{
    c.SwaggerEndpoint("/swagger/v1/swagger.json", "Crashreport API V1");
});
```

Figures 6 and 7: Implementation of the Swagger interfaces in the ASP.Net *Startup.cs* class.

7.1 SwaggerUI interface

The first Swagger interface is SwaggerUI. SwaggerUI is the standard Swagger interface, this interface shows clear documentation for the Web API. In Figure 8 you can see the top of the SwaggerUI page, this page has a custom description at the top of the page. The second image (Figure 9) shows the bottom of the page, where you can see the data models.
In the third image (Figure 10) the Post method is opened and the "try out" option is selected. This gives you the chance to try the API method out yourself.



Hive API v1

/swagger/v1/swagger.json

This Web API has 2 standard routes, the /odata and the /api route.
The only difference between the routes is that the odata route sends metadata in its response.

Querying:
The methods in the controllers in this web api are Querable. This means you can add query parameters in your urls.
Here are some sample queried requests in the CrashReport (in this example we use the crashreport methods: /api/crashreport)

```
/api/crashreport?$top=5                              - will return the top 5 results
/api/crashreoprt?$filter=projectid le 3              - will return all crashreports with a projectId lower/equal to 3
/api/crashreport?$orderby=crashcollectionid desc     - will return all crahreports ordered by descending crashcollectionId
/api/crashreport?$select=id,message                  - will return id and message field of all crashreports
```

Querys can be combined like this:
/api/crashreport?$select=id,message&$filter=crashcollectionid ge 50&$top=5
This returns the id and message fields of the first 5 crashreports with a crashcollectionid greater or equal to 50

For further querying documentation, check this url: https://www.odata.org/documentation/odata-version-2-0/uri-conventions/

AppVersion

GET /api/AppVersion Fetches all appversions (Is queryable)

Bug

GET /api/Bug Fetches all bugs (Is queryable)

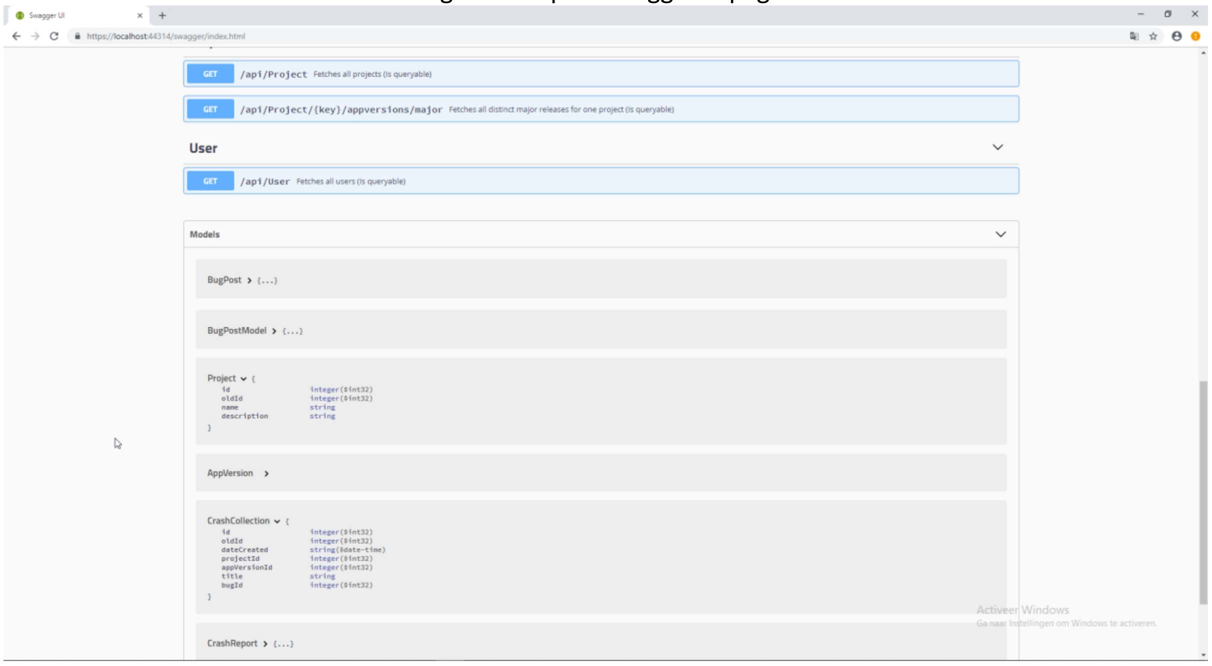
POST /api/Bug adds one bug (Is queryable)

GET /api/Bug/{key} fetches one bug by a id

GET /api/Bug/{projectId}/{majorRelease} fetches all bugs of a project's major release

CrashCollection

Figure 8: top of SwaggerUI page.



GET /api/Project Fetches all projects (Is queryable)

GET /api/Project/{key}/appversions/major Fetches all distinct major releases for one project (Is queryable)

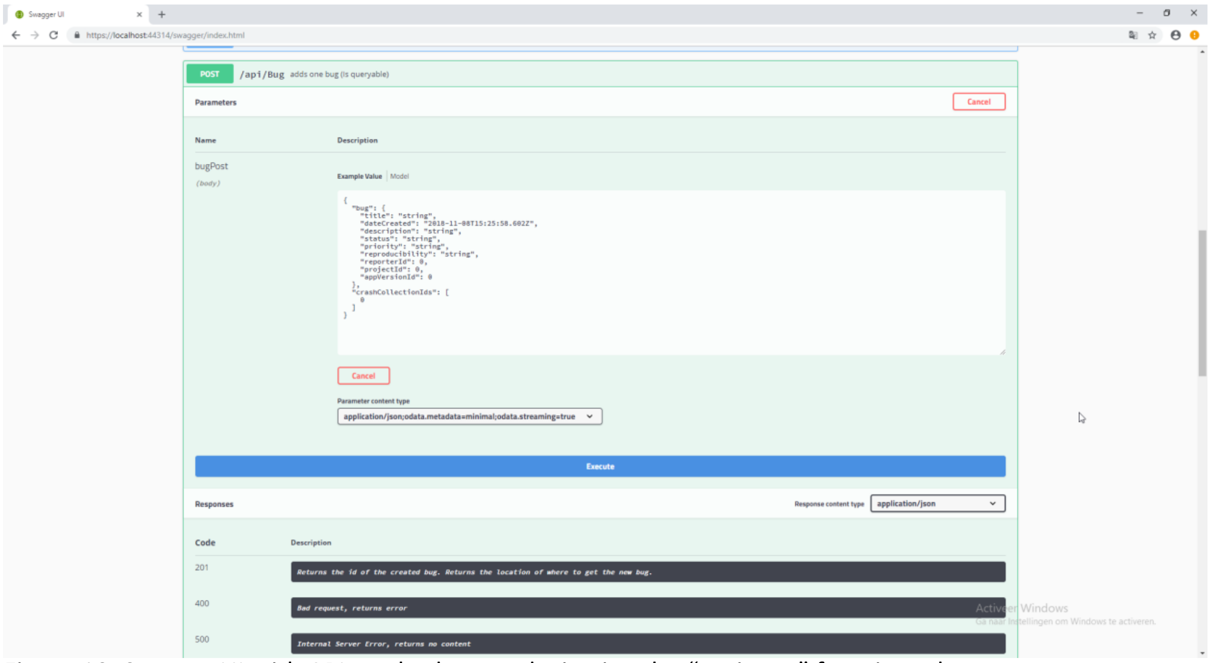
User

GET /api/User Fetches all users (Is queryable)

Models

```
BugPost > {...}

BugPostModel > {...}

Project v {
   id                 integer($int32)
   oldId              integer($int32)
   name               string
   description        string
}

AppVersion >

CrashCollection v {
   id                 integer($int32)
   oldId              integer($int32)
   dateCreated        string($date-time)
   projectId          integer($int32)
   appVersionId       integer($int32)
   title              string
   bugId              integer($int32)
}

CrashReport > {...}
```

Figure 9: Bottom of SwaggerUI page viewing data models.


Swagger UI
https://localhost:44314/swagger/index.html
POST
/api/Bug adds one bug (Is queryable)
Parameters
Cancel
Example Value
Model
Cancel
Execute

Name	Description
bugPost *(body)*	

```
{
  "bug": {
    "title": "string",
    "dateCreated": "2018-11-08T15:25:58.602Z",
    "description": "string",
    "status": "string",
    "priority": "string",
    "reproducibility": "string",
    "reporterId": 0,
    "projectId": 0,
    "appVersionId": 0
  },
  "crashCollectionIds": [
    0
  ]
}
```

Parameter content type: application/json;odata.metadata=minimal;odata.streaming=true

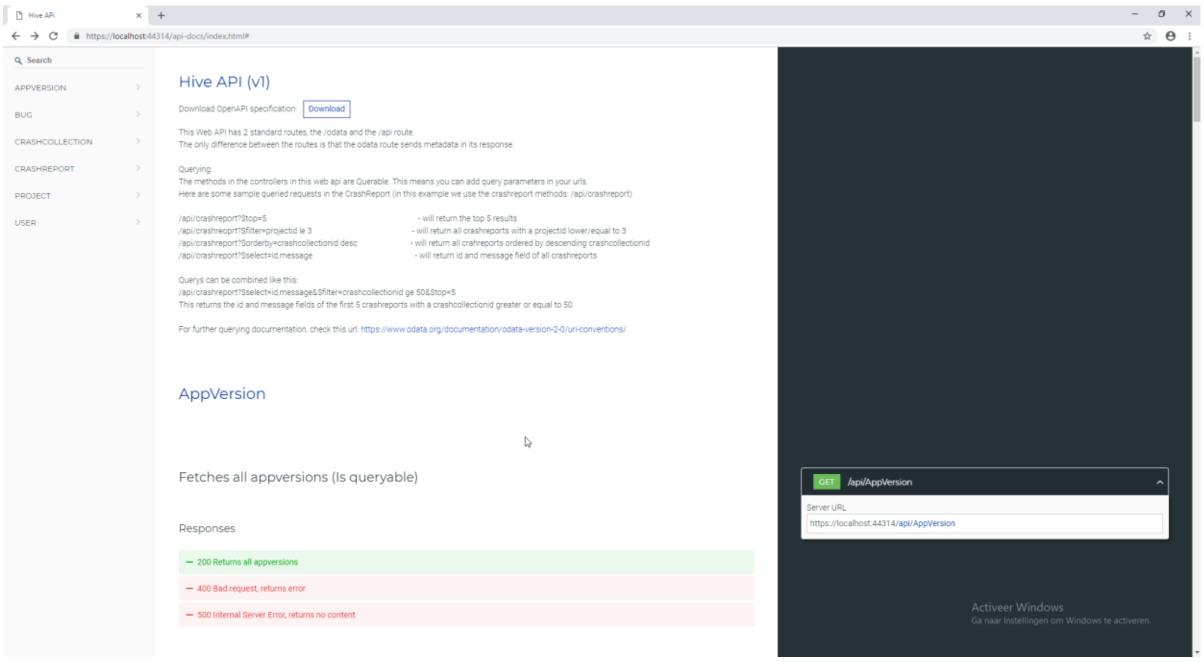
Responses — Response content type: application/json

Code	Description
201	*Returns the id of the created bug. Returns the location of where to get the new bug.*
400	*Bad request, returns error*
500	*Internal Server Error, returns no content*

Figure 10: SwaggerUI with API method opened, viewing the “try it out” function where you can try out the API method.

7.2 Redoc interface

The Redoc interface for Swagger looks very nice but contains less information. The Redoc interface is displayed in Figures 11 and 12. The interface takes your full screen-width and is spread in 3 parts.
To the left of the screen is a navigation bar, where you can select API methods, which scrolls the screen down to that specific method. In the middle you have the most information, the description of the page is shown and most of the API method’s information. To the right, you have the Methods themselves, which show the server URL when expanded.
The problem with this interface is that, because it isn’t fully developed yet, it has no “Try out” function. This option is really nice to have for testing an API method manually, especially for POST methods, as you can easily add a body to your request with the Try Out option.



Hive API (v1)

Download OpenAPI specification: Download

This Web API has 2 standard routes, the /odata and the /api route.
The only difference between the routes is that the odata route sends metadata in its response.

Querying:
The methods in the controllers in this web api are Querable. This means you can add query parameters in your urls.
Here are some sample queried requests in the CrashReport (in this example we use the crashreport methods: /api/crashreport)

```
/api/crashreport?$top=5                                   - will return the top 5 results
/api/crashreoprt?$filter=projectid le 3                   - will return all crashreports with a projectId lower/equal to 3
/api/crashreport?$orderby=crashcollectionid desc          - will return all crahreports ordered by descending crashcollectionid
/api/crashreport?$select=id,message                       - will return id and message field of all crashreports
```

Querys can be combined like this:
/api/crashreport?$select=id,message&$filter=crashcollectionid ge 50&$top=5
This returns the id and message fields of the first 5 crashreports with a crashcollectionid greater or equal to 50

For further querying documentation, check this url: https://www.odata.org/documentation/odata-version-2-0/uri-conventions/

AppVersion

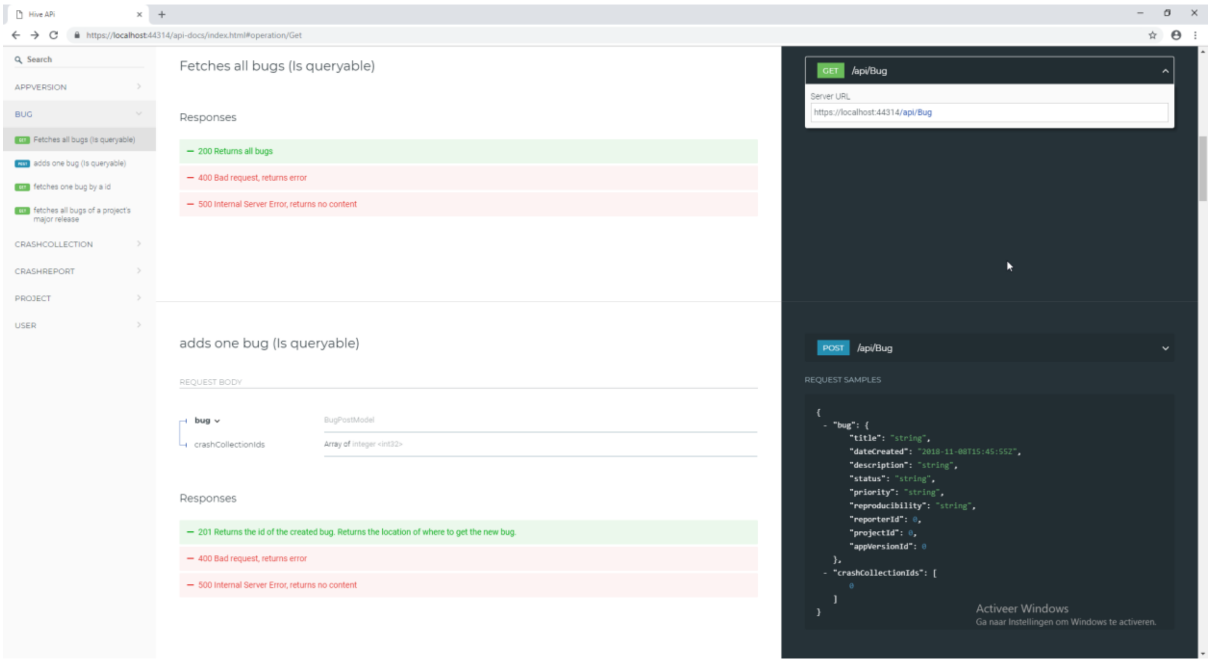
Fetches all appversions (Is queryable)

Responses

— 200 Returns all appversions
— 400 Bad request, returns error
— 500 Internal Server Error, returns no content

GET /api/AppVersion
Server URL
https://localhost:44314/api/AppVersion

Figure 11: Redoc interface top page


Hive APi
https://localhost:44314/api-docs/index.html#operation/Get
Search
APPVERSION
BUG
GET Fetches all bugs (Is queryable)
POST adds one bug (Is queryable)
GET fetches one bug by a id
GET fetches all bugs of a project's major release
CRASHCOLLECTION
CRASHREPORT
PROJECT
USER

Fetches all bugs (Is queryable)

Responses

— 200 Returns all bugs
— 400 Bad request, returns error
— 500 Internal Server Error, returns no content

GET /api/Bug
Server URL
https://localhost:44314/api/Bug

adds one bug (Is queryable)

REQUEST BODY

bug	BugPostModel
crashCollectionIds	Array of integer <int32>

Responses

— 201 Returns the id of the created bug. Returns the location of where to get the new bug.
— 400 Bad request, returns error
— 500 Internal Server Error, returns no content

POST /api/Bug

REQUEST SAMPLES

```
{
  - "bug": {
        "title": "string",
        "dateCreated": "2018-11-08T15:45:55Z",
        "description": "string",
        "status": "string",
        "priority": "string",
        "reproducibility": "string",
        "reporterId": 0,
        "projectId": 0,
        "appVersionId": 0
    },
  - "crashCollectionIds": [
        0
    ]
}
```

Figure 12: Redoc interface – mid page

7.3 Conclusion

The cleaner UI does not outweigh the benefits which come from the additional functionalities in SwaggerUI. Due to this, we chose to use SwaggerUI.

8. How do I apply Test-Driven Development within my project?

8.1 What is Test-Driven Development?

Test-Driven Development is a software development process in which you repeat a short development cycle. This cycle, visualized in Figure 13, starts at writing test cases for requirements. These tests fail because there is no code that makes the test pass. After writing the tests, you write the code to make the tests pass. When the tests pass, you refactor your working code to clean and improve it. After refactoring your code, the cycle repeats at writing tests.
When working Test-Driven, any new implementation affecting the existing functionality will be easily noticed, because the unit tests cases corresponding to that particular feature will start failing. This keeps developers aware of problems.

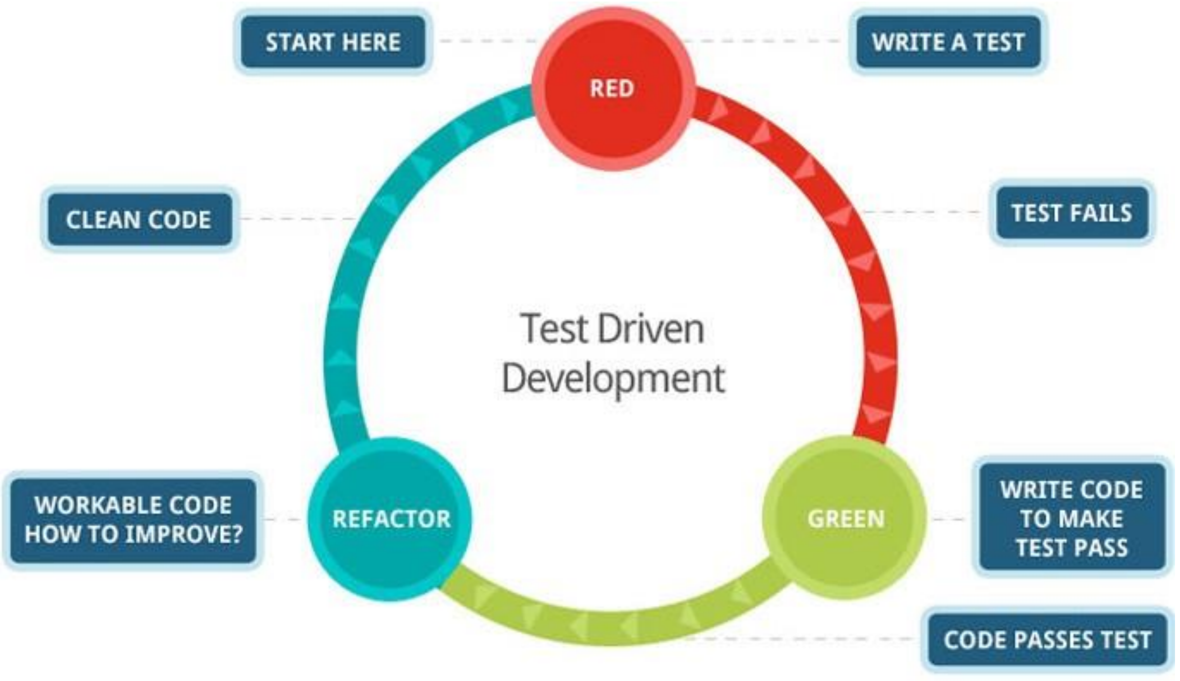


Figure 13: Image clarifying the Test-Driven Development cycle.
Retrieved from https://cdn-images-1.medium.com/max/1600/1*tZSwCigaTaJdovyWlp5uBQ.jpeg

8.2 How can I apply it within my project?

This question has a rather easy answer: I can build my application following the Test-Driven Development cycle mentioned earlier. This way, when new code breaks any of the older functionalities, the corresponding unit tests to that feature will fail, keeping me aware of any problems within my application.

9. What is the current CrashFix database structure?

I used MySQL Workbench to make a connection with the CrashFix database. The first thing I saw was that there were 30 tables, which seemed like a lot to me. When looking at the tables, I had no idea what some of the tables/data fields were used for. There is no documentation for the database, therefore I decided to make a document to explain what certain tables and data fields were for. In this document, I left out a lot of tables, because they weren't being used or were not relevant for the application I was going to make.

After looking further into the tables and what content they had, it became clear to me that a lot of the tables weren't being used. Also, some tables could be merged into one, making the database less complex. I decided to generate an Entity-Relationship Model, which can be seen in Figure 2. When generating this model, I noticed it didn't generate any relationships. After looking into that issue, I realised that the database had no official foreign keys, meaning that the tables weren't connected. To visualize the (non-official) relationships between the tables, I manually added relationships between the tables.

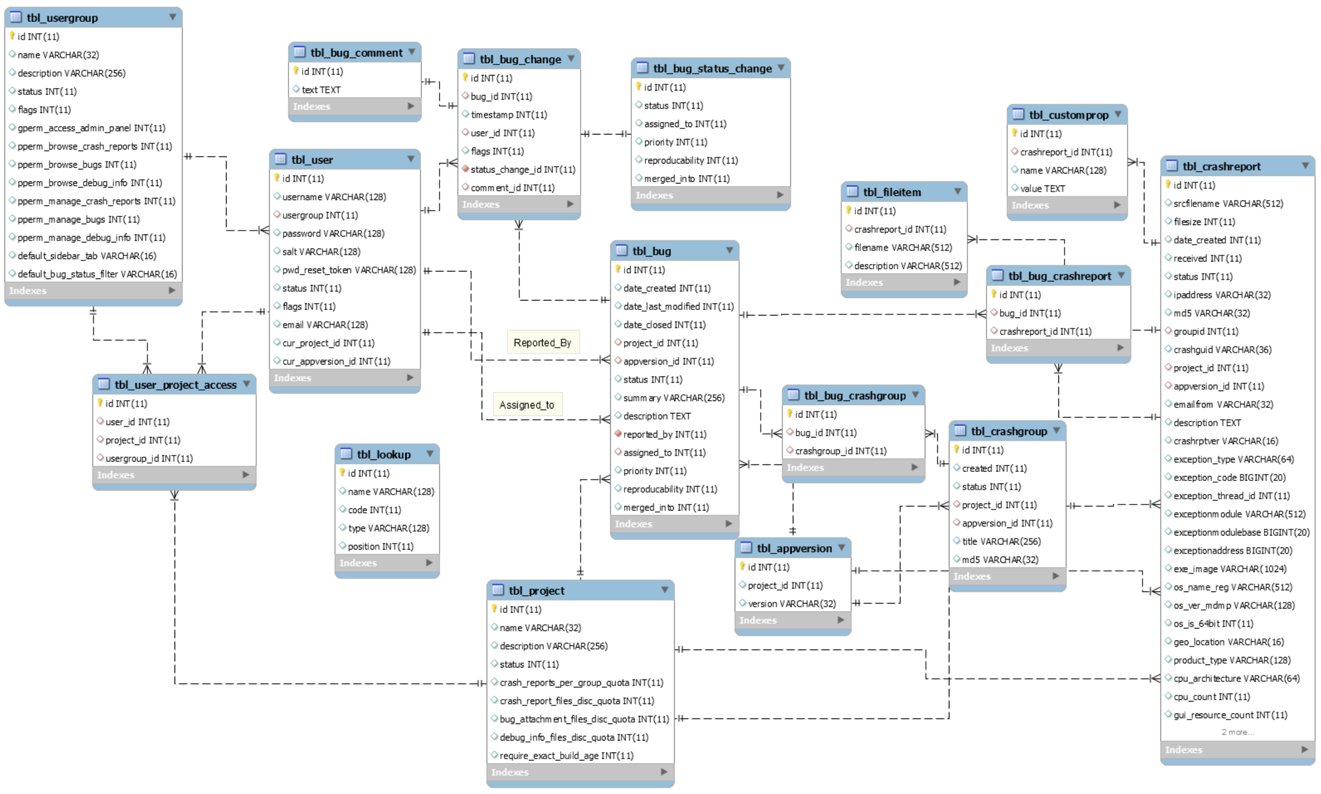


Figure 2: Entity-Relationship Model of the CrashFix database. To keep the model readable, 14 tables were omitted because of a lack of relevance/usage.

10. Entity Framework Core (EF) and OData

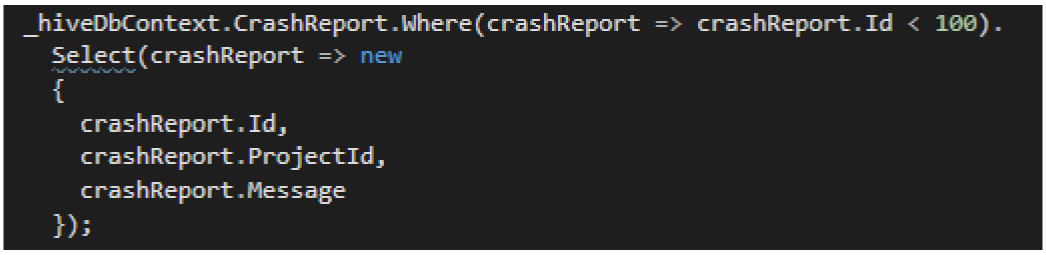
When working with ASP.Net core for building a Web API, the usage of Microsoft's Entity Framework Core (EF) can take care of the most of the data-access code that's usually needed. This can simplify my project and is therefore worth looking into. I will look into the possibility of using the EF for building a new database for my application and how I can use it for unit testing my application where needed.

My company supervisor mentioned the OData protocol and asked me to look into the possibility of using this in my project. I discovered that when using the EF, you can use OData controllers that implement the OData protocol automatically. More about this can be found in 9.2 Using OData with EF.

10.1 Using EF for building a new database

The official Microsoft documentation states that: "With EF Core, data access is performed using a model. A model is made up of entity classes and a derived context that represents a session with the database, allowing you to query and save data.".
Using the Entity Framework, your database tables are created from your entity classes. This gives you the possibility to create a database simply by writing C# entity classes. Another possibility is to generate a model from an existing base. This means you first create a database, and let the EF generate a model with entity classes based on your database.

When using the EF, data is retrieved using Language Integrated Queries (LINQ). This replaces making SQL coding SQL queries yourself and makes retrieving your data a lot easier. An example of a LINQ query is displayed in Figure 14.



```
_hiveDbContext.CrashReport.Where(crashReport => crashReport.Id < 100).
  Select(crashReport => new
  {
    crashReport.Id,
    crashReport.ProjectId,
    crashReport.Message
  });
```

Figure 14: Entity Framework Core query example

10.2 Testing with EF

When you've already created your database model, unit testing with the Entity Framework is easy. You can use SQLite as an in-memory database, which you can create using the same database model as you have in the non-unit test program. This lets you test with an almost identical database and is rather easy to set up.

10.3 Using OData with EF

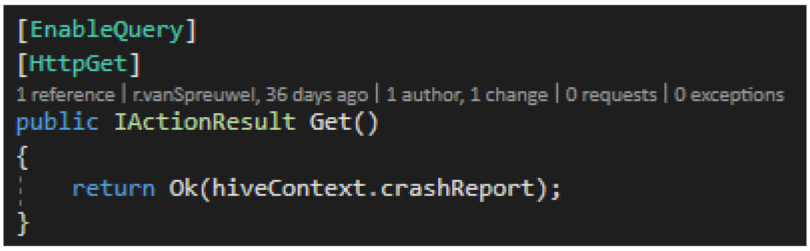
10.3.1 What is the OData protocol?

As the OData website states: "OData (Open Data Protocol) is a standard that defines the best practice for building and consuming RESTful APIs.". The protocol itself is very big and has many rules and models. The things I found most relevant for my project is the URI conventions and query options. With the querying in the OData protocol, you can select specific data simply by adding query strings to the API URL.

I will leave a link towards the protocol in case the reader is interested in the exact details: https://www.odata.org/.

10.3.2 Usage of the OData library in ASP.Net Core

While looking into OData, I noticed there was an OData library for ASP.Net Core, this library can be used to support the OData query syntax in your Web API. When using this library, you can create a Web API Controller deriving from the ODataController. This class supports the reading and writing of data using the OData formats. The biggest benefit of this is that you can enable querying on your API methods. When enabling queries on a simple GET method returning all objects in a table, you can create requests with all sorts of queries. This can be used to filter lists, select specific fields, expand with related entities etc. In Figure 15 a code example is displayed, showing a simple GET method where queries are enabled using the *"[EnableQuery]"* attribute.



```
[EnableQuery]
[HttpGet]
1 reference | r.vanSpreuwel, 36 days ago | 1 author, 1 change | 0 requests | 0 exceptions
public IActionResult Get()
{
    return Ok(hiveContext.crashReport);
}
```

Figure 15: code example of a simple Get method.

Some simple example queries are:
api/crashreport?$filter= projectid le 5
api/crashreport?$select= id, projectid

The first example returns all CrashReports where the project Id is lower or equal to 5.
The second example returns the Id and project Id of all CrashReports.

Despite the benefits of this, the usage of OData controllers has the following downsides as well:

- OData controllers use different routing than MVC. This routing isn't compatible with the Swagger documentation tool. You have to use a workaround in the ASP.Net Core *startup.cs* class to have both working.
- Although it's possible to make both tools work, the Swagger tool doesn't document all the possible query options.
- Testing the OData controllers is difficult, because of a large number of queries that are possible.

10.4 Usage decisions

We will use the Entity Framework to build a new database, which we will fill with the data from the old database. The idea is to make a class that will periodically fetch new data from the CrashFix database and insert it into ours. For doing this, we can let the EF Core generate a database model on the existing database, which makes it easier.

Although the usage of the OData library has its downsides, we think the possibility of being able to query our data easily compensates for that. Therefore, we will be using the OData library for building our Web API.

The testing of both the Web API will be partly done using an in-memory SQLite database where needed.