Department of Informatics

Technical University of Denmark
Lautrupvang 15, 2750 Ballerup

CloseFit

Bachelor project 2016/17

Abstract

This report is part of a full bachelor project completed as part of the Bachelor of Engineering (B.Eng) IT studies at the Technical University of Denmark (DTU) - Center for Bachelor of Engineering Studies (Diplom). Along with this report the produced software (a webapp and a mobile app) constitute the full product of the project.

The purpose of the project has been to develop a prototype web application exposing a service that should help make shopping for clothes online easier and more enjoyable by helping the user to choose clothes of better fitting sizes. A project formulation posed by Mathias Harboe from IT Minds ApS. Beside the web application, a mobile app has been developed to exemplify the use of the service.

As written in the conclusion the purpose of the project has been fulfilled by comparing a user's fitting sizes in other brands to that of other uses and present the distribution of sizes that these users found fitting for the desired clothing brand. The proposed solution has shown, however, that it is not a viable implementation for a production system in its current state and needs further development and care re-evaluation.

Consultant(s):

Henrik Bechmann – Technical University of Denmark

Mathias Harboe – IT Minds ApS

Student Name:	Student Number:	Signature:
Andreas Stensig Jensen	S134408	

Contents

1. Introduction

1.1 Project start

As a bachelor project, this project's desired product is a (software) prototype that can satisfy the project's problem statement and lay the groundwork for further development in order to reach an eventual product that is suitable for business use (production). It does not target a complete and fully polished business-ready product.

The purpose of this report will be to document the projects progress, the product's development, and the solution.

The project, raw product, and this report is targeted towards IT students, IT engineers, and software developers, while the end-user of the raw product are people shopping for clothes online.

1.2 Problem formulation

The formal problem formulation is based on the product of the problem analysis conducted in section 2, and is given here in italics.

Develop a online service that can compare different brands of clothing in different sizes, between submitted request data and stored data, to produce the relevant requested comparison data.

The service shall be available to both customers and web shops, easily usable through a Content Management System (CMS) plug-in, and demonstration of the service must be done through a simple mobile app.

The service must store its own data and allow graphical admin access through a website.

The service must be able to accept new data.

1.3 Milestone plan

As part of the initial start-up phase of the program a milestone plan has been created in order to lay out a clear plan for the duration of the project. This plan includes the different tasks that must be performed (the deliverables) for each key phase of the project and their milestones. Figure 1.1 shows the produced plan with the deadlines.

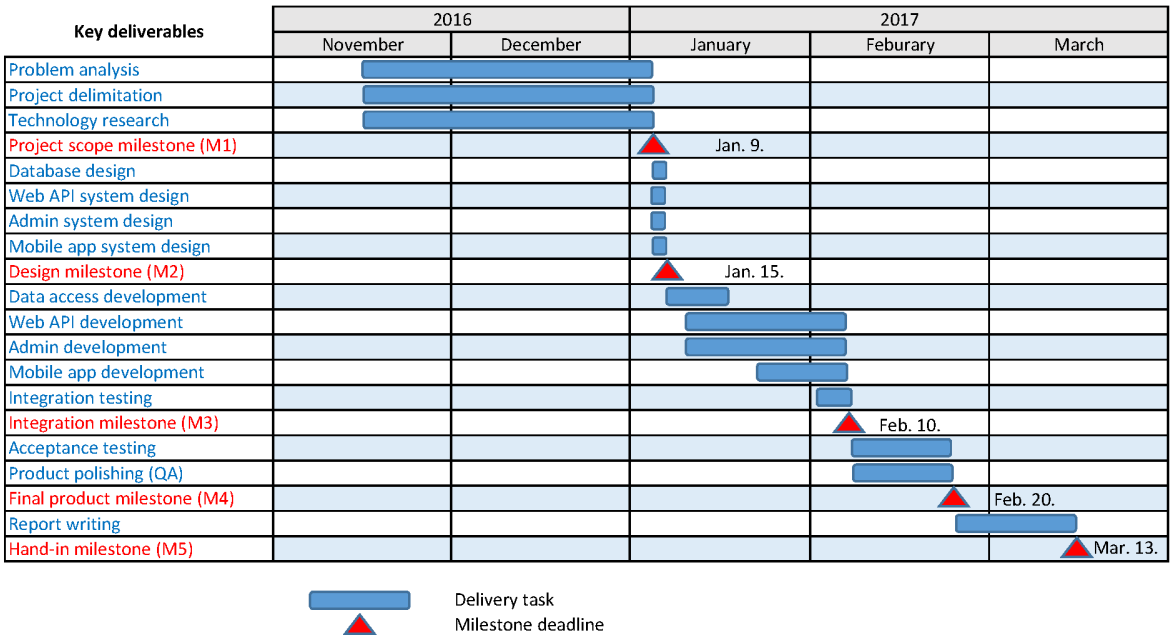


Figure 1.1: Milestone plan for the project.

1.4 Clothing Comparison Concept

Part of the foundation of this project and the key purpose is the concept of clothing comparison. As such it is useful to explain here what this concept entails as its use continues throughout the report.

By a *clothing comparison* or *comparison request* is meant an end user making use of the system by doing two things:

- Specify a clothing type/category and a brand for which the user wishes to get a response.
- Specify a set of reference data that is brands in the same clothing category and the sizes that the user deem to be fitting matches for these brands.

The project solution is to use such a comparison request to generate a response that is meant to help the user to make a better choice of clothes, i.e. based on the reference data that is part of the request.

In extension of the comparison request concept, a clothing *relation*, as used throughout the project, is a data set from a user containing unique brands (in the same clothing category) for which each is given a size that the user find to be fitting. This definition matches the reference data from the *comparison*

request but is primarily used for the system's stored data.

This concept is derived in part from the project description and from dialogue with the the stakeholder. It is also the understanding of both the author and the stakeholder that this solution will not make use of data analysis or artificial intelligence to conduct these comparisons, but simply present pure statistical data.

2. Problem Analysis

In order to create a problem formulation that defines the abstract problem domain, the project description, found in appendix B, has been put through a problem analysis phase. Through this analysis, which is described in the sections below, the primary product has been the System Requirements Specification (SRS), which in turn has formed the basis for the formal problem formulation of section 1.2.

The analysis phase have also produced simple use cases in order to capture the requirements' realisation in the system flow.

2.1 System Requirements Specification

The SRS makes up the requirements for the system, in the case of this project all of the software products.

The requirements are primarily derived from the project description in appendix B. Due to the brief nature and general description given for the project, with its focus being on proof-of-concept prototyping, the freedom to specify more concrete requirements has been given to the author by the stakeholder. These added requirements have been conceived and approved through further dialogue with the stakeholder in the project's conception phase and initial start.

The requirements are separated into groups of functional and none-functional in order to distinguish between the inherent restriction they each create for the system: whether it is on the functionality or the behaviour. This distinction is further emphasised through the ID (bold) given to each requirement, **FRxx** for functional requirements and **NFRxx** for none-functional.

2.1.1 Functional Requirements

The functional requirements listed below specify requirements to the system that are linked to its functionality - what it must be able to do. These are the requirements that are identified as primary concerns and will be the basis of the acceptance testing that evaluates the success of the product's ability to satisfy the problem domain.

FR01 The system shall be able to accept a clothing comparison request, comparing the request clothing data against the systems' clothing data, and resulting in all the matching requested clothing data, across different brands and sizes.

FR02 The system shall store the clothing data.

FR03 The system shall enable admin access to stored data.

FR04 The system shall accept new data to be stored.

2.1.2 None-functional Requirements

The none-functional requirements listed below specify requirements to the system that are not strictly linked to functionality but behaviour - how it must work. These will not be critical to the acceptance testing, but never the less help dictate how the system must work.

NFR01 The system shall operate as an online service.

NFR02 The system shall be be integrate-able with web shop CMS plug-ins.

NFR03 The system shall be accessible both to web shops and customers.

NFR04 The system shall demonstrate its customer access through a simple mobile app.

NFR05 The system shall enable admin access through a graphical website.

NFR06 The system shall store all of its data privately.

2.2 Use Cases

Following the analysis of the requirements, use cases describing the key execution flows of the system have been produced. These can all be found in appendix D.

The use cases, as part of the problem analysis, have been created in order to describe the general flow of the system when it is used and how it realise the functional requirements. Some use cases focus on the system from the end user's perspective, and these are the service interface use cases intended for the user using a sub-system to access the webservice. Others focus on the core webservice system itself as seen from the perspective of a sub-system using the webservice directly, and these are the service system use cases.

The use cases are only used as an early analysis and modelling tool for the system architecture and the solution domain. It will not be the purpose of the use cases to model the details of the execution flows and system interactions. Such is only deemed necessary for systems in which the use cases describe the full limitations of the solution domain, and not simply form a basis for the analysis of the system architecture.

Furthermore, the use cases serve to give the stakeholder a concise overview of the system's workings and how it is used.

2.3 Requirements Traceability Matrix

In order to easily get an overview of which use case map to which requirements a Requirements traceability matrix (RTM) has been created and is found in figure 2.1. A marked cell denotes a mapping between a requirement and a use case, showing that the use case realises the given requirement. Notice that all use cases and requirements are denoted by their IDs. The requirements are found in section 2.1.1 and the use cases in appendix D.

Only the functional requirements have been mapped in the matrix due to their primary concern and link to the acceptance testing of the product. Furthermore, the use cases have not–and will not–be expanded to such a detailed level that all of the none-functional requirements could be captured in their description, and as such they are not mapped here.

Use Case ID	**Functional Requirements**			
	FR01	FR02	FR03	FR04
UCI1	x			
UCI2		x		x
UCS1				
UCS2	x			
UCS3		x		x
UCS4			x	

Table 2.1: Requirements Traceability Matrix mapping functional requirements to use cases.

3. Project Delimitation

3.1 Project Scope

With the project focusing on a proof of concept prototype product there are some delimitations for the project scope that limits that which this project covers–subjects, tasks, and questions which are deliberately ignored and deemed out of scope. These specific delimitations to the scope are mentioned here below, and while they would be fully relevant in a production solution, time and resource constraints for the project warrants their removal from scope following agreement with the stakeholder.

3.1.1 Data Domain

Throughout the project the system is only required to focus on one arbitrary type of clothing for its data domain; specifically t-shirts. The problem solution will comment on implementations and further development extending the solution to including any number of different clothing articles/categories. But as a proof of concept the solution will only be required to develop with shirts being the stored clothing category data. Other clothing types may of course be included in the solution if time permits.

3.1.2 User Domain

The use cases identify a number of users of the system, all of whom can interact with the system's API (the webservice). Although it is the vision of the stakeholder to make the service usable through CMS plug-ins in a production environment, a prototype of such plug-ins are not part of the scope of this project due to added resource cost of learning new programming languages[1]. Rather than writing such plug-ins, the simple mobile app will be used to sufficiently demonstrate how a user can use a middleware sub-system (service interface in the use cases) to interact more seamlessly with the service rather than with the pure API.

3.1.3 Data Integrity

For the purpose of the prototype, an initial focus will not be put on the integrity of data submitted to the service. In a production environment it is feasible that the system would wish to control who can submit data to the system through

[1] Such plug-ins are often scripted in PHP.

the use of various security measures, API keys being one. Such a focus is not maintained in the prototype, which will allow anyone to submit data to be stored in the system per default. The problem solution will instead simply discuss further development to include such features in the system and reflect on the matter.

3.2 Development Strategy

The development strategy for the problem solution is one somewhat related to the Unified Process and the iterative development cycles shown in figure 3.1.

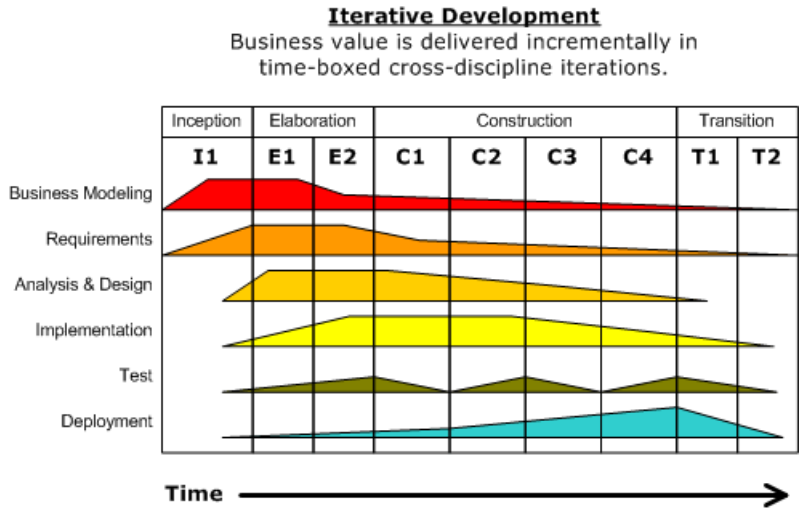


Figure 3.1: Typical diagram of the Unified process development cycle, showing the distribution of work in the different phases. Source: https://commons.wikimedia.org/wiki/File:Development-iterative.png

While not strictly iterative in the sense of progressively building unto the system and delivering it to the stakeholder, the project will make use of pseudo iterations once the requirements and analysis has concluded, focusing on developing the solution in iterations dictated by its natural segmentation: A webservice, a website, and a mobile app to use the webservice.

Since the product is only a proof of concept, the project does not strive to transition beyond the first few construction iterations in terms of the iterative development model.

In a project with a more concrete project description and problem domain, a test-driven development strategy may have proven more fitting in order to pro-

gressively expand and develop the solution based on clear-defined requirements and their test cases.

A basic worklog will also be kept for project and can be found in appendix C.

4. Problem Solution

Following the analysis phase on the problem domain, and delimitation of the project scope, development of the solution is the next phase of the project.

This section starts with an overview of the final solution to offer a top-down understanding of the product, followed by a more detailed explanation of each of the solution's central modules alongside reflections made during their development.

4.1 System Overview

The project's solution to the problem domain consists foremost of two separate software systems: An ASP.NET Core web application[1] serving as the service system and an Android app to exemplify how a service interface should interact with the web application.

While the Android app has a straight-forward system architecture, that of the web application is much more complex, as shown in the system overview figure 4.1. Section 4.2 goes into more details of the structuring of this part of the solution, but as emphasised in the figure the web application is segmented into three areas: Application, Infrastructure, and Core; each containing at least on Dynamic-link library (DLL) assembly.

This structure promote modularity of the system and helps emphasising the high cohesion low coupling tenant of Object-oriented programming (OOP). It also helps to cleanly encapsulate the responsibilities of the system into separate units:

- CloseFit.Application.WebAPI is the executable web application context, containing both ASP.NET WebAPI and ASP.NET MVC to support web-service and website in one unit.
- CloseFit.Infrastructure.Commands, -.Queries, and -.Utilities encapsulate most of the business logic and manipulation of the domain data.
- CloseFit.Infrastructure.DataAccess encapsulates the database context and use the Entity Framework Object-relational mapping (ORM)[2] for communication with an SQL database.
- CloseFit.Core.Entities encapsulates the entity domain data models that are stored in the database and managed by the DataAccess unit.

[1] https://www.asp.net/core
[2] https://msdn.microsoft.com/en-us/library/aa937723(v=vs.113).aspx

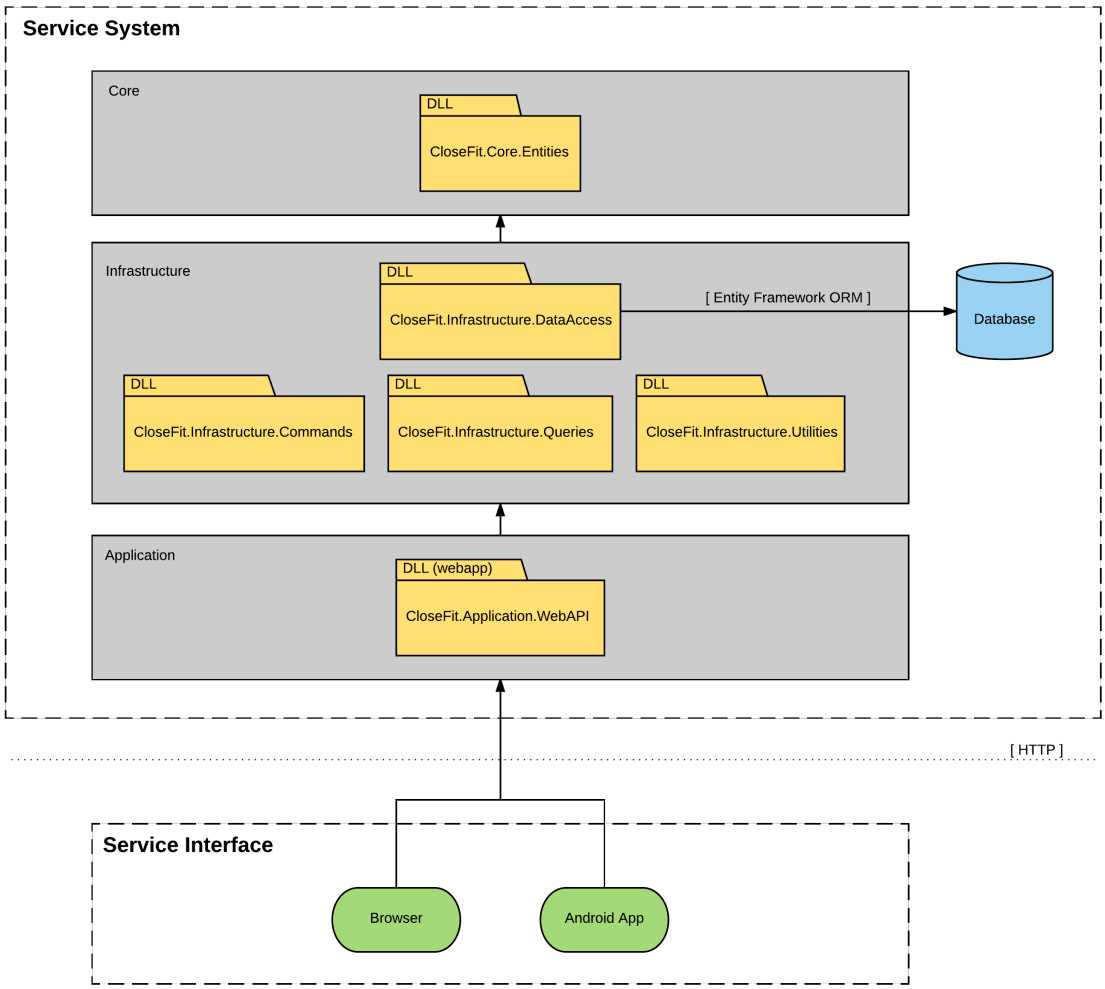


Figure 4.1: Overview of the central modules of the system. Folders denote .NET DLL assemblies.

While figure 4.1 does not emphasise direct communication between the units, the overall hierarchical structure is shown, and communication–as well as dependency–between the units and the segments go only horizontal or upwards, never downwards.

For accessing the service system and downloading the Android app as well as the solution source code, see appendix A.

4.2 Service System

With the service system covering two distinct webapp areas in both exposing a webservice and offering a graphical website, this part of the system makes up the core of the problem solution and the central part of the project's product. It also poses the natural question of whether the solution should be two separate programs or if it should be gathered under the same codebase as one unit of execution.

The latter has been chosen due to the ASP.NET Core framework being able to host a webservice and traditional Model-View-Controller (MVC) website in the same codebase.

4.2.1 Framework - ASP.NET Core

In today's day and age there are many frameworks that are able to easily host a webapp, be it a website or a webservice. Because the project has had no requirements to the frameworks used in regards to the service system, two frameworks has been the primary consideration for the project:

- **ASP.NET Core**[3] is primarily a C# framework but is being expanded to all of the .NET languages (F# and VB)[4]. This frameowkr is entirely focused on web applications.
- **Spring**[5] is a Java framework. The framework covers both desktop, mobile, and web applications.

Both are open-source frameworks and while .NET application traditionally are more easily hosted on Windows systems both are cross-platform enabled. Both frameworks also make use of server-side rendering of views (HTML pages), which makes website design much more powerful.

These two frameworks were the prime contenders due to familiarity with the author, and while both would serve the purpose of the project's requirements

[3] https://www.asp.net/core
[4] https://docs.microsoft.com/en-gb/dotnet/articles/core/index
[5] http://projects.spring.io/spring-framework/

ASP.NET Core has been chosen because of a preference for C# over Java as well as the author having more experience with this given framework.

4.2.2 System Architecture

The architecture for the service system, as presented in figure 4.1, has from the beginning been heavily inspired by the so-called Onion architecture presented in [Pal17]. It is an architecture familiar with the author and is well-suited for web applications as it takes into consideration the frequency by which frameworks and ORMs are updated and changed for these types of software systems.

By seeing the application as an onion with inward layers rather than a building/ladder with floors/steps, the emphasis of this software architecture is that no dependency can work outwards. Code may only depend on the services exposed in its own layer or the layers closer towards the core. This helps to make the system more maintainable when it comes to modularity and changing modules without disrupting dependencies, especially towards the core of the model, which encapsulates the domain.

This model also helps to cleanly separate the domain model from the rest of the system, not making it depend on containing software that makes use of it.

Combining the service system unit from the overview in figure 4.1 with the onion model produces the figure 4.2. Where the general dependency and communication in figure 4.1 only travels upwards, the corresponding dependency and communication only travels inwards towards the core in this model (or within the same layer). Note that the DLL modules are not shown in this figure but remain in their corresponding layers as in 4.1.

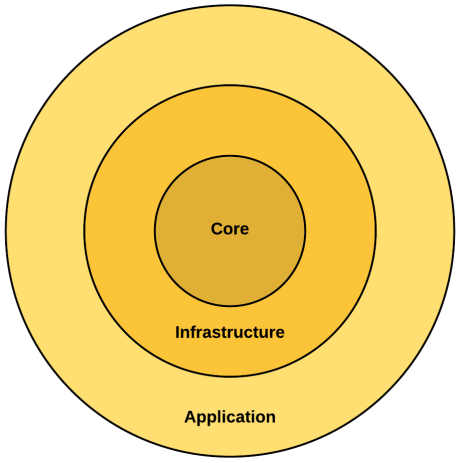


Figure 4.2: The service system code-project structure visualised in the Onion architecture.

The one problem with this structure is the Application layer's dependency on the data access framework (the ORM) which is located in the Infrastructure layer. This cannot be wholly avoided in the ASP.NET Core framework. Because the framework uses dependency injection to avoid strong coupling between the code modules[6] the web application's startup configuration must define the services that are able to be injected, and this includes the ORM, which in this case is Entity Framework. More on this will be given in section 4.2.4.

A (needlessly complex) alternative to this dependency could be to expose the data access as a separate service on the application layer–a separate execution module behind a RESTful HTTP interface. Such a solution would be more viable if the project had a requirement to host the data on a separate server from that of the rest of the service system.

4.2.3 Core - Data Model

The core of the webapp is the data model, the data entities as shown in figure 4.1. These models are mapped to the data contained in the database and are used to represent and manipulate the data throughout the rest of the service system.

These models and the structure of the database have been the target of a lot of thought and experiment during the development of the project. The project description and the system requirements do not expose any obvious data structure, but at the same time it is clear that the data domain has the potential

[6] https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection

to be very volatile–new clothing categories, brands, and sizes could be added regularly in a production environment.

Prototype Design

Two primary designs have had attention during the initial development phase in which the data model was created. Both of these were conceived and evaluated against how they might execute a comparison request, as formulated in section 1.4.

The first solution focused on easy SQL execution, minimising joins and the number of sub-queries required to perform a comparison request. The resulting database design is shown in figure 4.3. In this design, two "meta tables" exists in `ShirtSizeType` and `ShirtBrandType` that contain entries for the possible sizes and brands available for a clothing category–shirts in this example.

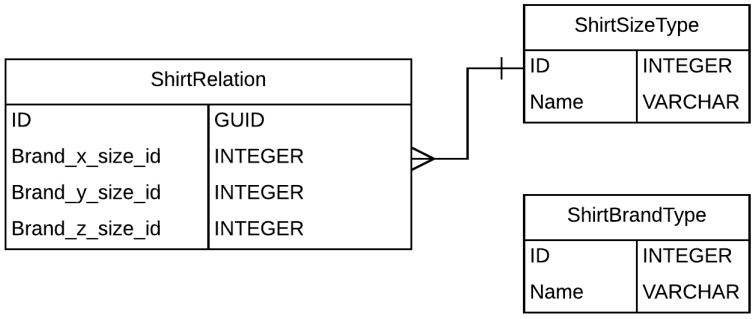


Figure 4.3: The first data model designed for the system. Expressed as a database model diagram. A Shirt is used as an example clothing type.

The `ShirtRelation` table would hold an entry for each shirt relation in the database, wherein each column (beside the ID) is an available brand, corresponding to an entry in the brand meta table (note that there is not direct mapping there). Each value in these columns would then be a foreign-key to a size mapping in the size table, or null if no size was given for this brand in the relation.

This design had the advantage of being able to do a comparison very easily, since the size mappings are already directly in the relation table. Yet it suffers the considerable flaw that any additional shirt-brand that might be added to the domain model would require the relation table to be updated with an additional column, thus expanding all of the existing data with a new null-value.

The second solution, as visualised in figure 4.4, was conceived in direct response to the disadvantage of the first solution, focusing rather on not having

to update the table columns when new brands are added, than on being able to execute a comparison on a direct table.

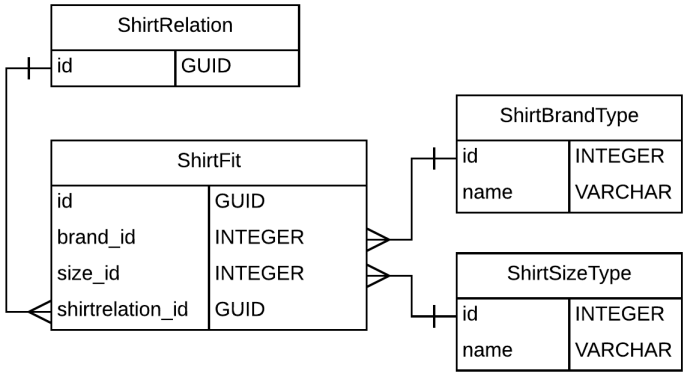


Figure 4.4: The second data model designed for the system. Expressed as a database model diagram. A Shirt is used as an example clothing type.

This solution, like the first, has two meta tables that would contain entries for possible sizes and brands in the given clothing category–shirts in this example.

Instead of having a column for each available brand in the relation table, the relation table only serves to hold an ID. This ID would be used in a new table that holds all entries regardless of which relation they belong to for the given clothing category. Just a foreign-key to the relation marks which entries are related.

Instead of the distinct brand columns, this new table simply has one column to hold a foreign-key to the brand meta table. Thus a fit-entry holds a reference to both the size and brand it is valid for, as well as which relation it is part of.

While this solution removes the first problem of requiring the table column count to be updated with the addition of new brands, it adds a different disadvantage. In order to carry out a comparison the database would have to make an intermediate query to gather all the `ShirtFit` entries in their respective relations, and only when all of these intermediary relation entries have been created can they be filtered based on the reference data in the comparison request, and the desired result brand.

This disadvantage of having to pre-process all of the table before even executing the comparison request was very undesirable, and as such the first data model was selected for the prototype.

It should also be noted that a disadvantage for both solutions is the need for all of the tables to be duplicated for a new clothing category. As shown in the

package diagrams in appendix F.2.1 there exist the exact same entity classes between the TShirt and Sneakers packages.

With the data model in place, the entity classes for the service system have been created, as shown in the core package diagram in appendix F.2.1. Though only the Tshirt models were created initially, Sneakers models were soon added to the domain as well to demonstrate the system's ability to manage different clothing categories.

Because of Entity Framework's features, the entity models have not had to store the foreign-keys to the size tables directly. Instead they can reference the Size meta entity objects (e.g. `TshirtSize`) and the ORM will link them correctly in the database.

4.2.4 Infrastructure - Data Access & Business Logic

The infrastructure layer of the webapp contains the majority of business logic and also contain the `DataAccess` package, which holds the database context–the ORM integration. The package diagrams are found in appendix F.2.2.

Data Access

The data access module (`CloseFit.Infrastructure.DataAccess`) is in of itself very simple. While ASP.NET Core does not come with a build-in ORM and can integrate into a variety of such frameworks, their recommendation is Entity Framework[7].

A powerful feature of this framework is its *Code First* concept[8]. This concept either creates a new database, or updates an existing database, based on the entity classes that are used to define the data model. These entities are stored in the database context (the `ApplicationDbContext` class) and enables Entity Framework to compare the entity classes to their related tables in the connected database.

These changes are tracked via migrations–a common concept among ORMs that implement this *Code First* feature. The migrations are themselves classes in the webapp, wrappers of SQL functions acting on the database, and form a history of changes done to the database based on how an entity model changes, or of they are added or removed from the database context. In the case of the latter two, the migration will either create a new table for the new entity model, or drop the table that the removed entity model was mapped to.

In the cases where migrations modify existing tables, the content of the tables are dropped since it must update the design of the tables. This process is

[7] https://msdn.microsoft.com/en-us/library/aa937723(v=vs.113).aspx
[8] https://msdn.microsoft.com/en-us/library/jj193542(v=vs.113).aspx

visualised (in a simplified way) in figure 4.5

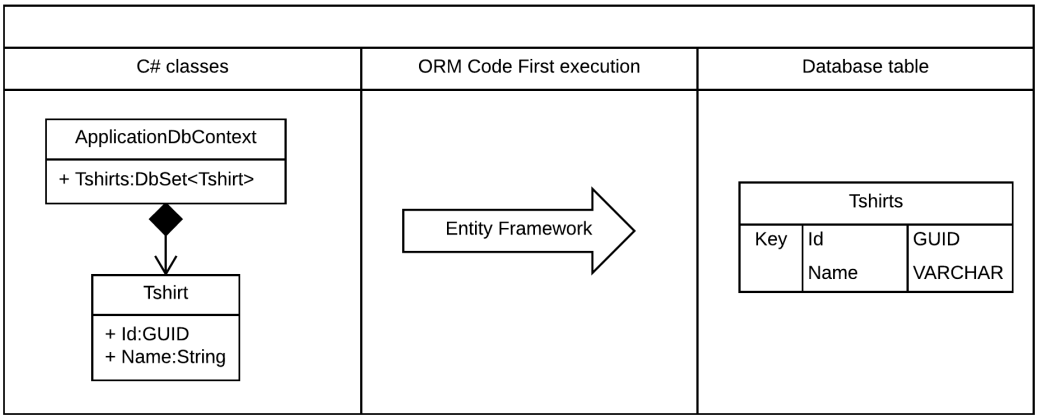


Figure 4.5: Simplified exmaple of how Entity Framework's Code First strategy converts entity classes to database tables.

Using this powerful framework, the webapp's use of an SQL database is managed through the `ApplicationDbContext` class, which is then offered as a service to the rest of the Application layer through dependency injection.

Business Logic

The majority of the business logic for the webapp has been encapsulated in three DLL assemblies in the infrastructure layer: Commands, Queries, and Utilities.

Connecting the application layer with the data access for a webapp is a common task, and one that is often handled by means of following the Repository and Unit of Work design patterns. It is, however, not the only way, and [Ham17] discusses how one might make use of the Query-Command design pattern instead to achieve the same goals.

This project has adopted using the Query-Command pattern in order to clearly encapsulate the well-defined responsibilities that must be handled when the Application layer access the database. This means that read-actions towards the database always go through a derived class of the abstract `Query` class, and write actions always go through a derived class of the abstract `Command` class.

The abstract base classes `Query` and `Command` wrap the execution of the derived class in order to carry out any tasks common to all of either the Commands or Queries, before the derived class is called. While this feature is not used

in this solution, it is the basis for further development that will likely include authentication for the actions based on technologies such as API keys.

This feature is shown in figure 4.6. The execution function that the derived classes implement is protected and not called by the user of the Query or Command class directly. Rather it is called after the public Execute function is first called, wherein the common tasks can be done.

```
19 references | Andreas S. Jensen, 37 days ago | 1 author, 2 changes | 0 exceptions
public void Execute() {

    // Add common tasks, authentication, and authorization here

    DoExecute();
}

13 references | Andreas S. Jensen, 37 days ago | 1 author, 1 change | 0 exceptions
protected abstract void DoExecute();
```

Figure 4.6: Code snippet of how the base Query and Command classes wrap the derived execution in a function that the user calls first. Code found in Query.cs (and Command.cs).

Another common feature of this design pattern is also a handler class that process a queue of commands or queries, since they all derive from a common base class. Such a feature is not used in this project, and was not considered when choosing between this pattern and the Repository and Unit of Work patterns.

As such, the implementation of commands and queries in this project can be seen as bloating up repository functions into separate classes; after all they do not handle anything a repository and a unit of work could not do. Such an observation is justified in its current state. Though this choice does also come with the added bonus of making unit testing more concise, creating a unit test class for each command and query and only mocking their dependencies, as opposed to repositories where all of the dependencies must be mocked for the repository test class.

Data Model Implications & Fatal Flaw

While the need for duplication of the general clothing models (Relation, Size, and Brand) has been clear from the beginning of the data model being determined, as mentioned in section 4.2.3, it is here in the Infrastructure layer that the other implication of this chosen model comes into play and shows its flaws.

Figure 4.7 shows a code snippet from the comparison query class for Sneakers (`SneakersComparisonQuery`). Because the current data model has each sepa-

rate brand as a column in the relation tables, and the comparison result is only interested in rows where the desired brand actually has a (size) value, the code does not have one generic column it can check for a value or a specific brand ID, as would be the case with the alternate (second) data model suggested in section 4.2.4. Instead the code must use a switch case on the desired brand, with each case altering the query to look for entries with a value specifically in *their* column, since these columns are defined by the class properties (AdidasOutdoorSize, AdidasGolfSize, and AltraFootwearSize for the `SneakersRelation` class).

```
// Entries with requested result brand
switch (resultBrandId) {
    case SneakersBrand.IdEnum.AdidasOutdoor:
        query = query.Where(s => s.AdidasOutdoorSize != null);
        break;
    case SneakersBrand.IdEnum.AdidasGolf:
        query = query.Where(s => s.AdidasGolfSize != null);
        break;
    case SneakersBrand.IdEnum.AltraFootwear:
        query = query.Where(s => s.AltraFootwearSize != null);
        break;
    default:
        throw new ArgumentOutOfRangeException();
}
```

Figure 4.7: Example of a switch case required under the current data model in order to correctly sort a comparison request after a desired brand. Code found in SneakersComparisonQuery.cs.

This is problematic not only because it takes up a lot of code, and this sort of switch cases are found quite a few places in the commands and queries; it is problematic in the sense that it puts a lot of responsibility on the developer to ensure that all possible brands are processed in the switch case. This is also makes it very important that an exception is thrown as the default case so as to inform the developer that maybe a case is missing.

This implication of the current data model has turned out to be such a flaw that it *should not* be part of a solution undergoing further development and into proper production. It is a flaw risking unmapping of part of the data domain model and one that can happen many places in the code; such switch cases are undoubtedly not a sustainable way to expand and scale this prototype.

It is however a flaw discovered so late in development that a full change of the data model has not been feasible. While the prototype lives up to the

requirements, as section 5.2 will show, it does so on a flawed data model.

4.2.5 Application - WebAPI & MVC

The Application layer of the architecture model hosts the executable for the service system, which is a webapp for this project. Traditionally with ASP.NET an MVC website and a WebAPI webservice were two different project types, but under ASP.NET Core these two have been combined and can easily be hosted on the same webapp.

The package diagrams for this DLL can be found in appendix F.2.3.

WebAPI & MVC Area

ASP.NET Core implements WebAPI and MVC in much the same manner, mapping URI paths to classes deriving from a Controller base class and its functions, as explained in [Mic17e]. This can easily make the code very confusing to navigate with no clear destinction between what is a webservice controller and what is a website controller, except for whether they return a View or a JSON object.

To help organise the code the MVC website logic has been put under an Area (see [Mic17a]), specifically named 'Admin'. This is evident in the code by all of the website classes being located in the CloseFit.Application.WebApi/Areas/Admin folder. When using the website it is evident from the URI. Where the standard ASP.NET Core routing is by the scheme /{Controller}/{Action}/ the routing for an area is /{Area}/{Controller}/{Action}/, where Area, Controller, and Action are the names of the area, controller, and action[9]. Thus all the website URIs start with /Admin.

Because most of the business logic is delegated to the code located in the Infrastructure layer, the webapp itself does not do much more than basic authentication (see next section), mapping input and view models to the domain models neede by the Command and Query classes, and returning either JSON objects or View objects, depending on whether a WebAPI or MVC controller was called. In the case of the latter, ASP.NET Core's Razer View Engine is used to parse the .cshtml view files into HTML and return it to the requesting browser.

The mapping from input and view models to the domain entity models (and vice versa) is done via the AutoMapper library[10].

[9] In ASP.NET Core these names can either be mapped explicitly with the [Route] attribute, or they can be inferred from the class and function names.

[10] http://automapper.org/

Website Authentication & Authorisation

In order to fulfil the requirements and expose a website for admin access to the stored data, the webapp also includes basic authentication and authorisation access. While use of the webservice is open to everyone in this prototype, access to the website (beyond a login screen) is restricted to two accounts: an admin account that has full Create, Read, Update, Delete (CRUD) access to the data, and an analyst account that only has access to view the stored data.

This latter account has been added in order to show off how authorisation schemes work in the framework when there are more than one, and is based on [Mic17d]. The admin and analyst account have two different roles, but rather than using these roles directly to authorise access to controller classes and their action functions (using the [Authorize(Roles = "..")] attribute), two policies have been defined in the webapp's `Startup` class as shown in figure 4.8.

```
// Authorization schemes
services.AddAuthorization(options => {
    options.AddPolicy(AuthConst.Policies.View, policy
        => policy.RequireRole(AuthConst.Roles.Admin, AuthConst.Roles.Analyst));
    options.AddPolicy(AuthConst.Policies.Edit, policy
        => policy.RequireRole(AuthConst.Roles.Admin));
});
```

Figure 4.8: Defining the authorisation schemes for the webapp. Code found in Startup.cs of the WebAPI project.

With `AuthConst` simply being a struct holding string constants, the authorisation defines two schemes: *View* that requires the user to either have an admin or analyst role, and *Edit* that requires the user to have an admin role. The controller classes and functions are then annotated with authorisation attributes such as [Authorize(AuthConst.Policies.View)] in order to specify that the user must fulful the View policy in order to use the required controller class or function.

All of this functionality is part of ASP.NET Core's Identity framework as explained in [Mic17c]. This framework has two primary functionalities:

- It adds a set of extra tables to the database in order to store identities (with hashed password field), roles, etc.
- It injects a User identity into all HTTP contexts when the user access a WebAPI or MVC Controller class. This User identity is foremost used to check authorisation, but is still available to be retrieved by the software code.

This latter point is particularly used in the website view files (.cshtml extension) where the user's permissions are checked to change whether the HTML contains

links to create, edit, and delete data (in the case of an admin).

Data Persistence Services

Because the requirements have no mention of being able to sign up new users, and this functionality is not implemented in the webapp, there is the problem of ensuring that the two pre-defined users–the admin and analyst–exist in the system when the service system is deployed for the first time and creates a fresh database.

This persistence of data also comes in use with the clothing categories' meta data (brands and sizes), since these are based on enums in the code and stores to a database for convenient sake; thus updates to the enums in the code should ensure that the system, when being restarted/redeployed, would update the database as well.

In order to ensure this data persistence, services have been created that are executed in the end of the webapp's `Startup` class, and thus in the end of its startup cycle.

Specifically, the two users and their roles are ensured in the `Startup.EnsureRolesAndSystemAdmin` function, while the meta data is persisted using the `DatabaseMetaSynchronization` class in the Services folder.

It should be noted that the current prototype does not implement this data persistence for all of the meta models (due to time constraints on the development), only the `SneakersSize` and `SneakersBrand` models. For anything but a development environment these services should be finished and integrated in order to properly persist the meta data to the database upon database creation.

Error Handling

The error handling has had to be implemented in a special way. While ASP.NET Core has standard error handling for generating error webpages and error responses in the case of WebAPI, the project makes use of custom errorhandling middleware as suggested by [And17a].

This is based on ASP.NET Core's middleware technology (see [Mic17b]) that allows the software to inject code into the pipeline processing an HTTP request. This is done for this project with the `ApiMvCErrorHandlerMiddleware` class and the `ApiMvCErrorHandlerMiddlewareExtensions` class in the Utils folder.

The Extensions class uses a static function to add the custom error handling middleware to the pipeline, but before doing so it either adds ASP.NET Core's standard error handling as well, or more verbose error handlers for development, depending on whether the startup of the webapp is set to a development

environment or not. By doing this the webapp retains the more-than-sufficient error generation of the framework.

However, since the custom error handling middleware was added later in the pipeline, this code will catch an error before the standard error handling does so[11]. By doing this, the project is able to inspect the request that generated an error–or rather, an exception which was not caught in the controllers.

Because there is both a webservice and a website hosted in the same webapp it would not make sense to send an error page's HTML content as a response to a webservice call that generated an error. Thus the custom error handler checks if the request was on the webservice path (which has a common "/api" URI root by convention), and if it does it generates either a HTTP 400 or HTTP 500 error response based on the exception, with the exception message as the response body.

If the error-generating request is not related to the webservice the exception is left propagating back to the default error handling middleware that was added, letting it generate the error HTML pages.

Logging

Logging is of course implemented in the project as well in order to offer a history of both general information and errors produced by the system. For this project NLog.Web is the logging library used, as per [Rod17]. It is a logging framework specifically created to work with ASP.NET and ASP.NET Core, offering powerful control over logging content, style, placement, and more. All through a simple xml config file (nlog.config in the root of the Application.WebAPI project).

Using the framework, the solution's WebAPI and MVC controllers log to a app.log file in C:/CloseFit/Logs/, and if the current log file is older than the day at which a new entry will be made, the old log file is archived in a Archive subfolder named with a timestamp at the time of the archive. This folder will store log files up to a year before deleting them. All of these settings are set in the nlog.config file and can easily be changed even when the service system is deployed on a server.

Website Resources and Minification

Lastly it shall be mentioned that the background image used for the website is with the permission from [Pix17], and in order to automatically minify the website's static files (js and css scripts) the Gulp framework has been used as per [Mic17g].

[11] The later middleware is added to the pipeline the earlier it receives an error risen in the executing code.

4.2.6 Further Development

Data Model

While there are many aspects of the service system that could and should be further developed before the solution is ready for production, the most pressing matter is that of the data model, as explained in sections 4.2.3 and 4.2.4. Three immediate approaches would be to either:

- Implement the second solution from section 4.2.3 to see if the result is more maintainable with domain expansions, or
- conceive a third and different data model that does not suffer the disadvantages of the two first models, or
- explore the posibility of using the .NET Reflection library.

The two first options are self-explanatory, but the third option is rather different. With keeping the current data model, the goal for this approach would be to try and use the build-in Reflection library[12] of the .NET framework to process the entity model objects not as strongly-typed objects, i.e. reference their properties directly, but rather retrieve a collection of properties for a given object that are of a certain type. This gives the flexibility to treat all of the column-properties of the entity model (the different brands) as one collection rather than defined individual references.

An example of this is already used in the solution. Figure 4.9 is taken from the Command-derived class used by the website to create a new T-Shirt relation. Because it does not make sense to make a relation that has less than two entries in it, the command verifies that this condition is upheld.

```
int nonNullCount = NewRelation.GetType().GetProperties()
    .Where(p => p.PropertyType == typeof(TshirtSize))
    .Count(p => p.GetValue(NewRelation) != null);
```

Figure 4.9: Example of how reflection can be used to process all properties of a given type, as a collection. Code found in CreateTshirtWebsiteCommand.cs

Doing so with strongly-typed references would require a series of if-blocks that incremented a counter if their specific property had a value (for the prototype there are four T-Shirt brands, so four such properties on the relation entity model). With Reflection, the .NET framework uses the meta data of the class type to return a collection of properties that have a given type (here `TshirtSize`), and then count the number of properties for which there is a value on the relation-to-be-created object (NewRelation variable in the figure).

[12] https://msdn.microsoft.com/en-us/library/f7ykdhsy(v=vs.110).aspx

User Data Submission

The prototype does not implement any authorization on the data being submitted to the webservice. While this was out of scope for the prototpye, further development should evaluate whether data submitted from users is considered reliable, or if the service should only open up to accepting data received from webstores, in which case API keys may be a means to identify and authenticate the users trying to post data via the webservice.

Data Persistence Services

As mentioned earlier in section 4.2.5 only part of the data persistence services are implemented in regards to the domain model's meta data. These services should be a priority in further development of the solution in order to ensure a functional database both during development and production.

Tests

And last but not least, tests for the service system should be added and revised, as only out-dated unit tests for the Infrastructure layer exists in this prototype. Section 5.1 touches on this further.

4.3 Service Interface

In order to demonstrate how the service system could and should be used to fulfil the requirements for the project, a concrete service interface unit has been development as a proof of concept. As per the requirements, this unit is a mobile app.

4.3.1 Framework - Android

The prototype for demonstrating a service interface was chosen to be an Android app because it is a framework familiar to the author, in contrast to other mobile platforms such as iOS.

In order to reach a wide target of end users with the application it has been set to a minimum version of Android 4.4, which according to [And17b] covers approximately 87% of all Android users as of March 6, 2017. By doing so the app is compatible with the vast majority of Android devices currently in use, while still having access to the newer features of the platform since Android apps are limited by their minimum-required SDK version. Usage of API features newer

than the required minimum have to be escaped with additional visioning checks and alternate execution paths in the code for old incompatible devices.

4.3.2 System Architecture

From the start of the development process the goal was to keep the service interface prototype as simple as possible with its purpose only being to demonstrate how the service system would be integrated into from the outside.

Because of this philosophy a simple architecture for the Android app has been chosen from the beginning. As shown in figure 4.10, by segmenting the code into its three core purposes, compare, upload, and meta (and a util directory for shared code responsibilities), it could be cleanly organised without sacrificing readability or maintainability of the codebase. Thus the responsibilities have been encapsulated by the packages as follows:

- `dk.asj.closefit.meta`: Models and their service logic related to the service system meta data.
- `dk.asj.closefit.compare`: Models, services, and views related to a comparison request.
- `dk.asj.closefit.upload`: Models, services, and views related to uploading new data to the service system.
- `dk.asj.closefit.util`: Models shared across the other base packages.

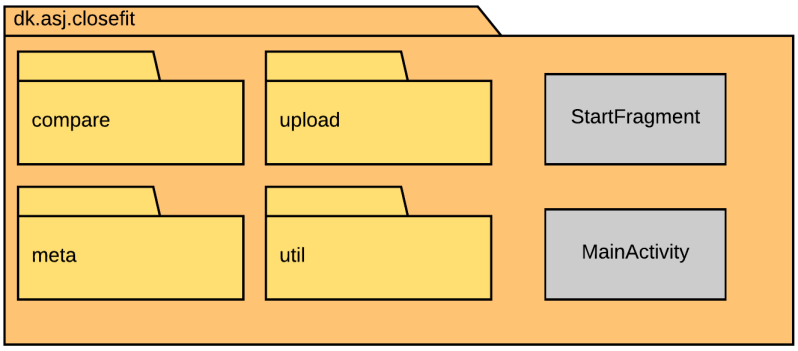


Figure 4.10: Overview of the Android app code structure seen from the root package dk.asj.closefit.

In these packages domain models, services, and views are themselves gathered in appropriate sub-packages to help keep a clean structure of the codebase.

Beside the structure of the code, the architecture of the software follows the pattern common to Android apps, in that singleton classes are used to maintain

the business data of the app when it is required by different views. This is also the case for the CloseFit app, where two singletons manage the data to be uploaded to the service system, or the data to be used for a comparison query (the `UploadSingleton` and `RequestSingleton` classes respectively).

The views of the app are contained in one Activity through the use of Fragments, with the SupportFragmentManager used to navigate between views[13].

Appendix F.1 contains package diagrams for the app and serves to give a more wholesome overview of the content abstracted in figure 4.10.

4.3.3 Integration to the Service System

HTTP

The system overview in figure 4.1 emphasises how both the browser and the Android app integrates with the service system through HTTP. This is of course due to the web application exposing a website and a webservice. As such the CloseFit app also uses HTTP to communicate with the RESTful webservice and make use of it.

While the the standard Android library has its own classes for consuming web content, the Spring Rest template, as explained in [Sof17], has been used for the CloseFit app instead. The Spring framework has strong integration with Java and also Android, and offers a much more simplified way to consume RESTful webservices such as the one the service system exposes. It also makes use of type safety when parsing objects, and can handle a variety of parsers in different formats. The one used in this project is the Jackson JSON converter recommended in [Sof17], as the webservice returns JSon objects.

The web communication of this framework is still not asynchronous, however, meaning that the use of the Spring RestTemplate class is blocking the executing thread when it is reading and writing on the HTTP protocol. This discourages the direct use of the framework on the UI (main) thread of the Android app as it would result in the app becoming unresponsive while web content is being processed. As such, the Android framework's build-in AsyncTaskLoader[14] features are used to delegate the consumption of the RESTful communication to separate threads from the UI.

The app has three such loaders, `EntityMetaModeLoader` for GETing the meta data for the supported clothing types, `RequestResultLoader` for POSTing a comparison request and receive the result, and `UploadReferenceLoader` for POSTing new data.

[13] https://developer.android.com/guide/components/fragments.html
[14] https://developer.android.com/reference/android/content/AsyncTaskLoader.html

Presenting the Results

With the `RequestResultLoader` used to get a comparison request result from the service system, the data is available to the app as a `ResultGraph` object. While this could simply be displayed to the user in a textual manner, the author has valued a graphical representation of the results both for a better user experience and for a more easy overview of the result. For this the 3rd-party library of [Geh17] was used to generate a basic bar graph, an example of which is shown in figure 4.11.

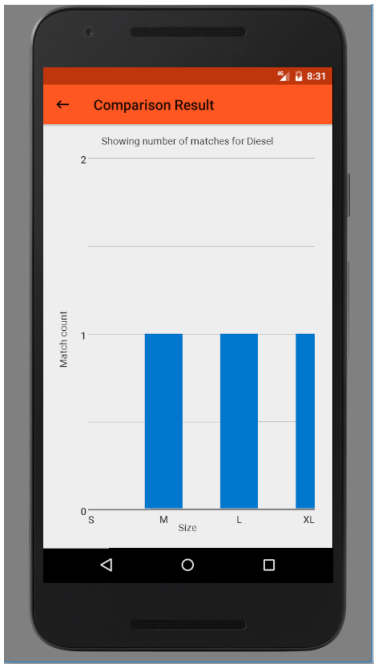


Figure 4.11: Example of the bar graph displaying the result of a comparison request.

The results delivered by the `EntityMetaModeLoader` class is an `EntityMetaModelList` object, a list encapsulating the meta data for each available clothing category. The structure of the meta data is generic between the different categories: there are two URIs–one for upload and one for comparison request, an array of (name,ID) pairs for brands, and an array of (name,ID) pairs for sizes, as shown in figure 4.12.

```
{
  "entityMetaModels": [
    {
      "comparisonRequestUri": "/api/tshirts/compare",
      "dataInputUri": "/api/tshirts/upload",
      "categoryName": "T-Shirts",
      "brands": [
        {
          "id": 0,
          "name": "adidas"
        },
        {…},
        {…},
        {…}
      ],
      "sizes": [
        {
          "id": 0,
          "name": "S"
        },
        {…},
        {…},
        {…}
      ]
    },
    {
      "comparisonRequestUri": "/api/sneakers/compare",
      "dataInputUri": "/api/sneakers/upload",
      "categoryName": "Sneakers",
      "brands": [
        {
          "id": 0,
          "name": "adidas Outdoor"
        },
        {…},
        {…}
      ],
      "sizes": [
        {
          "id": 0,
          "name": "38"
        },
        {…},
        {…}
      ]
    }
  ]
}
```


Figure 4.12: Example of a response to a meta request, showing the different clothing categories available and their included meta data in sizes and bands (some of which are minimised for viewving in this figure).

Because of this generic structure the app does not have to use different models or services to process the different clothing categories. As such the upload and compare packages do not have separate T-Shirts and Sneakers classes. By doing so the app can remain unchanged and uncaring as to which categories are added to the service system's domain, so long as they are included in the meta response. This meta data is then generically parsed into the upload and compare views as needed, being stored in the `UploadSingleton` and `RequestSingleton` respectively.

Lastly, the `UploadReferenceLoader`'s result has no advanced viewing related to its display. The the response from the service system upon data being uploaded

is simply a Boolean TRUE for the data was saved, or an error–either HTTP 400 if the input data was faulty or HTTP 500 if an internal error occurred. These results are simply displayed on the app in textual format.

For a more complete overview of the various screens/views in the app, see the acceptance tests in appendix E.

4.3.4 Further Development

With the CloseFit app only being a proof of concept there are obvious areas that could and should be touched upon with further development in order to ready the software for full public production. There are the common tasks such as obfuscation and minifying of the source code[15] in order to make the app as small as possible while removing unused code and help making reverse engineering of the app's code more difficult (unless the app should remain open-source). A more polished looks of the app if also a given.

Though the functionality of the app is covered, and as section 5.2 will show the requirements are fulfilled, automated tests should be written for the code since it lacks unit tests in its current state. These would help ensure proper functionality of any expanded code. Furthermore, as discussed in section 4.2.6, it should be evaluated whether data submitted by users is reliable and should continue to be permitted, and possibly do so with the use of registered API keys, one which should be a static part of the app in such a case.

[15] https://developer.android.com/studio/build/shrink-code.html

5. Testing

Testing of any software system, whether during development or in production, is important to ensure that faults, bugs, and missing functionality is kept to a minimum. This project has made use of two overall testing methods during different stages of the process: unit testing and acceptance testing.

Beside these two methods of testing, the system has also undergone continuous manual testing during its development. Rather than having a set framework to automate tests and ensure functionality, this method is simply the developer executing part of the program to ensure newly added/changed code behaves as expected. Unit testes automate this process but also require the added resources of writing the tests, and the tighter the coupling of the code-under-test the more difficult it becomes to write automated tests due to the dependant-upon code most be mocked.

5.1 Unit Testing

When the development of the service system began unit testing was originally implemented in order to test the Query and Command assemblies (CloseFit.Infrastructure.Queries.dll and CloseFit.Infrastructure.Commands.dll respectively) since these parts of the code implemented the vast majority of the system's business logic. Any communication to the database would go through these assemblies regardless of whether the origin of the execution was from the webservice endpoints or the website endpoints.

These unit tests have however not been kept up to date and are excluded from the build order/compilation of the system code base (the visual studio solution). This has been a deliberate oversight due to time constraints and rapid development of the service system, but for continued development these tests should be expanded, corrected, and reimplemented in order to ensure automated tests of the system's business logic, and thus help ensure the compliance of the acceptance testing.

The outdated unit test code directories are found in the *CloseFit/src/* directory and follows the naming convention *[assembly-under-test].Tests*. Thus there are two, one for Commands and one for the Queries respectively.

The unit tests were implemented using the xUnit.net framework [Fou17] due to its strong integration with the code language and development environment of the service system. In order to not alter the development database during each execution of a unit test, the unit tests made use of a mock of the database context which worked on an in-memory database, created specifically for each test function, as per [Mic17f].

5.2 Acceptance Testing

Acceptance testing is an important an integral part of a the final stage of a piece of software, making sure that its compliance to the system's requirements are met. This is the case also for this project, even as proof-of-concept prototype.

The tests, carried out by the author, are detailed in appendix E. A test is made for each use case since the use cases map directly to the system's requirements, as specified in section 2.1. Therefor, being able to document the use cases by showing the system's expected behaviour ensures that the requirements are met.

The full test sequences are found in the appendix, but table 5.1 shows an overview of each of the tests: where it can be found and the status of it, as well as which use case it tests and a short description of the use case scenario.

Use Case ID	Description	Appendix	Result
UCI1	A user makes a comparison request with an interface	E.1	Passed
UCI2	A user uploads data with an interface	E.2	Passed
UCS1	The system retrieves clothing meta data.	E.3	Passed
USC2	The system executes a comparison request	E.4	Passed
UCS3	The system saves uploaded data	E.5	Passed
UCS4	An admin logs in	E.6	Passed

Table 5.1: Results of acceptance testing and where to find the test.

5.2.1 Nonfunctional Requirements

Because the acceptance testing does not map directly to the non-functional requirements (these are not mapped to the use cases), this list serve to document the fulfilling of the nonfunctional requirements. For the list of the requirements, see section 2.1.2

NFR01 The acceptance tests have documented the system being accessible through HTTP both to an Android app and a 3rd-party program (Postman).

NFR02 The exposure of the service as a RESTful HTTP webservice makes it easily intergratable with a web shop CMS plug-in.

NFR03 Usage of the webservice is not authenticated and requires only HTTP access.

NFR04 An Android app exemplifies how the webservice may be used by a user.

NFR05 The website exposed alongside the webservice is accessuble only through login, either as admin (with write access) or an analyst (with read access).

NFR06 The service system uses an industry standard ORM to store its data in an SQL server locally on the server on which the system is hosted.

6. Conclusion

6.1 Proof of Concept

It has been shown and documented through sections 4 and 5 that the product of this project lives up to the requirements set out for the problem domain and successfully demonstrates a proof of concept on how a solution might work: Exposing a webservice for comparing clothing articles of given brands and sizes to that of other users and their matches, offering access to these data through an authenticated website, and exemplifying how this webservice might be used through a mobile app.

The solution makes use of pure statistical presentation of data more so than any data processing, simply showing a user the distribution of the service's data for which there is a match between the user's reference data and desired data, i.e. brands and sizes that matches the user and what sizes have generally been seen fitting for other users for a given brand.

These have been achieved with the use of the ASP.NET Core framework for developing a web application exposing both a webservice and a website at the same time, and with an Android App making use of the webservice to present and add data.

6.2 Future Improvements

It has also been documented in section 4 how this proof of concept has many areas in which further development is both advised and even required before the solution is in any production state. Most notable is the need for a considerable revising of the data model proposed and used in this prototype, due to its lack of proper scalability of the data domain model. As well as the services that persist the domain meta data to the database used for storing the application data.

Appendices

A. Solution Resources

Solution Code

The code can be downloaded at http://88.99.126.154/source.zip and contains both the Android and the ASP.NET Core codebase. The code has been developed in Android Studio and Visual Studio respectively, and as such the raw source code can be found in the folders:

- Android app: closefit_app/app/src/main/java/dk/asj/closefit
- ASP.NET Core webapp: CloseFit/src

Solution Product

The Android app for the service interface can be downloaded at http://88.99.126.154/closefit.apk. This file must be run on an Android phone to be installed. Requires at least Android 4.4.

The service system is accessible on http://88.99.126.154/CloseFit/ (and the web-deploy zip[1] can be downloaded at http://88.99.126.154/closefitwebapp.zip). To access the website on http://88.99.126.154/CloseFit/Admin there are two users:

- Admin: Username *admin*, password *EditCloseFit17*.
- Analyst: Username *analyst*, password *ViewCloseFit17*.

It is not recommended to try and deploy the service system locally, but if needed it is most easily done by debugging the code through Visual Studio, thus running the system on IIS Express with an MSLocalDB database. If IIS Express is not able to launch the program it may be needed to change the launch port for the application. Do so by editing
CloseFit/src/CloseFit.Application.WebAPI/Properties/launchSettings.json
and change applicationUrl value with a port other than 8080.

The webservice is accessible through http://88.99.126.154/CloseFit/api and has five endpoints (URI added to the end of the previous URL):

- [GET] /meta: Get the current meta data.
- [POST] /tshirts/compare: Comparison request for T-Shirts.
- [POST] /tshirts/upload: Add new T-Shirts relation data.
- [POST] /sneakers/compare: Comparison request for Sneakers.
- [POST] /sneakers/upload: Add new Sneakers relation data.

[1] https://www.iis.net/downloads/microsoft/web-deploy

For the /upload endpoints JSON input data must be given (in the body) in the format from figure A.1. The integer values are IDs and valid values are in the range [0;3] for T-Shirts, and [0;2] for Sneakers.

```
{
    "InputData" : [
        {
            "Brand" : 0,
            "Size" : 0
        },
        {
            "Brand" : 1,
            "Size" : 1
        },
        // ...
    ]
}
```

Figure A.1: Data format of a HTTP POST request to any of the ../upload webservice endpoints.

For the /compare endpoints JSON request data must be given (in the body) in the format from figure A.2. The integer values are IDs and valid values are in the range [0;3] for T-Shirts, and [0;2] for Sneakers.

```
{
    "ReferenceData": [
        {
            "Brand": 2,
            "Size": 3
        },
        // ..
    ],
    "ResultBrand": 1
}
```

Figure A.2: Data format of a HTTP POST request to any of the ../comapare webservice endpoints.

B. Project description

Copy of the project description follows on the next page.

Project title

CloseFit

Project description

With a lack of proper sizing and measurement standards for some clothes, picking out clothes that are a good fit can be difficult without physically trying them on. This makes online shopping for clothes a burden and can lead to bad user experiences with having to return goods due to inaccurate measurements.

With CloseFit, the purpose is to make the user experience of shopping clothes online more enjoyable. By helping to make a better choices between clothes in different sizes, the goal is to achieve better accuracy for acceptable fits, and thus lower the amount of sales that require returning.

CloseFit is intended to be a standalone online service that customers and web shops alike can utilize to achieve a better user experience and sales that are more accurate.

C. Work Log

This appendix contains a list documenting the work performed throughout the project period and serves as means of documentation. Each log entry is on a weekly basis and only covers the general work topics of each week in favour of the extensive material a day-by-day log would be.

Week 45 - 50 2016
These weeks saw little active work on the project due to time constraints on other projects and courses from the University which took priority. For the project, these weeks mainly contained research into technologies, frameworks, and development strategies. The initial conception of the project, its purpose, and requirements.

Week 51 2016
Finished milestone plan, project start, and research in ASP.NET Core for combined WebAPI and MVC codebase.

Week 52 2016
Preliminary project scope phase with problem analysis and project delimitation.

Week 1 2017
Further development of the project scope; producing an SRS for the project and use cases mapping to the requirements.

Week 2 2017
Finished project scope phase (milestone M1) and begun design phase (milestone M2); analysing the use cases for system architecture and initial domain design.

Week 3 2017
Experimented with different data model designs for the domain model and database design. Problems regarding type-safety in Entity Framework and performing a comparison request.

Week 4 2017
Data model decided upon. Data access and initial business logic (command and query) created with unit tests. Basic WebAPI started. Live-testing with Postman.

Week 5 2017
WebAPI expanded as well as business logic. Unit tests left behind for now in favour of testing with the android app having started initial design.

Week 6 2017
WebAPI and android app functionality finished along with their required business logic (commands and queries). Integration between app and WebAPI working fully.

Week 7 2017

Started on MVC website and expanding business logic with required new command and query classes. Added authentication and authorisation to website as well as identity persistence service to the service system.

Week 8 2017
Added example meta data persistence service to service system. Custom error handling to take care of sending back error messages for WebAPI calls. Added Logging to both website and webservice controllers. Finished website.

Week 9 2017
Fixed last code bugs. Stylised app and website with color theme and minor UX elements. Finished integration phase (milestone M3).

Started preparation of hosting environment on remove server for service system.

Week 10 2017
Published service system to remote hosting server. Finished overview- and package diagrams. Finished report writing and revising. Final product and report ready (finished milestone M4) and ready for hand-in (milestone M5).

D. Use Cases

Here follows the system's use cases that are used to model the abstract system behaviour and documents the SRS functional requirements' needed involvement in the system.

The use cases make use of a few different users for roles: An end user is a person interacting with a service interface, and a service user is an entity making use of the service system's web API directly. This distinction is made because the use cases are separated into two sub-systems, the service interface and the service system, denoted with UCI and UCS IDs respectfully; the roles that interact with these two systems may differ.

As shown in the use case diagram in figure D.1 there are also an *Admin* role and a *Web API User* role. The former is self-explanatory, while the latter is a user that makes use of the service API directly, e.g. through a browser, and not through one of the system's service interfaces.

Each use case is marked with a *scope* to further identify which subsystem they address.

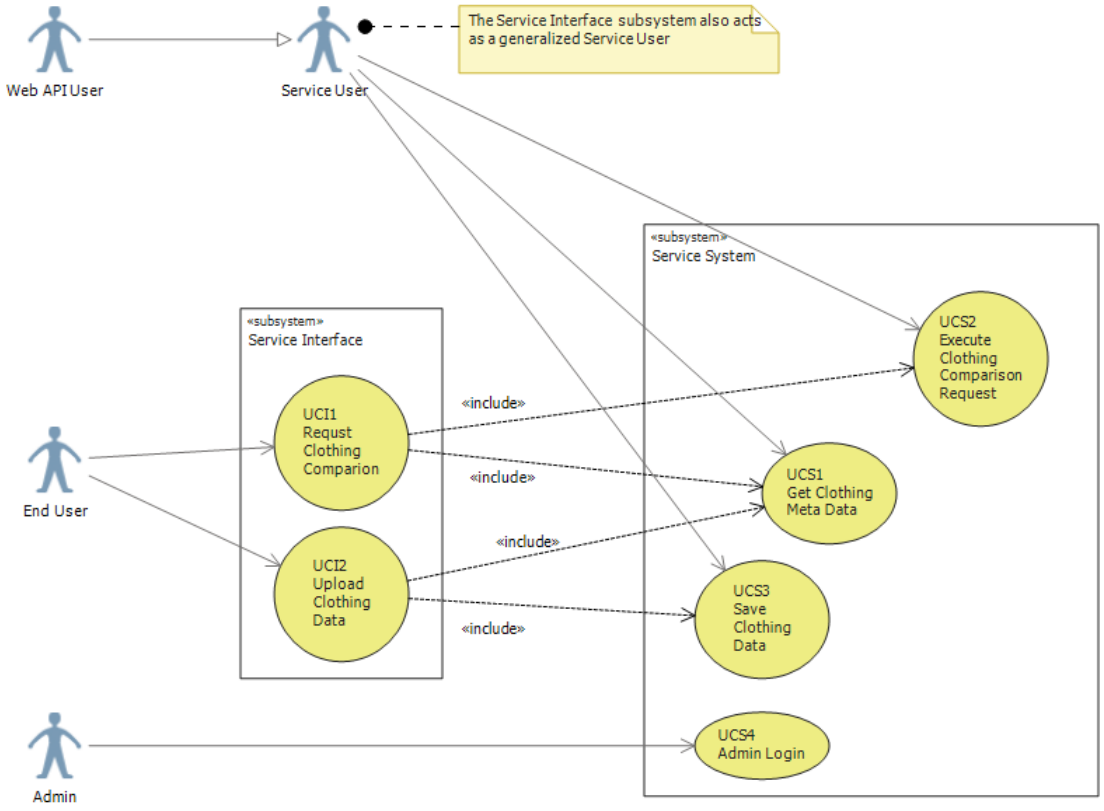


Figure D.1: Use case diagram highlighting associations between use cases, roles, and subsystems.

D.1 UCI1 Request Clothing Comparison

ID
UCI1

Description
An end user requests a comparison between submitted clothing data and the data stored in the system, and receives the result.

Scope
Service interface

Primary actor
End user

Preconditions
The end user has access to a service interface.

Postconditions
The end user has been presented with the result of the comparison.

Trigger
The end user notifies the service interface that he wishes to compare clothing data.

Main flow

1. The service interface requests meta data from the service system.
2. *include(UCS1 Get Clothing Meta Data)*
3. The service interface constructs a comparison request form from the received meta data.
4. The service interface presents the end user with the input form.
5. The end user fills out the form and submits it.
6. The service interface submits the comparison request, with its data, to the service system.
7. *include(UCS2 Execute Clothing Comparison Request)*
8. The service interface displays the received result for the end user.

Alternate flow

3. (a) No meta data
    i. The service interface informs the end user that there can be made no requests.
    ii. End of use case.
6. (a) No input data

i. The service interface informs the end user that no request was submitted.

ii. End of use case.

D.2 UCI2 Upload Clothing Data

ID
UCI2

Description
An end user uploads clothing data which is added to the system's stored data, to be used in future clothing comparisons.

Scope
Service interface

Primary actor
End user

Preconditions
The end user has access to a service interface.

Postconditions
The end user has received confirmation that the data was uploaded.

Trigger
The end user notifies the service interface that he wishes to upload data.

Main flow

1. The service interface queries the service system for submission meta data.
2. *include(UCS1 Get Clothing Meta Data)*
3. The service interface displays a submission form based on the received meta data.
4. The user fills in the form and submits it.
5. The service interface submits the data to the service system.
6. *include(UCS3 Save Clothing Data)*
7. The service interface displays the received result to the user.

Alternate flow

1. (a) No access to the service system
    i. The service interface displays an error.
    ii. End of use case.
3. (a) Empty meta data
    i. The service interface informs the the end user that no submission can be made.
    ii. End of use case.

5. (a) No data filled in by the user
    i. The service interface notifies the user that no data was submitted.
    ii. End of use case.

D.3 UCS1 Get Clothing Meta Data

ID
UCS1

Description
A service user requests, and receives, meta data on the clothing data–brands and possible sizes–stored by the service system.

Scope
Service system

Primary actor
Service user (web API user or service interface)

Preconditions
The service user has access to the service system web API.

Postconditions
The service user has received the clothing meta data.

Trigger
The service user sends a request to the service system for meta data.

Main flow

1. The service system receives the request.
2. The service system generates meta data of the brands and possible sizes available to the service.
3. The service system returns the meta data to the service user.

Alternate flow
None

D.4 UCS2 Execute Clothing Comparison Request

ID
UCS2

Description
The service system executes a service user's clothing comparison request, returning the result.

Scope
Service system

Primary actor
Service user (web API user or service interface)

Preconditions
The service system is running.

Postconditions
The service system has returned the result of the comparison to the user.

Trigger
The service user sends a clothing comparison request to the service system.

Main flow

1. The service system extracts the clothing data from the request.
2. The service system queries its stored data with the request-data.
3. The service system returns a comparison between the request- and queried data.

Alternate flow

1. (a) No submitted data
    i. The service system returns an error.
    ii. End of use case.

D.5 UCS3 Save Clothing Data

ID
UCS3

Description
The service system saves the new clothing data uploaded to it by a service user.

Scope
Service system

Primary actor
Service user (web API user or service interface)

Preconditions
The service system is running.

Postconditions
The service system has returned the result of the save action.

Trigger
The service user uploads clothing data to the service system.

Main flow

1. The service system stores the data.
2. The service system returns a confirmation to the service interface.

Alternate flow

1. (a) Incompatible data
    i. The service system ignores the incompatible data entry.
    ii. The use case continues from step 1.
   (b) No data to be stored
    i. The service system returns a notification that no data was stored.
    ii. End of use case.

D.6 UCS4 Admin Login

ID
UCS4

Description
An admin logs into the service system and is presented with CRUD options for the stored data.

Scope
Service system

Primary actor
Admin

Preconditions
The admin has access to the service website.

Postconditions
The admin is logged in and is presented with CRUD options for the service data.

Trigger
The admin navigates to the login website.

Main flow

1. The website prompts the admin to enter his credentials.
2. The admin enters his credentials and chooses to log in.
3. The website authenticates the credentials.
4. The website redirects to the authorised admin web area with CRUD options.

Alternate flow

3. (a) No input data
    i. The website displays an error to the admin.
   (b) Bad credentials
    i. The website informs the admin that the credentials could not be authenticated.

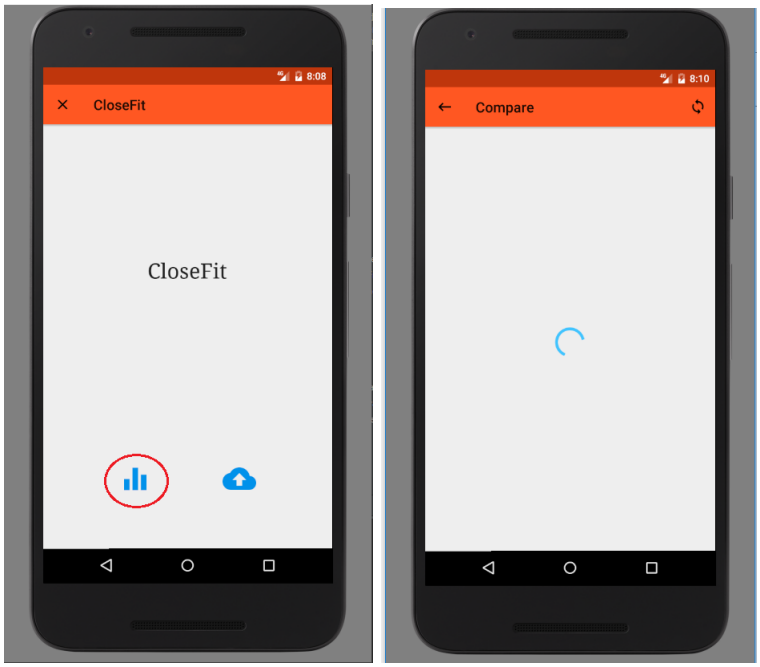
E. Acceptance Tests

This appendix contains the tests produced to verify the product's ability to live up to the requirements from the SRS in 2.1. This ability is documented by replicating each use case's steps in the system through screen-shots of the system behaviour. Emphasised controls are marked with a red oval, and use case steps that are not directly related to an action in the user's view is not documented unless it is on the service system (backend).

E.1 Use Case UCI1

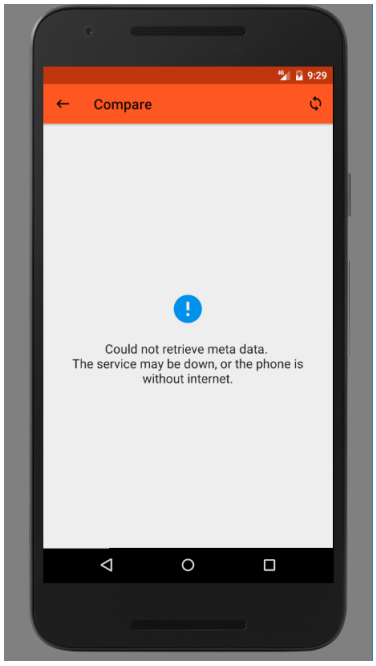
Tested by: *Andreas Stensig Jensen*
Expected result: *The user has been presented with the result of the comparison request.*
Test sequence:

1. Open the android app CloseFit and press the comparison icon in the lower left. While waiting for the meta data a loading screen is shown.

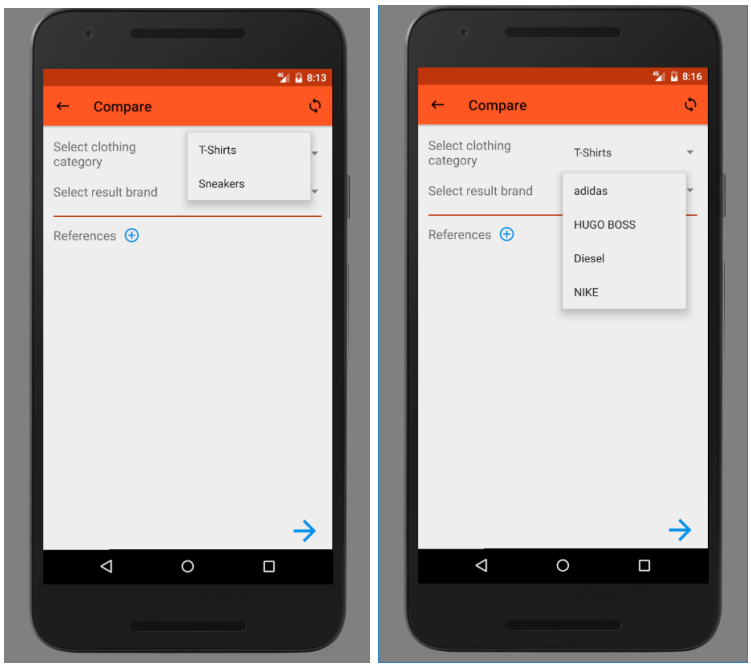


2. *See section E.3.*
3. The app parses the meta data. No actions required.

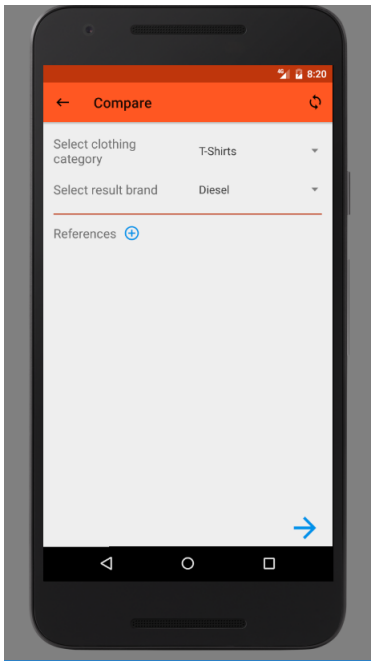
(a) If no meta data can be retrieved from the service the app shows an error.



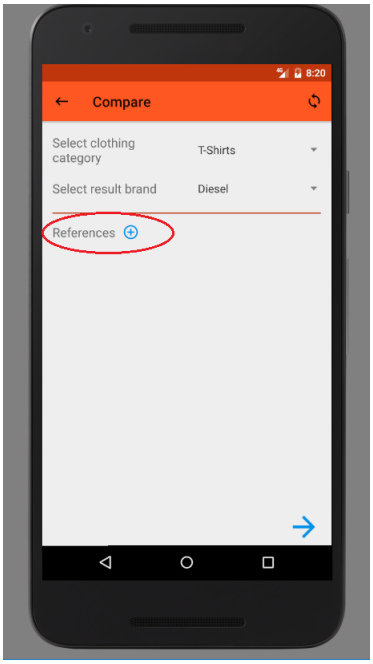
4. The app loads the meta data and prepares the view with options based on the data (content of categories and brand drop-downs shown). No actions required.



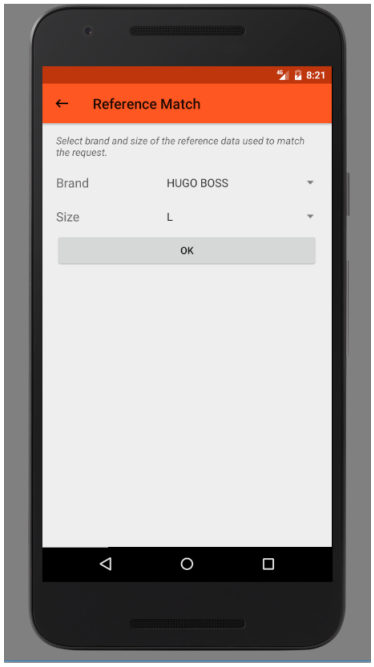
5. Select the category of the clothing type you are wanting to compare against, and the brand you wish a comparison for. Here T-Shirts and Diesel respectively.



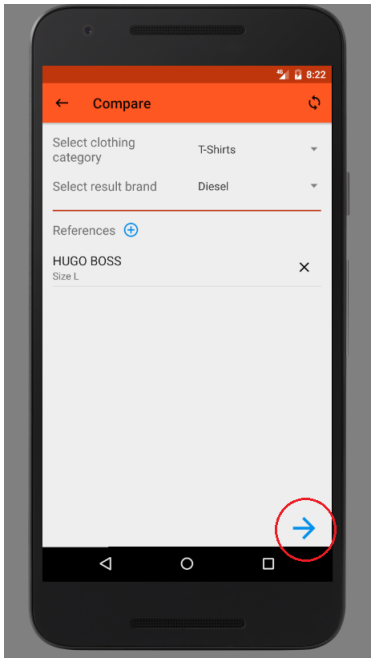
Open view to add reference data by pressing the blue plus next to 'References'.



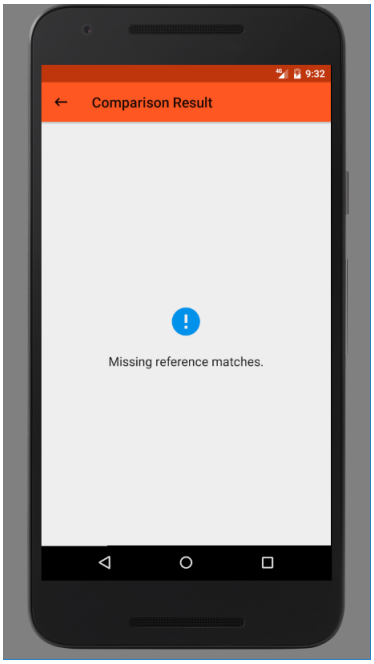
Select the brand and size, from the drop-downs, to use as a comparison reference. Here HUGO BOSS size L. Press OK.



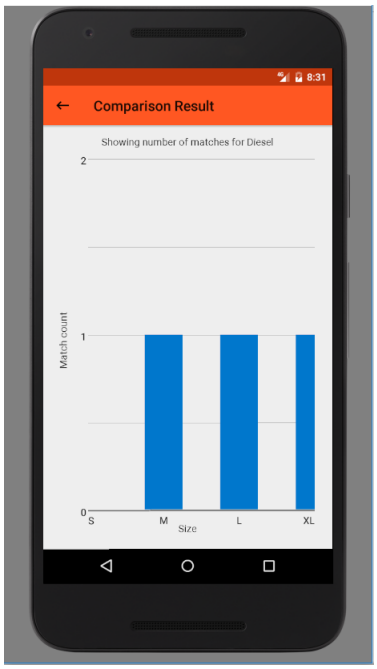
Submit the comparison request by pressing the blue arrow icon on the lower right.



6. A loading screen is shown while the app submits the request to the service system. No actions required.
   (a) If no references were supplied an error is shown.



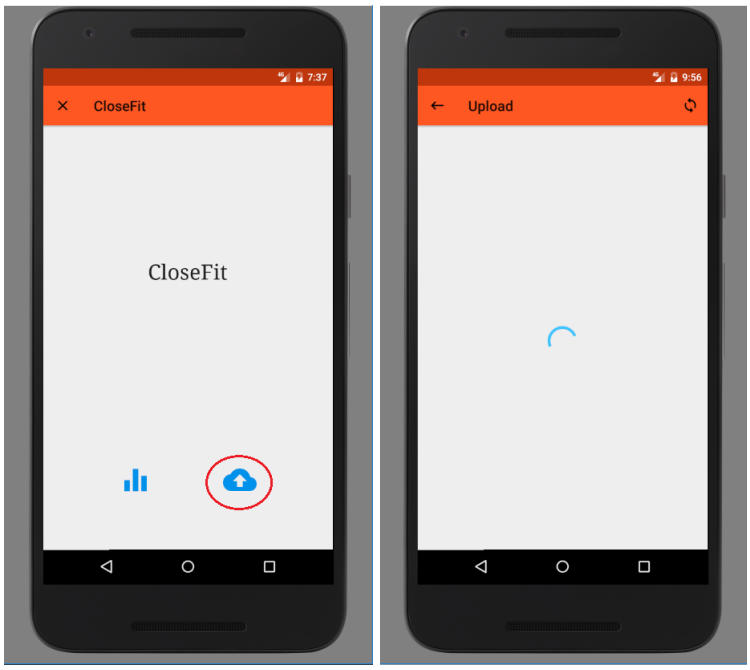
7. *See section E.4.*
8. The received result of the request is shown.



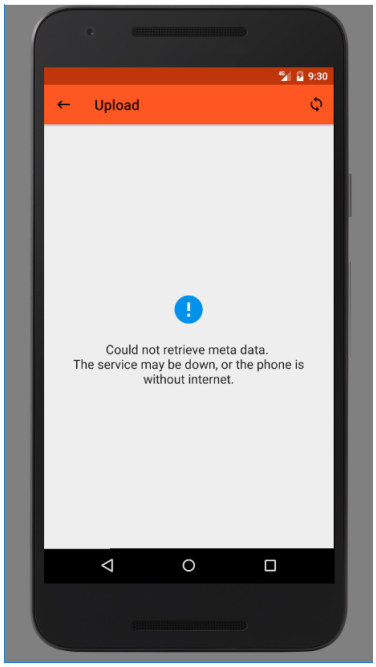
E.2 Use Case UCI2

Tested by: *Andreas Stensig Jensen*
Expected result: *The user has received confirmation that the data was uploaded.*
Test sequence:

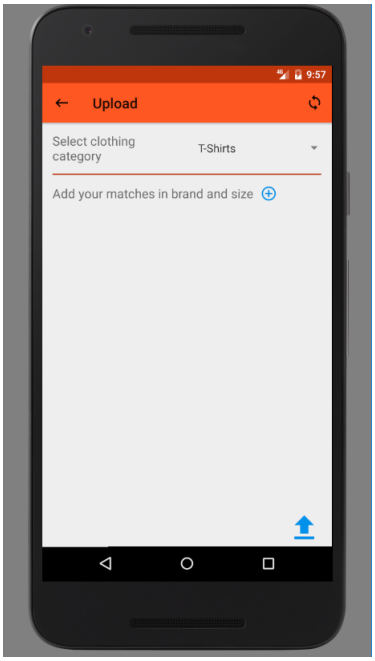
1. Open the android app CloseFit and press the uppload icon in the lower right. While waiting for the meta data a loading screen is shown.



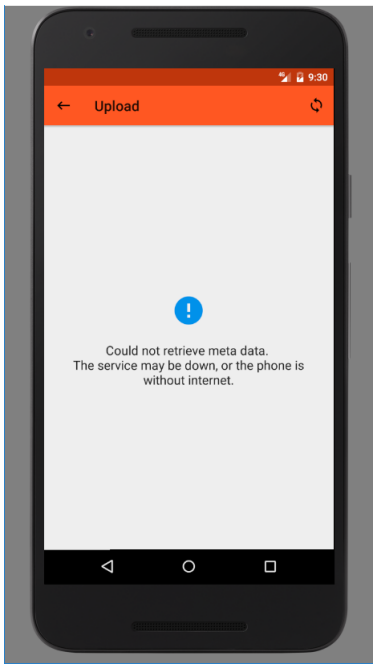
(a) If the app cannot establish a connection with the service system an error is shown.



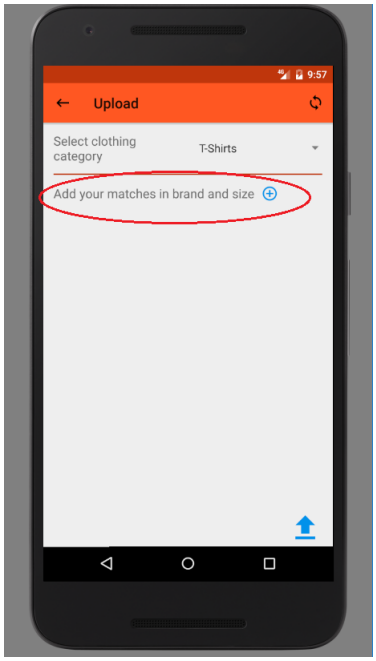
2. *See section E.3.*
3. Once the meta data has been loaded the app presents a view with the clothing categories for which data can be uploaded.



(a) If the app does not receive any meta data from the service system an error is shwon.

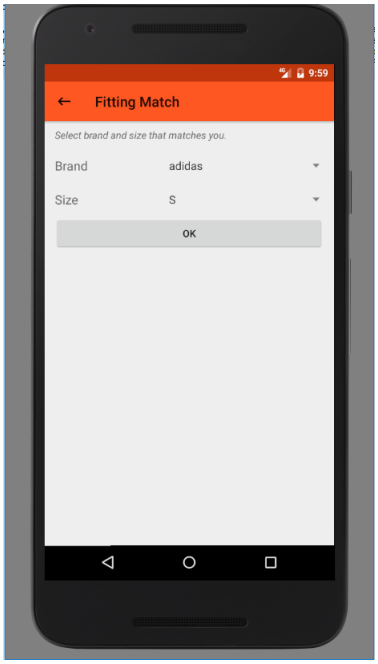


4. Select the clothing category to submit data for, in this case T-Shirts. Open view to add a matching brand and size by pressing the blue add-icon.

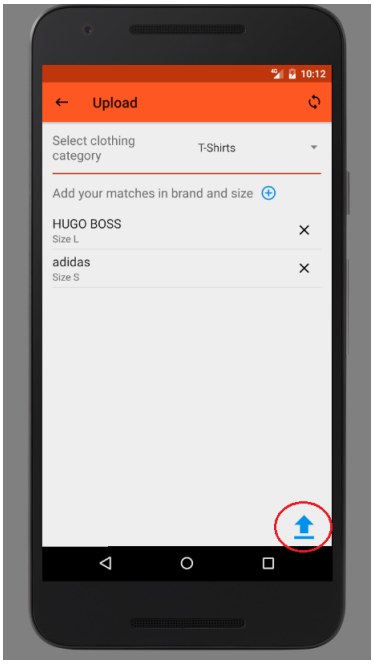


Select the brand and size that matches you. In this case adidas size S.

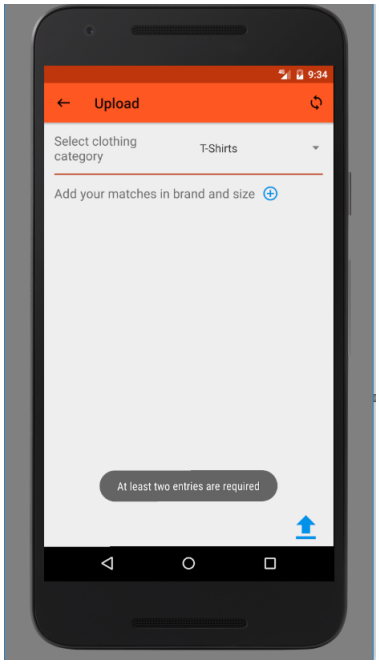
Press OK to store the choice.



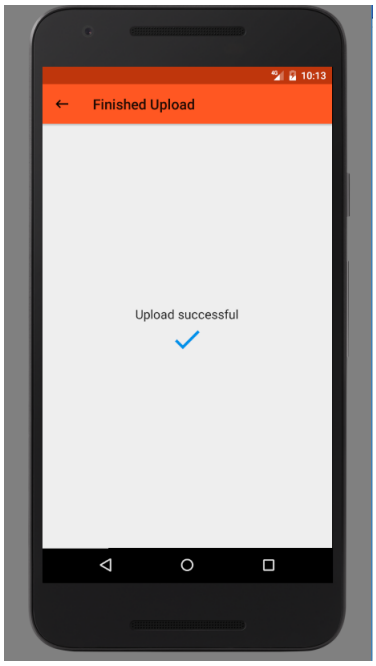
Add at least one more match, in this case HUGO BOSS size L, and start the upload by pressing the icon in the lower right.



5. A loading screen is shown while the app uploads the data to the service system. No actions required.
   (a) If no match data was added an error is shown.



6. *See section E.5.*
7. The app shows the result of the upload.



E.3 Use Case UCS1

Tested by: *Andreas Stensig Jensen*
Expected result: *The service system has returned the current clothing meta data.*
Test sequence:

- The web-service handling from the service system does not require any user interactions. A snippet of the system's log file is shown as documentation.

```
2017-03-06 00:26:43.9042|CloseFit.Application.WebAPI.Controllers.MetaController [INFO]  Meta request recieved.
2017-03-06 00:26:44.0912|CloseFit.Application.WebAPI.Controllers.MetaController [INFO]  Returning meta data: {
    Category: T-Shirts
    URIs: /api/tshirts/compare | /api/tshirts/upload
    Brand count: 4
    Size count: 4

    Category: Sneakers
    URIs: /api/sneakers/compare | /api/sneakers/upload
    Brand count: 3
    Size count: 3

}
```

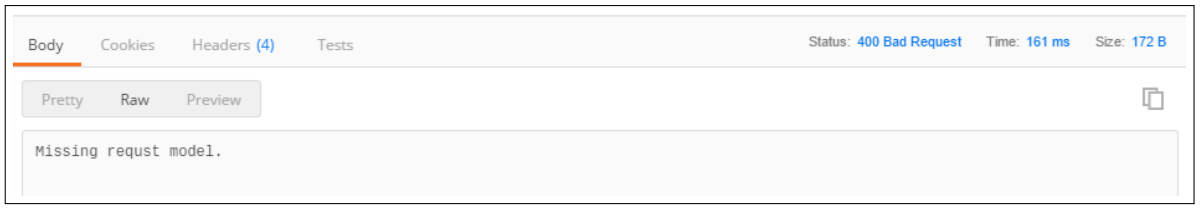
E.4 Use Case UCS2

Tested by: *Andreas Stensig Jensen*
Expected result: *The service system has returned the result of the comparison request.*
Test sequence:

- The web-service handling from the service system does not require any user interactions. A snippet of the system's log file is shown as documentation.

```
2017-03-06 00:27:03.5369|CloseFit.Application.WebAPI.Controllers.TshirtsController [INFO]  Received compare request {
    Result brand ID: 0
    Brand ID: 1, size ID: 2
}
2017-03-06 00:27:03.8705|CloseFit.Application.WebAPI.Controllers.TshirtsController [INFO]  Returning compare result: {
Mapped brand: adidas
    Size: L, count: 2
    Size: M, count: 1
}
```

- To test the alternate flow the app cannot be used. Instead a request must be made through a browser or program like Postman. Make a POST request to http://88.99.126.154/CloseFit/api/tshirts/compare.

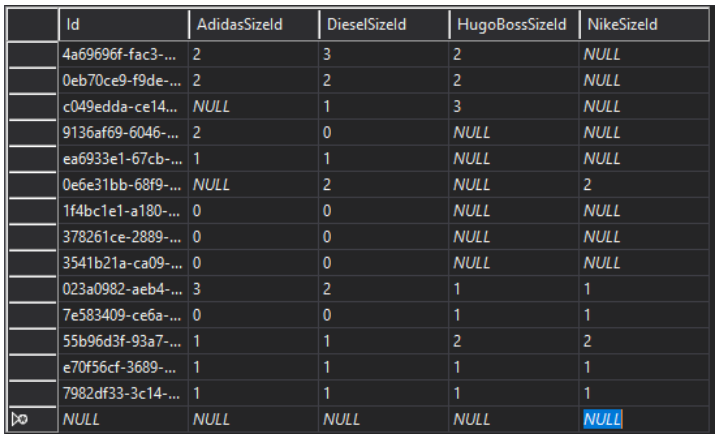


E.5 Use Case UCS3

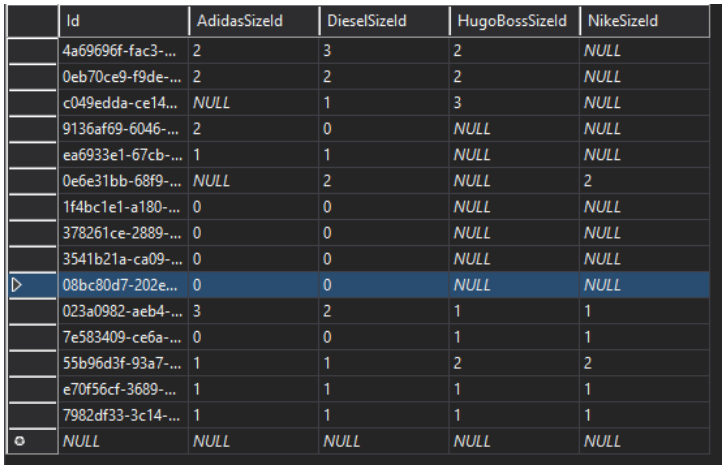
Tested by: *Andreas Stensig Jensen*
Expected result: *The service system has returned the result of the clothing data upload.*
Test sequence:

- The web-service handling from the service system does not require any user interactions. A snippet of the system's log file is shown as documentation, as well as a snapshot of the database before and after the upload with the new row highlighted.

```
2017-03-06 00:27:16.1152|CloseFit.Application.WebAPI.Controllers.TshirtsController [INFO]  Received clothing data: {
    SizeId: 0, brandId: 0
    SizeId: 0, brandId: 2
}
2017-03-06 00:27:16.3063|CloseFit.Application.WebAPI.Controllers.TshirtsController [INFO]  Data saved.
```

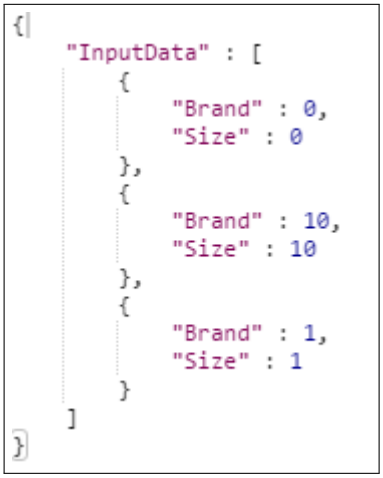


Id	AdidasSizeId	DieselSizeId	HugoBossSizeId	NikeSizeId
4a69696f-fac3-...	2	3	2	NULL
0eb70ce9-f9de-...	2	2	2	NULL
c049edda-ce14...	NULL	1	3	NULL
9136af69-6046-...	2	0	NULL	NULL
ea6933e1-67cb-...	1	1	NULL	NULL
0e6e31bb-68f9-...	NULL	2	NULL	2
1f4bc1e1-a180-...	0	0	NULL	NULL
378261ce-2889-...	0	0	NULL	NULL
3541b21a-ca09-...	0	0	NULL	NULL
023a0982-aeb4-...	3	2	1	1
7e583409-ce6a-...	0	0	1	1
55b96d3f-93a7-...	1	1	2	2
e70f56cf-3689-...	1	1	1	1
7982df33-3c14-...	1	1	1	1
NULL	NULL	NULL	NULL	NULL



Id	AdidasSizeId	DieselSizeId	HugoBossSizeId	NikeSizeId
4a69696f-fac3-...	2	3	2	NULL
0eb70ce9-f9de-...	2	2	2	NULL
c049edda-ce14...	NULL	1	3	NULL
9136af69-6046-...	2	0	NULL	NULL
ea6933e1-67cb-...	1	1	NULL	NULL
0e6e31bb-68f9-...	NULL	2	NULL	2
1f4bc1e1-a180-...	0	0	NULL	NULL
378261ce-2889-...	0	0	NULL	NULL
3541b21a-ca09-...	0	0	NULL	NULL
08bc80d7-202e...	0	0	NULL	NULL
023a0982-aeb4-...	3	2	1	1
7e583409-ce6a-...	0	0	1	1
55b96d3f-93a7-...	1	1	2	2
e70f56cf-3689-...	1	1	1	1
7982df33-3c14-...	1	1	1	1
NULL	NULL	NULL	NULL	NULL

1. (a) In order to test the alternative flow for 1a, make a POST request to http://88.99.126.154/CloseFit/api/tshirts/upload with the data shown below, either in a browser or a program like Postman. The data contains an invalid ID (10) for both brand and size, but as can be seen on the 2nd and 3rd image, a new relation is still added despite the input having an invalid mapping.



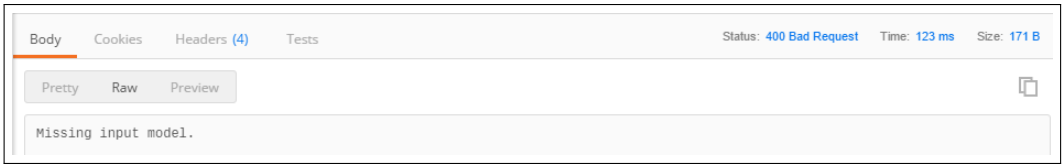
```
{
    "InputData" : [
        {
            "Brand" : 0,
            "Size" : 0
        },
        {
            "Brand" : 10,
            "Size" : 10
        },
        {
            "Brand" : 1,
            "Size" : 1
        }
    ]
}
```

	Id	AdidasSizeId	DieselSizeId	HugoBossSizeId	NikeSizeId
	4a69696f-fac3-...	2	3	2	NULL
	0eb70ce9-f9de-...	2	2	2	NULL
	c049edda-ce14...	NULL	1	3	NULL
	9136af69-6046-...	2	0	NULL	NULL
	ea6933e1-67cb-...	1	1	NULL	NULL
	0e6e31bb-68f9-...	NULL	2	NULL	2
	1f4bc1e1-a180-...	0	0	NULL	NULL
	378261ce-2889-...	0	0	NULL	NULL
	3541b21a-ca09-...	0	0	NULL	NULL
	08bc80d7-202e...	0	0	NULL	NULL
	023a0982-aeb4-...	3	2	1	1
	7e583409-ce6a-...	0	0	1	1
	55b96d3f-93a7-...	1	1	2	2
	e70f56cf-3689-...	1	1	1	1
	7982df33-3c14-...	1	1	1	1
▷	NULL	NULL	NULL	NULL	NULL

	Id	AdidasSizeId	DieselSizeId	HugoBossSizeId	NikeSizeId
	4a69696f-fac3-...	2	3	2	NULL
	0eb70ce9-f9de-...	2	2	2	NULL
	c049edda-ce14...	NULL	1	3	NULL
	9136af69-6046-...	2	0	NULL	NULL
	ea6933e1-67cb-...	1	1	NULL	NULL
	0e6e31bb-68f9-...	NULL	2	NULL	2
	1f4bc1e1-a180-...	0	0	NULL	NULL
	378261ce-2889-...	0	0	NULL	NULL
	3541b21a-ca09-...	0	0	NULL	NULL
	08bc80d7-202e...	0	0	NULL	NULL
▷	74fe29e3-298c-...	0	NULL	1	NULL
	023a0982-aeb4-...	3	2	1	1
	7e583409-ce6a-...	0	0	1	1
	55b96d3f-93a7-...	1	1	2	2
	e70f56cf-3689-...	1	1	1	1
	7982df33-3c14-...	1	1	1	1
	NULL	NULL	NULL	NULL	NULL

(b) In order to test the alternative flow for 1b, make an empty POST

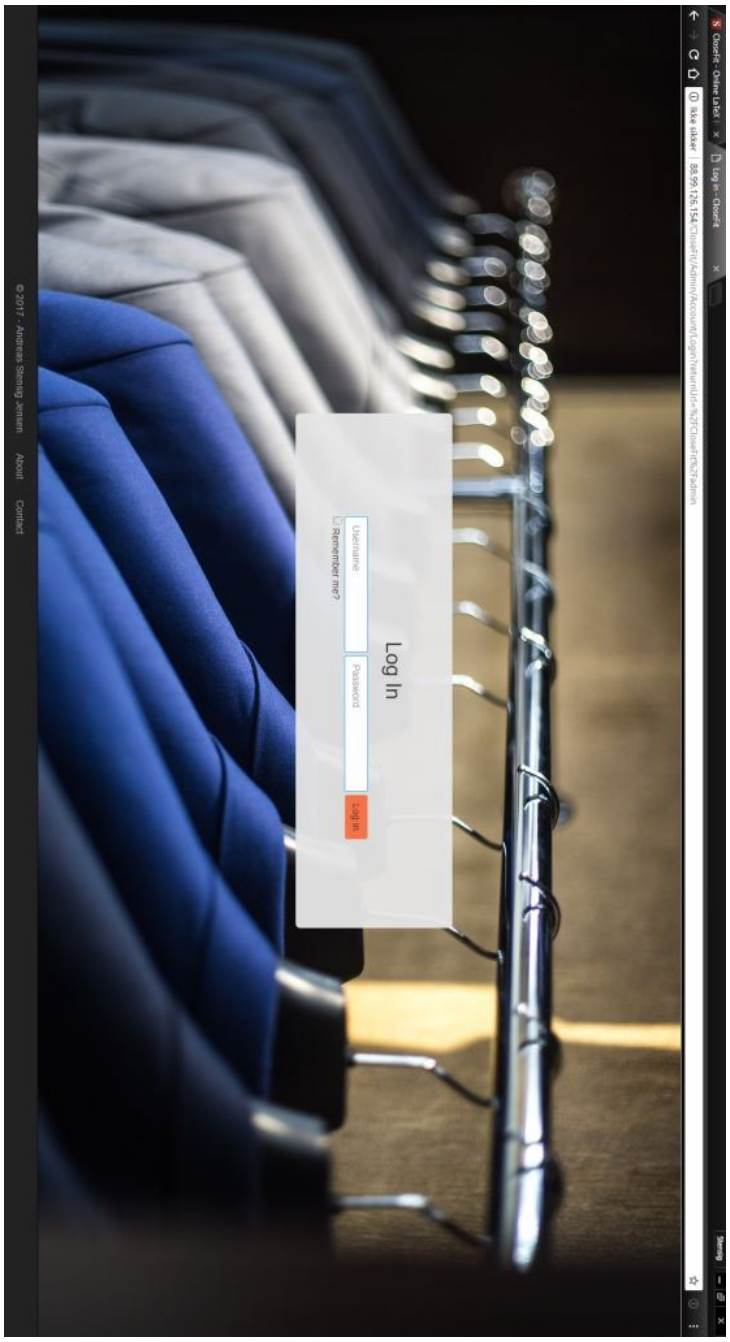
request to http://88.99.126.154/CloseFit/api/tshirts/upload, either in a browser or a program like Postman.



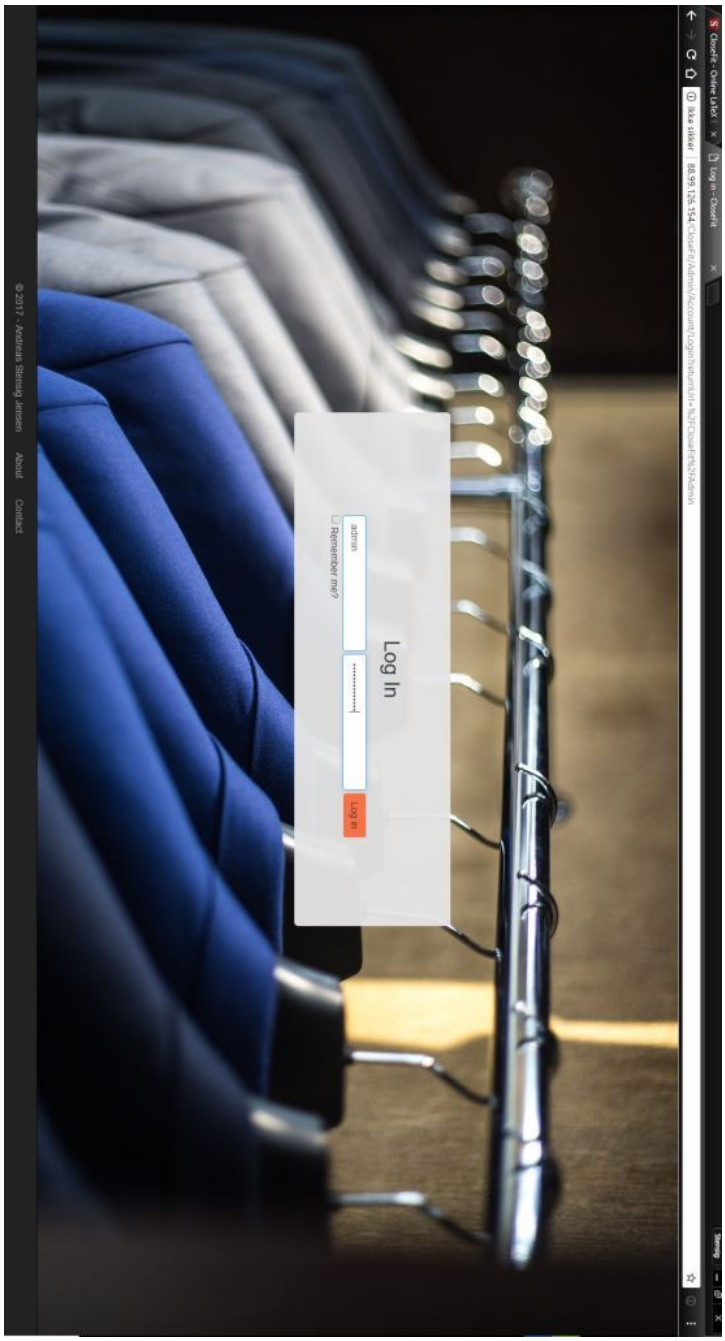
E.6 Use Case UCS4

Tested by: *Andreas Stensig Jensen*
Expected result: *The admin is logged in and is presented with CRUD operations.*
Test sequence:

1. Open the admin website from a browser at address http://88.99.126.154/CloseFit/admin.

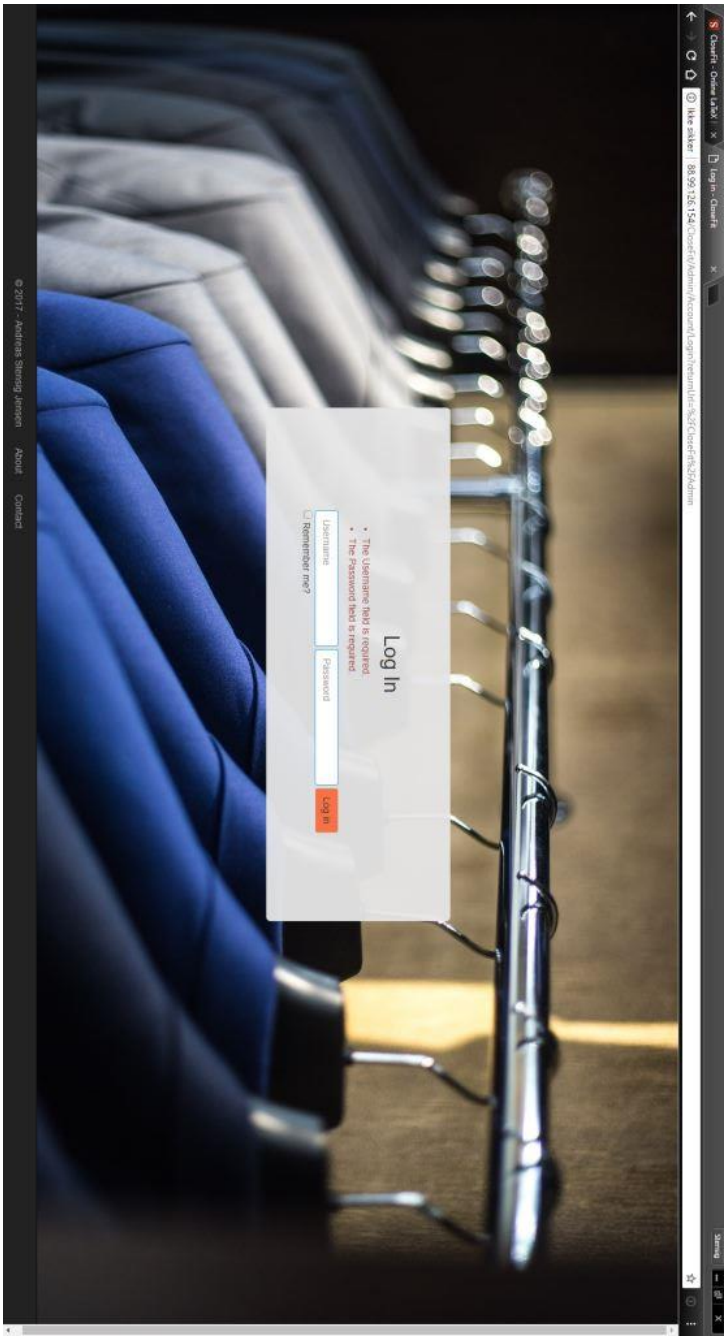


2. To log in, type in the admin credentials, username *admin* and password *EditCloseFit17*, and click 'Log in'.



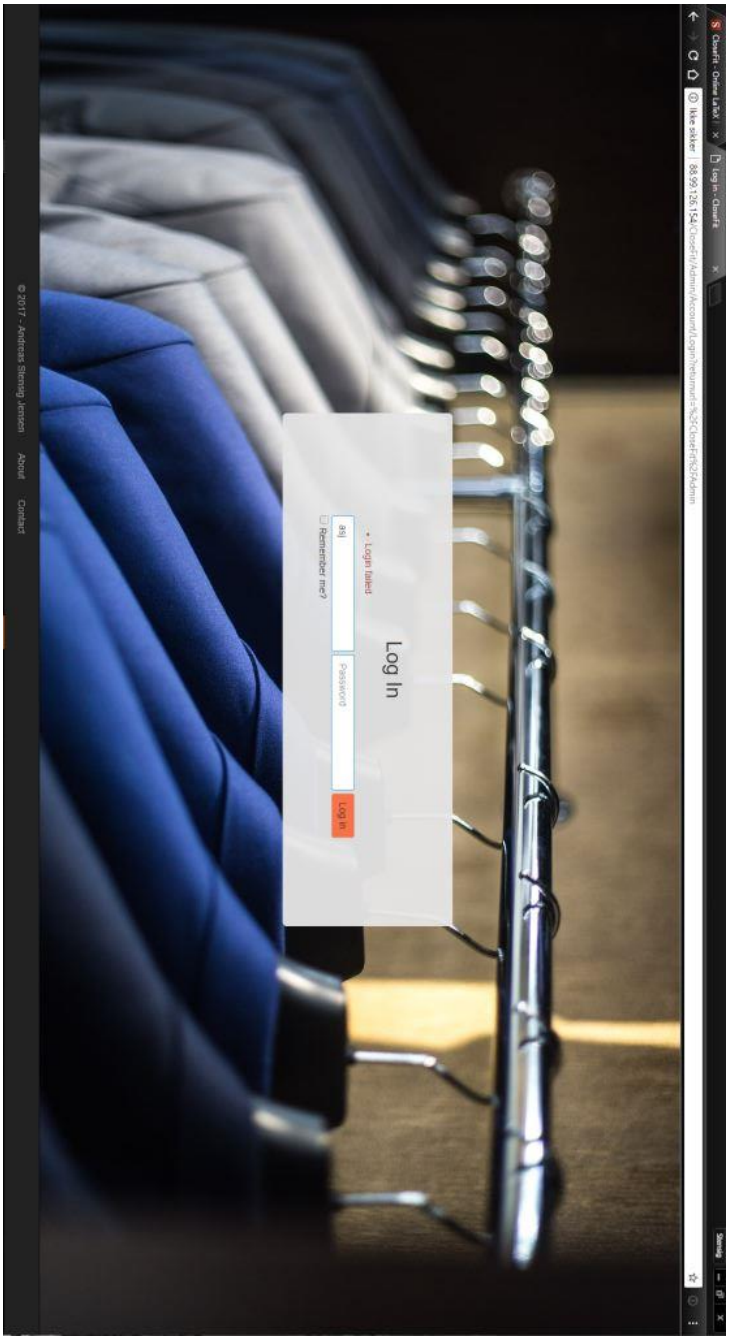
3. No action is required as the website authenticates log-in credentials.
   (a) If no credentials are entered when logging in, the website displays an

error.



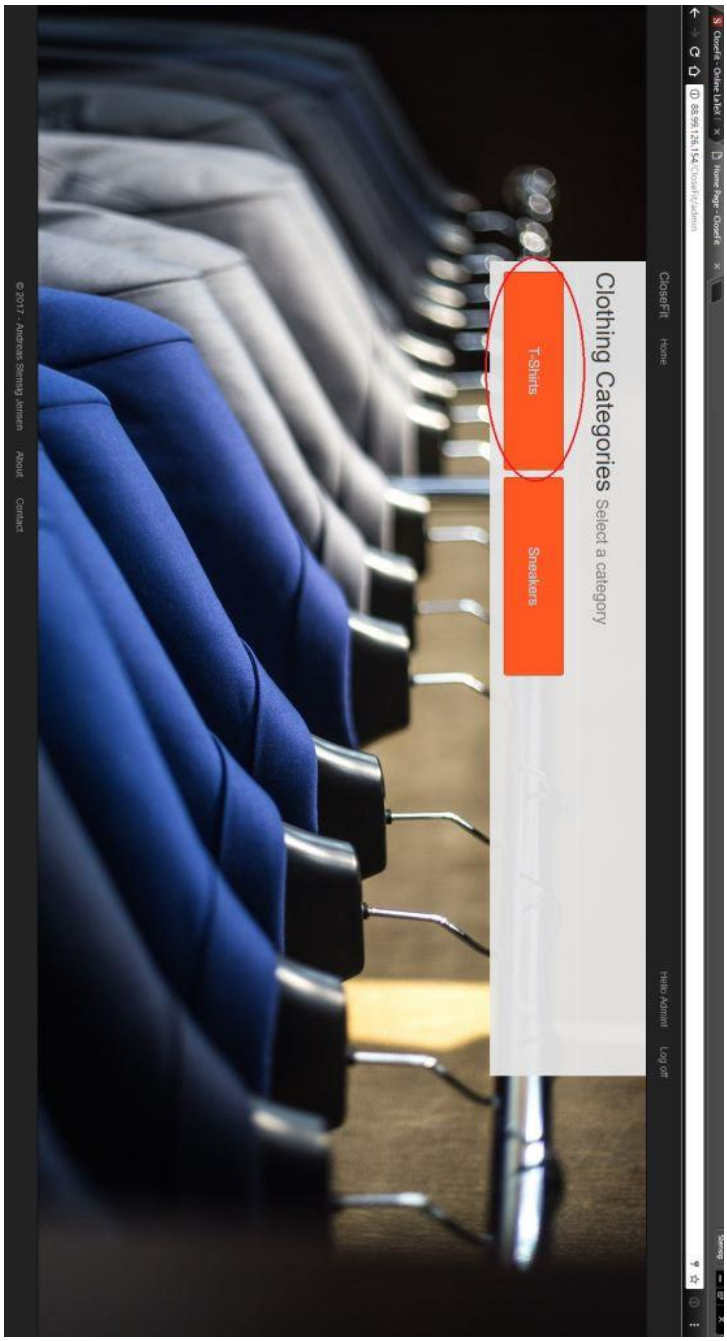
(b) If invalid credentials are entered when logging in, the website displays

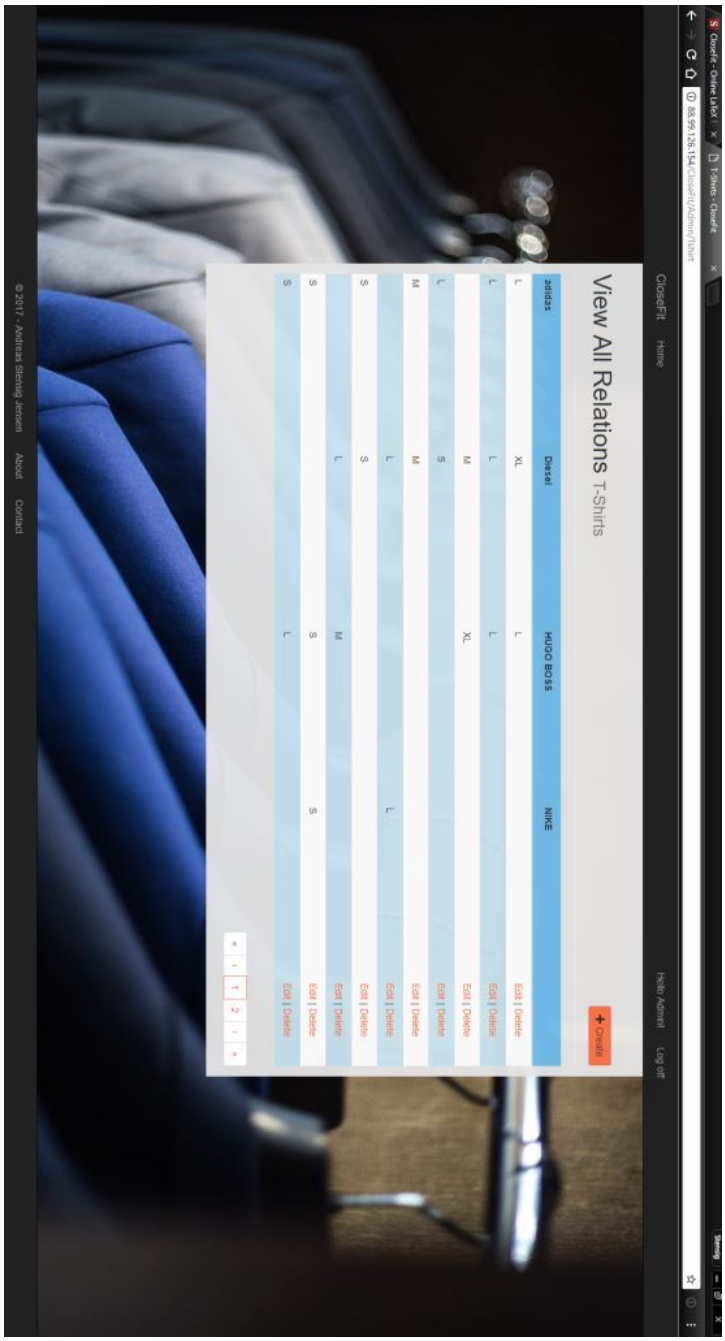
an error.



4. The website redirects you to an overview of the available clothing cate-

gories. Select T-Shirts by clicking on the button. This redirects you to the list-overview of the T-Shirts category, showing you all the stored relations with options to edit and delete them, as well as creating a new relation. The views of these further options are not shown here.


CloseFit
Home
Hello Admin!
Log off
Clothing Categories
Select a category
T-Shirts
Sneakers
© 2017 - Andreas Slemdy Jensen
About
Contact


CloseFit
Home
Hello Admin!
Log off
View All Relations T-Shirts
+ Create
adidas
Diesel
HUGO BOSS
NIKE
Edit | Delete
© 2017 - Andreas Stensig Jensen
About
Contact

F. Package Diagrams

This appendix contains the basic package diagrams of the project's product, separated into the app and the webapp for the service interface and service system respectively.

Package diagrams have been prioritised over class diagrams as the author did not feel that the detailed relationship of the many class diagrams would be beneficial to the overall understanding of the solution's internal composition.

The diagrams strive to comply with the UML standard. By default the classes with their visibility are listed inside of their containing package, unless the package contains sub-packages in which case a class is displayed by itself (in a grey box). The specific relationship between the individual packages are not further specified beyond the general "uses" relation. Such relations are not specified for the individual classes but the package in general for sake of simplicity.

It should also be noted that the diagrams only focus on Java classes for the app and C# classes for the webapp. Other resources are generally not represented.

F.1 Android app

The following are the package diagrams for the CloseFit Android app.

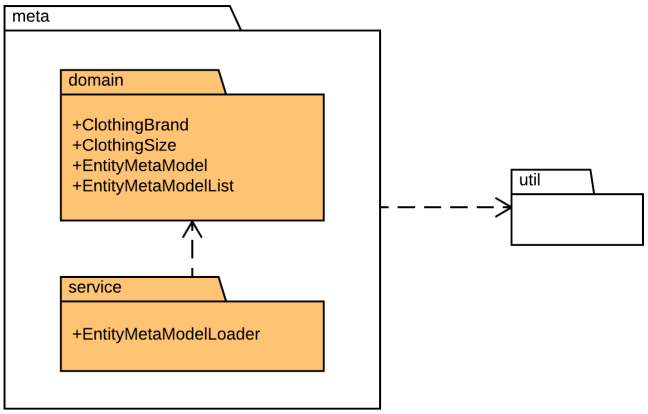


Figure F.1: dk.asj.closefit.meta package diagram.

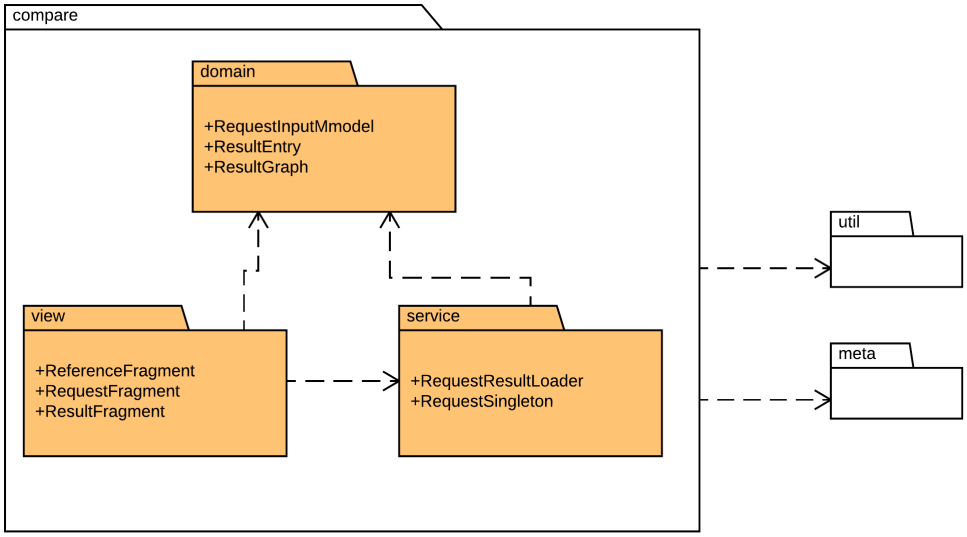


Figure F.2: dk.asj.closefit.compare package diagram.

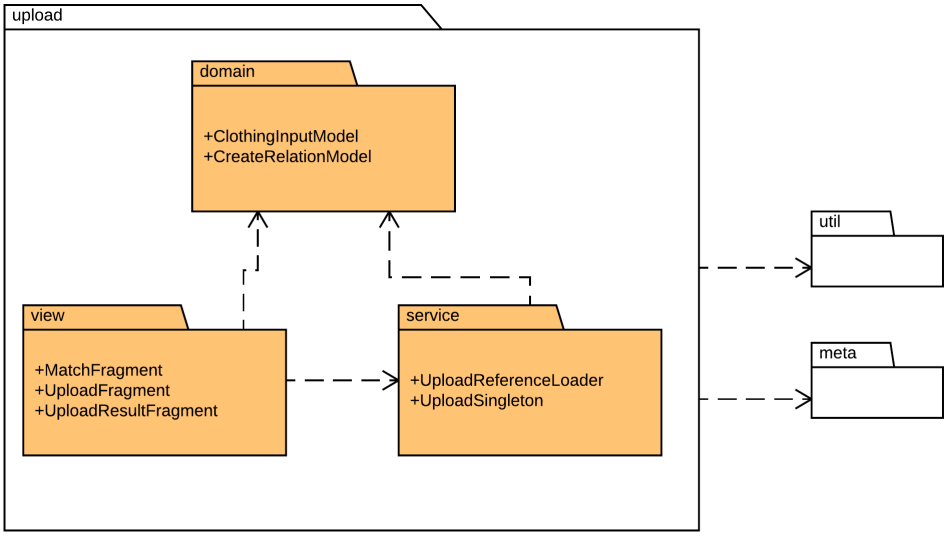


Figure F.3: dk.asj.closefit.upload package diagram.

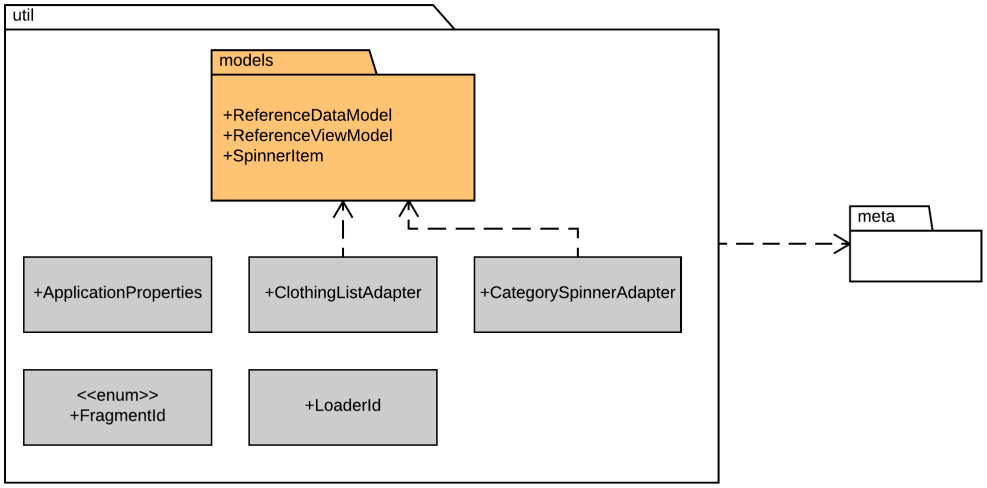


Figure F.4: dk.asj.closefit.util package diagram.

F.2 ASP.NET Core webapp

The following are the package diagrams for the CloseFit ASP.NET Core webapp.

F.2.1 Core

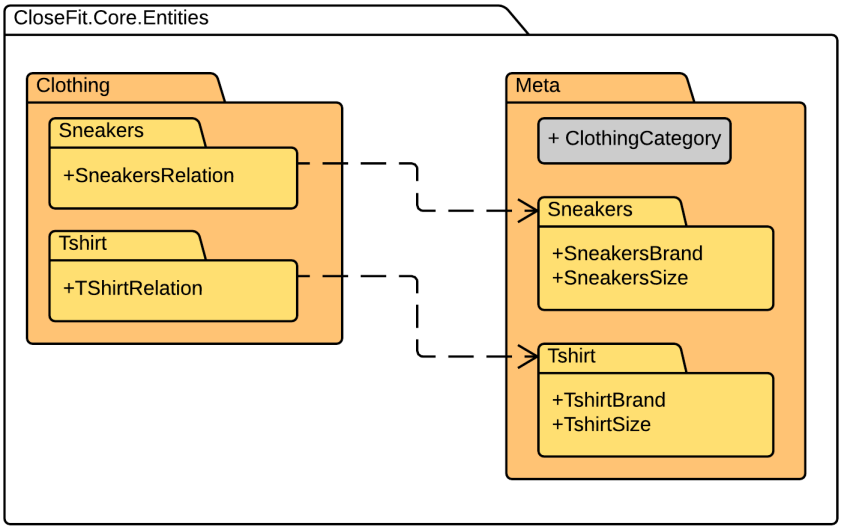


Figure F.5: CloseFit.Core.Entities package diagram.

F.2.2 Infrastructure

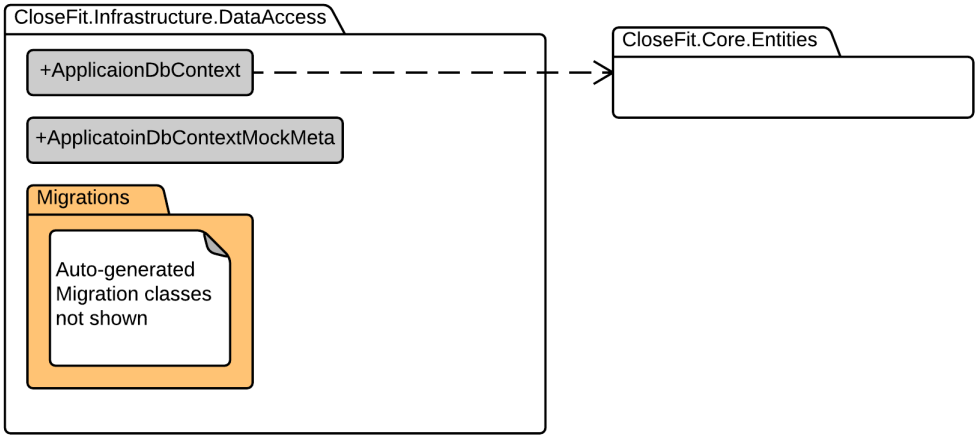


Figure F.6: CloseFit.Infrastructure.DataAccess package diagram.

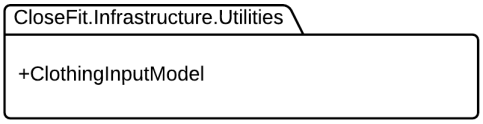


Figure F.7: CloseFit.Infrastructure.Utilities package diagram.

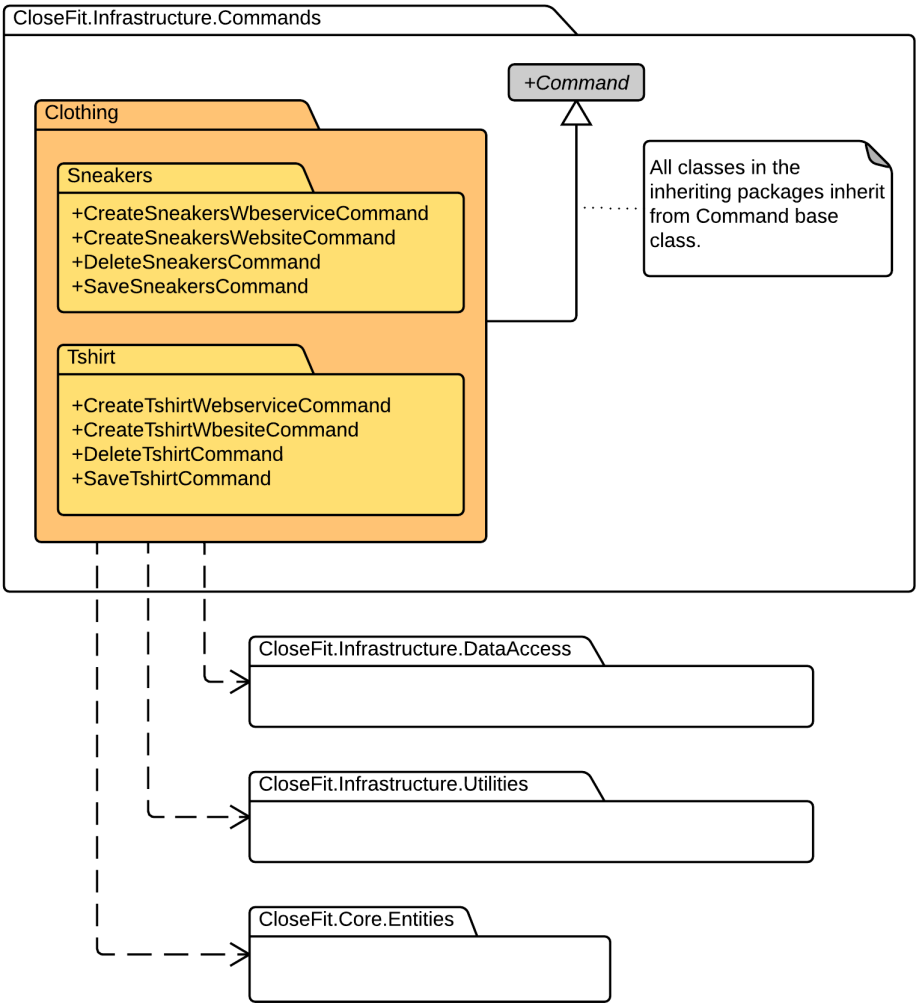


Figure F.8: CloseFit.Infrastructure.Commands package diagram.

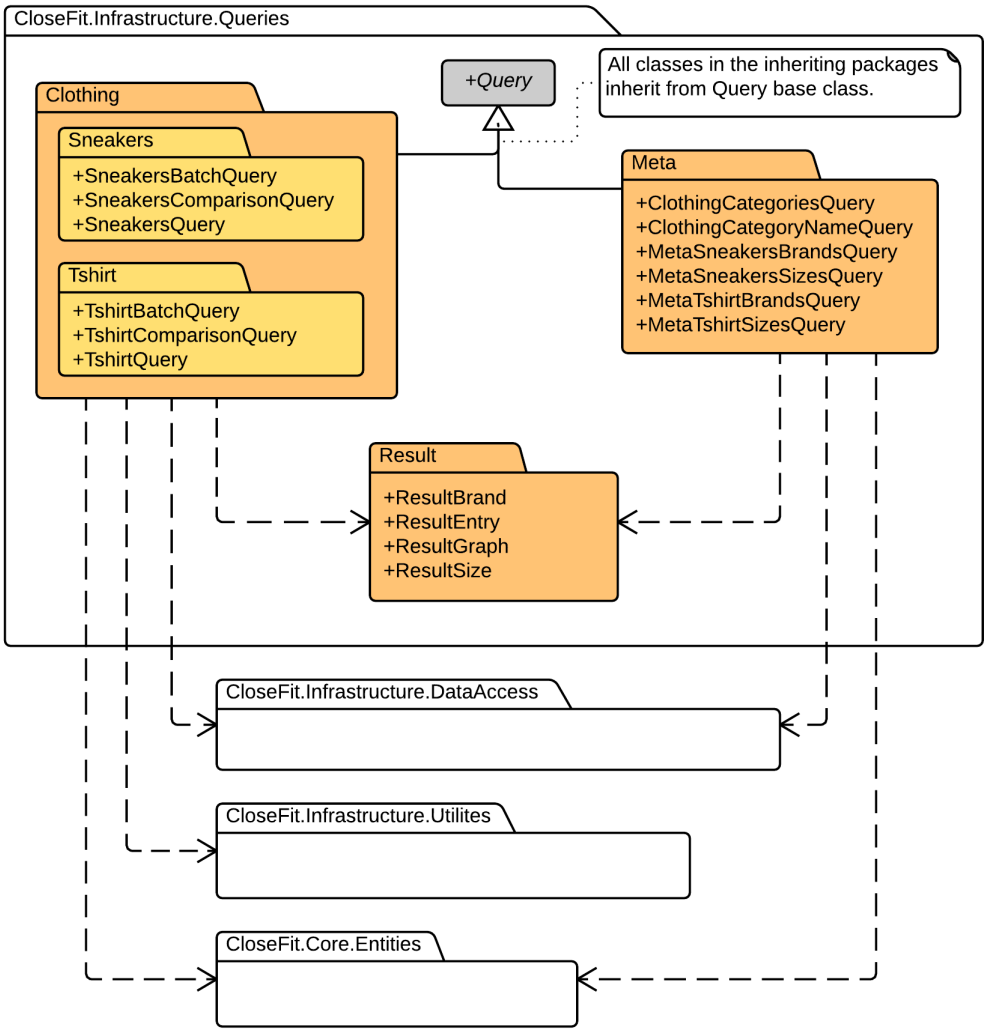


Figure F.9: CloseFit.Infrastructure.Queries package diagram.

F.2.3 Application

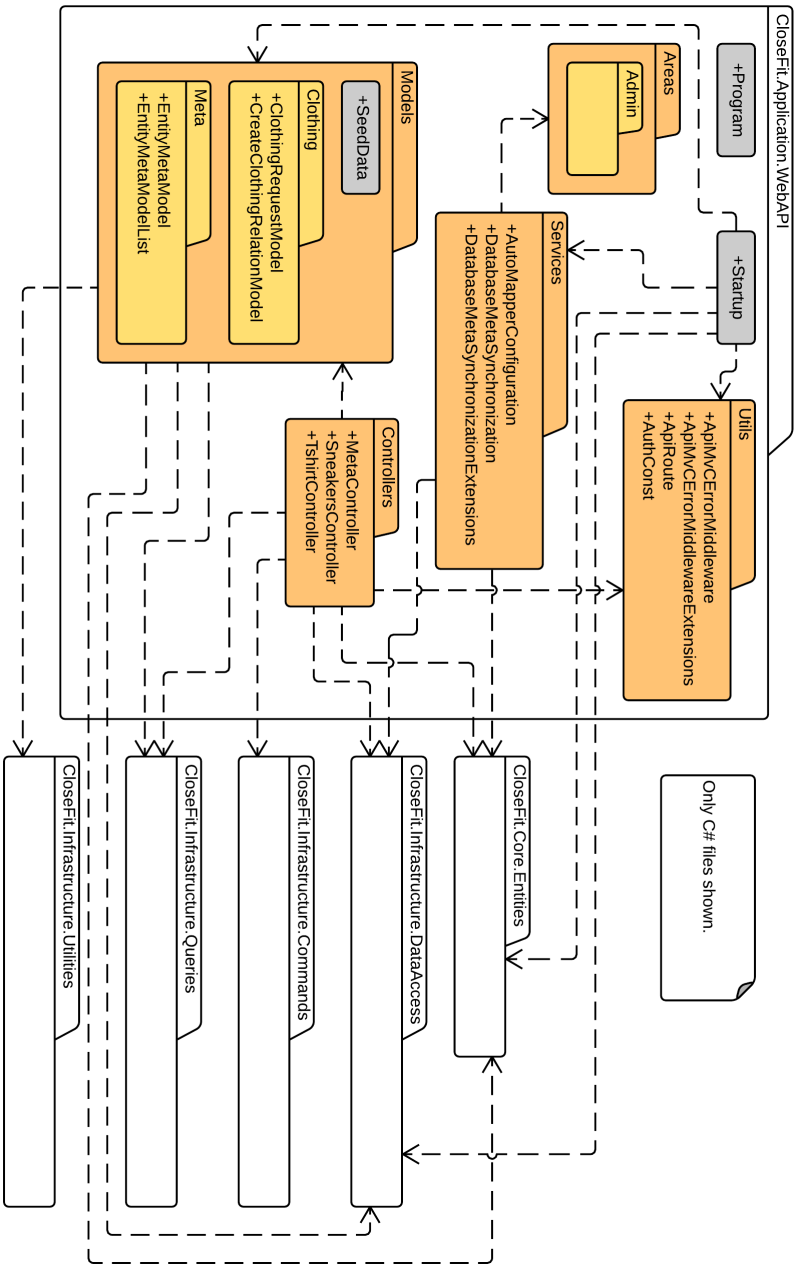


Figure F.10: CloseFit.Application.WebAPI package diagram.

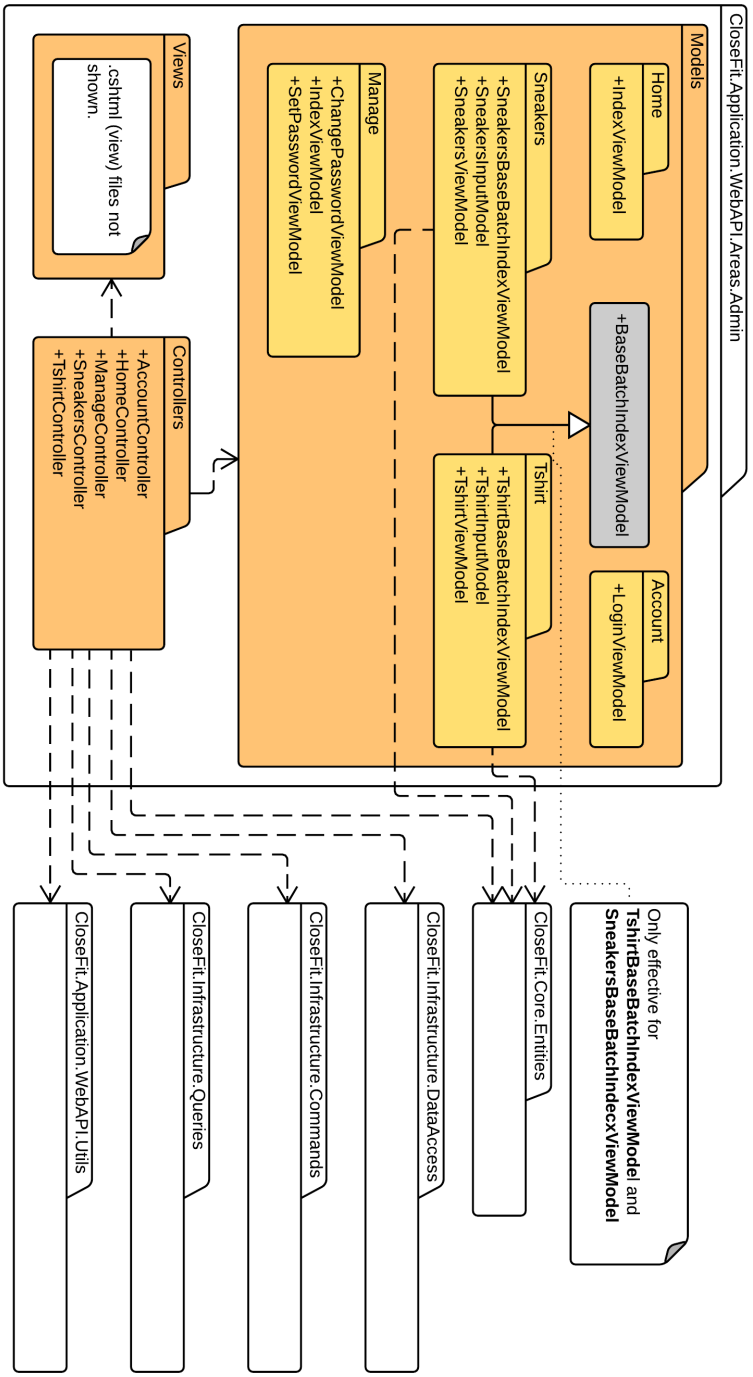


Figure F.11: CloseFit.Application.WebAPI (Areas/Admin folder) package diagram.

Bibliography

[And17a] Andrei. *Exception Handling Middleware*. Retrieved Feb. 2017. URL: http://stackoverflow.com/a/38935583/5935250.

[Fou17] .NET Foundation. *Getting Started with xUnit.net*. Retrieved Feb. 2017. URL: https://xunit.github.io/docs/getting-started-desktop.html.

[Geh17] Jonas Gehring. *GraphView - open source graph plotting library for Android*. Retrieved Feb. 2017. URL: http://www.android-graphview.org/.

[Mic17a] Microsoft. *Areas*. Retrieved Feb. 2017. URL: https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/areas.

[Mic17b] Microsoft. *ASP.NET Core Middleware Fundamentals*. Retrieved Feb. 2017. URL: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware.

[Mic17c] Microsoft. *Introduction to Identity*. Retrieved Feb. 2017. URL: https://docs.microsoft.com/en-us/aspnet/core/security/authentication/identity.

[Mic17d] Microsoft. *Role based Authorization*. Retrieved Feb. 2017. URL: https://docs.microsoft.com/en-us/aspnet/core/security/authorization/roles.

[Mic17e] Microsoft. *Routing in ASP.NET Core*. Retrieved Feb. 2017. URL: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/routing.

[Mic17f] Microsoft. *Testing with SQLite*. Retrieved Feb. 2017. URL: https://docs.microsoft.com/en-us/ef/core/miscellaneous/testing/sqlite.

[Pal17] Jeffrey Palermo. *The Onion Architecture*. Retrieved Feb. 2017. URL: http://jeffreypalermo.com/blog/the-onion-architecture-part-1/.

[Pix17] Pixabay. *Free High Quality Images*. Retrieved Feb. 2017. URL: https://pixabay.com/.

[Rod17] Joel Rodrigues. *NLog.Web: Getting started with ASP.NET Core*. Retrieved Feb. 2017. URL: https://github.com/NLog/NLog.Web/wiki/Getting-started-with-ASP.NET-Core-(project.json).

[Sof17] Pivotal Software. *Consuming a RESTful Web Service with Spring for Android*. Retrieved Feb. 2017. URL: https://spring.io/guides/gs/consuming-rest-android/.

[Ham17] Mosh Hamedani. *Repositories or Command / Query Objects?* Retrieved Jan. 2017. URL: http://programmingwithmosh.com/object-oriented-programming/repositories-or-command-query-objects/.

[And17b] Android. *Android Platform Verions*. Retrieved Mar. 2017. URL: https://developer.android.com/about/dashboards/index.html#Platform.

[Mic17g] Microsoft. *Introduction to using Gulp in ASP.NET Core*. Retrieved Mar. 2017. URL: https://docs.microsoft.com/en-us/aspnet/core/client-side/using-gulp.

Glossary

codebase The collection of the raw code that, when build, produces an application or software system. . 15, 30, 40, 44, 90

dependency injection A design method in which code modules that depend on the service of other modules do not maintain these modules themselves. Instead the module's dependencies are injected into it (often via a constructor argument), removing the creation and maintaining of the dependant modules to a centralised unit. . 17, 21

end user A person using the system. . 5, 8, 29, 46, 48, 50, 90

service interface A small sub-system that exposes the service API through a user friendly graphical interface (excluding the service website). Common name for both the prototype mobile app. . 8, 10, 13, 29, 30, 40, 46, 48, 50, 52–54, 80, 90

service system The online service and its backend. The outer boundary for this system is the web API and and admin website, not any interfaces exposing API. See service interface. . 8, 13, 15–17, 20, 24, 26–33, 37, 40, 45, 46, 48, 50, 52–55, 80, 90

service user An entity using the service API directly. This can be an end user manually using the web API or a service interface. . 46, 52–54

service website The website integrated into the service system codebase and which exposes graphical admin features. . 55

the author The author of the report, the student performing the project (Andreas Stensig Jensen, s134408) . 6, 7, 15, 16, 29, 32, 36, 80

the stakeholder The stakeholder for the project, also the external consultant; Mathias Harboe of IT Minds ApS. . 6–8, 10

Acronyms

CMS Content Management System. 4, 8, 10

CRUD Create, Read, Update, Delete. 25, 55, 72

DLL Dynamic-link library. 13, 16, 21, 24

MVC Model-View-Controller. 15

OOP Object-oriented programming. 13

ORM Object-relational mapping. 13, 16, 17, 20, 37

RTM Requirements traceability matrix. 9

SRS System Requirements Specification. 7, 44, 46, 56