Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Flutter on Windows Desktop: a Use Case Based Study

Stefan Zindl

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Marco Aiello
Supervisor:	Prof. Dr. Marco Aiello
Commenced:	February 1, 2021
Completed:	August 2, 2021

Abstract

In the last years, the number of different computer platforms increased from Desktop, mobile devices, tablets to the Web. Among others, cross-platform frameworks enable to target all platforms. One of those cross-platform frameworks is Flutter which is developed by Google and targets Windows Desktop in beta stage since 2020. Because of this early stage, it is relevant to verify how well Flutter already works on Windows Desktop. In the first part of this bachelor thesis, we compare a simple image gallery in Flutter and WPF with .NET 5. The implementation in both frameworks worked well with similar kind of realization. Our comparison concentrates on metrics such as code, startup time, and packaged size. In addition, we measure RAM usage and CPU usage. We measure these in two scenarios which we automated with a simulation script. In the second part, we focus on the available third-party extensions and the current missing functionalities of the Flutter framework. Our results indicate that we could implement the Flutter application with 55% less code and with a 70 times faster startup time. Surprisingly, Flutter uses less RAM most of the time, but instead, it needs more CPU to process the images. Nevertheless, there are some missing important functionalities for Desktop applications such as adding icons in the system tray or adding a menubar to the application. We show that some functionality is still missing in the current stage of the Flutter framework, but it has a good chance to become a well established framework for new developers.

Keywords: Desktop, WPF, Windows, Cross-Platform, Flutter, Use-Case Study

Kurzfassung

In den letzten Jahren entstanden viele neue Computer-Platfformen von Desktop, Mobiltelefonen, Tablets und das Web. Vorallem plattformübergreifende Frameworks haben dazu beigetragen, alle Plattformen zu unterstützen. Dazu zählt das neuartige Framework Flutter, welches von Google entwickelt und seit 2020 auch Windows Desktop in Beta Version unterstützt. Daher ist es relevant zu überprüfen, wie weit die Implementierung fortgeschritten ist, wofür wir es mit dem Desktop-Framework WPF vergleichen, indem wir eine einfache Bildergalerie in Flutter und WPF mit .NET 5 implementieren. Die Umsetzung in beiden Frameworks konnten wir mit kleinen Unterschieden gut bewerkstelligen. In unserer Bachelorarbeit konzentrieren wir uns im Rahmen des Vergleiches auf die Anzahl der Codezeilen, Startzeit der Applikation und die Größe des Installationspakets. Zwei weitere Metriken, RAM- und CPU-Verbrauch, messen wir in zwei Szenarien, welche wir vollständig automatisiert haben. Im zweiten Teil betrachteten wir auf das Framework genauer, welche zusätzlichen Erweiterungen es für Flutter bereits gibt und welche Funktionalitäten noch fehlen. Als Ergebnis fanden wir heraus, dass unsere Flutter-Anwendung mit 55% weniger Codezeilen implementiert werden konnte und die Applikation 70 Mal schneller startet. Überraschend war auch, dass Flutter während den Szenarien weniger RAM, stattdessen aber mehr CPU zum Verarbeiten der Bilder benötigt. In Bezug auf das Framework selbst haben wir festgestellt, dass es bereits viele Drittanbieter-Abhängigkeiten für Flutter auf Windows gibt, was es für Entwickler attraktiv macht. Nichtdestotrotz fehlen einige bekannte Funktionalitäten, wie Tray-Icons oder eine Menübar. Fehlende Funktionalitäten haben wir bei dem jetzigen Stand des Frameworks zu erwarteten, dennoch hat Flutter eine gute Chance ein etabliertes Framework für Neueinsteiger bei Entwicklern zu werden.

Schlüsselwörter: Desktop, WPF, Windows, plattformübergreifend, Flutter, Fallstudie

Contents

List of Figures

List of Tables

Acronyms

CPU Computer Processing Unit. 3, 4, 13

MVVM Model-view-viewmodel. 29

RAM Random Access Memory. 3, 4, 13

UI User Interface. 13

UWP Universal Windows Platform. 22

WPF Windows Presentation Foundation. 3, 4, 5, 7, 9, 13

XAML Extensible Application Markup Language. 15

1 Introduction

Flutter is an open source cross-platform User Interface (UI) tool-kit. It has one code base and can target mobile devices, Web and even Desktop. For this tool-kit there is some research with Flutter on mobile devices and Web development technologies such as React Native and Electron. Flutter on Windows is currently in beta stage which is a reason that there has not been much research about Flutter targeting Windows Desktop. Nevertheless, Desktop applications are still developed for business, professional, or performance-critical areas. Therefore we compare Flutter with the Desktop application framework Windows Presentation Foundation (WPF) with .NET 5 which we call WPF in this thesis. This bachelor thesis fills the research gap evaluating Flutter on Windows Desktop. To set our thesis in context, we limit our research by answering the following research questions:

Research Question 1:
Is Flutter an alternative compared to Desktop application framework WPF with .NET 5?
Research Question 2:
What are the current limitations of Flutter?

To answer the first research question, we need to define the area we work on. To get there, we develop a simple image gallery application in both frameworks. With each framework, we implement two use cases: importing images and displaying an image in full size. With a simulation script, we measure the Computer Processing Unit (CPU) and Random Access Memory (RAM) usage during both scenarios. In addition, we compare the packaged size, startup time, and code size.

For answering the second research question, we research missing features and packages for developing a Windows Desktop application. After that, we focus on some features which are missing in the framework itself.

Our thesis is structured as the following. In chapter 2, we focus on the history of the Desktop application for WPF, the .NET, and Flutter. In chapter 3, we have a look at related work to those frameworks. In chapter 4, we first describe our features and secondly, we describe the most significant parts which came up during our development process. In chapter 5, we describe how we set up and execute our experiment to get the values for measuring the metrics CPU usage and RAM usage, code and application size, and startup time. Chapter 6 shows the results of our experiment and interprets those to find possible answers for our first research question. Chapter 7 answers the second research question to find limitations of Flutter. In the last chapter, we give a brief conclusion and an outlook of Flutter and which further work can be done with this thesis.

2 Background Information

Before we can talk about our reference applications, we take a brief look into their technologies. The reference applications are developed in WPF with .NET 5 and Flutter. Therefore, we describe its architectures first and then we show programming examples in both frameworks. In section 2.3, we classify these frameworks into the broad available frameworks and on the history of the .NET ecosystem. Finally, we also show some similar technologies which can be used for developing Desktop applications.

2.1 WPF

WPF is one of the UI tool-kits for Windows and was developed by Microsoft and published in 2006 (Georg, 2018). It provides many features for developers to create the UI for applications. The features include layout, resources, controls, and other elements. WPF uses the declarative based XML language Extensible Application Markup Language (XAML) to create the layout of the application, which we focus on in subsection 2.1.2. C# is most used to develop the logic behind the UI. The latest version of WPF uses .NET 5 (see subsection 2.1.3) which provides the developer a broad variety of libraries such as I/O, security libraries, and data-access. In addition, .NET provides the NuGet package manager (Microsoft, 2021b), which allows adding more dependencies to the application.

2.1.1 Architecture

The main feature of our application is to display images onto the screen. For this reason, we take a look into the architecture of WPF, which is illustrated in the following Figure 2.1. The third and innermost layer is the operating system itself. It provides all device drivers, and basic functionality such as window manager and displaying graphics. Additionally, it provides other low level functionalities.

Common Language Runtime (CLR) is the runtime for the application. It manages and compiles the code before executing the general machine code (Warren et al., 2016). CLR is a part of .NET 5 (see subsection 2.1.3) framework.

The Unmanaged Code layer is divided into two sections. "[..] Media Integration Layer (MIL) and resides in milCore.dll. The purpose of the milCore is to interface directly with DirectX and provide basic support for 2D and 3D surface. This section is unmanaged code because it acts as a bridge between WPF managed and the DirectX/User32 unmanaged API" (Sushil, 2015). The second section contains WindowsCodecs, which is an other low-level API for processing decode/encode images into vector graphics and display them on the screen (Sushil, 2015).

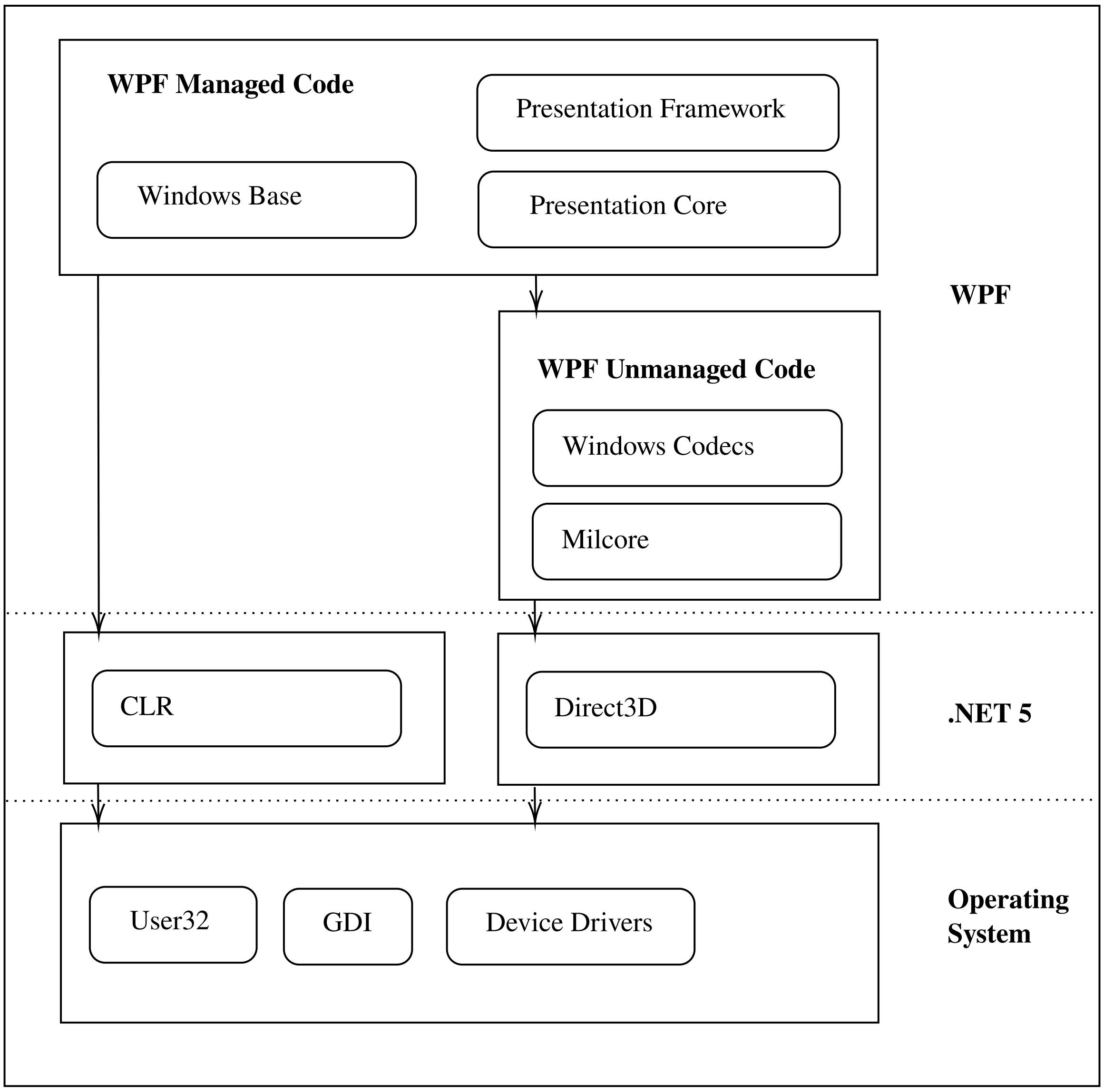


Figure 2.1: Overview of WPF architecture (Sushil, 2015)

The top layer is the WPF Managed Code which includes the Presentation Framework that provides developers the ability to create the layout with panels and content of the control. An example of a layout written in XAML is shown in Listing 2.1.

2.1.2 XAML

XAML is a declarative markup language for creating and developing a UI for the WPF. It enables the developer to create visible UI elements with the simple syntax in XML. It also includes layout elements such as panels, grid, and others (George, 2021).

```
<Window x:Class="WpfApplication1.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Image Gallery" Height="300" Width="300">
    <Stackpanel Orientation="Horizontal">
      <TextBlock Text="Good Morning" />
      <TextBlock Text="Hello World" />
    </Stackpanel>
</Window>
```

Listing 2.1: WPF example to create a window

Listing 2.1 shows an example of two text boxes inside a window with the specified attributes width and height which are wrapped inside a stack panel vertically orientated to place them above each other. XAML supports not only specifying the elements in a vertical and horizontal way, but also in depth, which means the further down elements are listed in code the more it shows in the front, which enables to overlap elements.

2.1.3 .NET 5

.NET 5 is an open source cross-platform Application Programming Interface (API) framework developed and published firstly in 2020 on GitHub by Microsoft (Microsoft, 2021a; Milanović, 2020). The goal of .NET 5 is to combine all prior .NET implementations among others such as .NET Core, .NET Framework, Xamarin with Mono, and to run on multiple devices. This also makes it possible to extend .NET 5 much easier to more platforms.

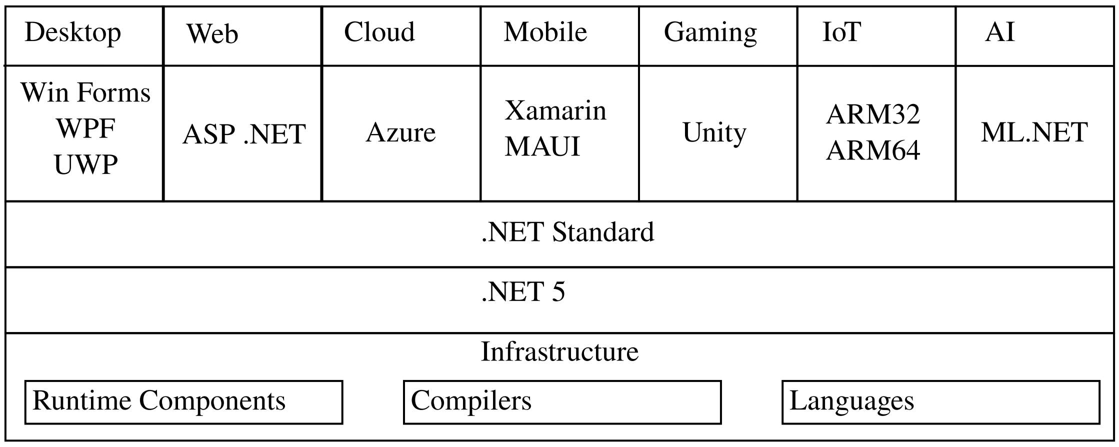


Figure 2.2: .NET - A unified platform (Richard, 2019)

Figure 2.2 shows the architecture and an overview of the frameworks that support .NET 5. You can see that .NET 5 supports the most important areas. From Desktop over mobile to Internet of Things (IoT). The low-level layer provides the runtime in which the compiled UI runs. It also provides the latest language feature of C# 9 (Milanović, 2020). The minimal version of Visual Studio for developing in .NET 5 is 16.8 (Lander, 2020).

```
<Project Sdk="Microsoft.NET.Sdk">
 <PropertyGroup>
   <OutputType>WinExe</OutputType>
```

```
    <TargetFramework>net5.0-windows10.0.19041.0</TargetFramework>
    <TargetPlatformVersion>7.0</TargetPlatformVersion>
    <UseWPF>true</UseWPF>
  </PropertyGroup>
</Project>
```

Listing 2.2: WPF .Net 5 Project .csproj confugration

After we create a new project, we first need to update the project configuration, which includes the project file .csproj. Listing 2.2 gives an example of how a configuration looks like. For a platform specific compilation for Windows we set the attribute `TargetFramework` to `net5.0-windows10.0.19041.0`.

2.2 Flutter

Flutter is an open source cross-platform UI tool-kit developed and published on Github by Google in 2017. The current version of Flutter is 2.2.0 which was released in May 2021 (Flutter, 2021c) . The goal of Flutter is to provide one single code base to target different platforms. There is already a stable version of Flutter for Android, iOS, Web, and Linux. Flutter on Windows it is on its beta stage. Flutter has received attention exponentially in the last months. 39% of all cross-platform developers already choose Flutter over other cross-platform frameworks. In 2019 it has risen by 9%. React Native is the most used cross-platform framework with a market share of 42% (Baron, 2021).

Flutter is still a growing and evolving framework. It has started with Mobile development, Web and now there are also projects using Flutter for embedded systems. Toyota started using Flutter for the UI in their automotive (Dilan, 2021). Not only Google uses Flutter, but also it is popular for mobile applications like Google Assistant, Google Ads, and from well-known companies like eBay, BMW, and Alibaba. Among those companies, also Cononical, the maker of Ubuntu, has started to replace its installer, which was written in Python (Sneddon, 2021). Flutter is not used only for applications on existing operating systems, but also Google develops the new operating system Fuchsia which uses Flutter for the UI (Google, 2021b).

Compared to WPF 2.1 which uses XAML for the UI and C# for the backend, Flutter uses the programming language for both the UI elements and its backend section 2.2.

2.2.1 Architecture

The Flutter Framework is built mainly in three layers, which are shown in Figure 2.3. For developers, the first layer is the most important one, which is the API to access features for the UI development. The API includes widgets, rendering, animation, painting gestures, and other high level functionalities (Flutter, 2021d).

The second layer includes the of the portable runtime which hosts the Flutter applications. It also includes core libraries for features such as animations, graphics, plugin architecture and toolchains (GitHub, 2021b).

The toolchains are mostly written in C and C++ and enable creating an executable for the specified platform. (Flutter, 2021b). Sells describes the toolchain for Flutter running on Windows: It creates a Shell which uses mostly Skia to render at native speed to an underlying DirectX surface. Additionally it creates a runner which loads and executes the Flutter application. The runner is written in `C` and `C++` and is added to the project. The developer has the ability to add native code to it (Sells, 2020).

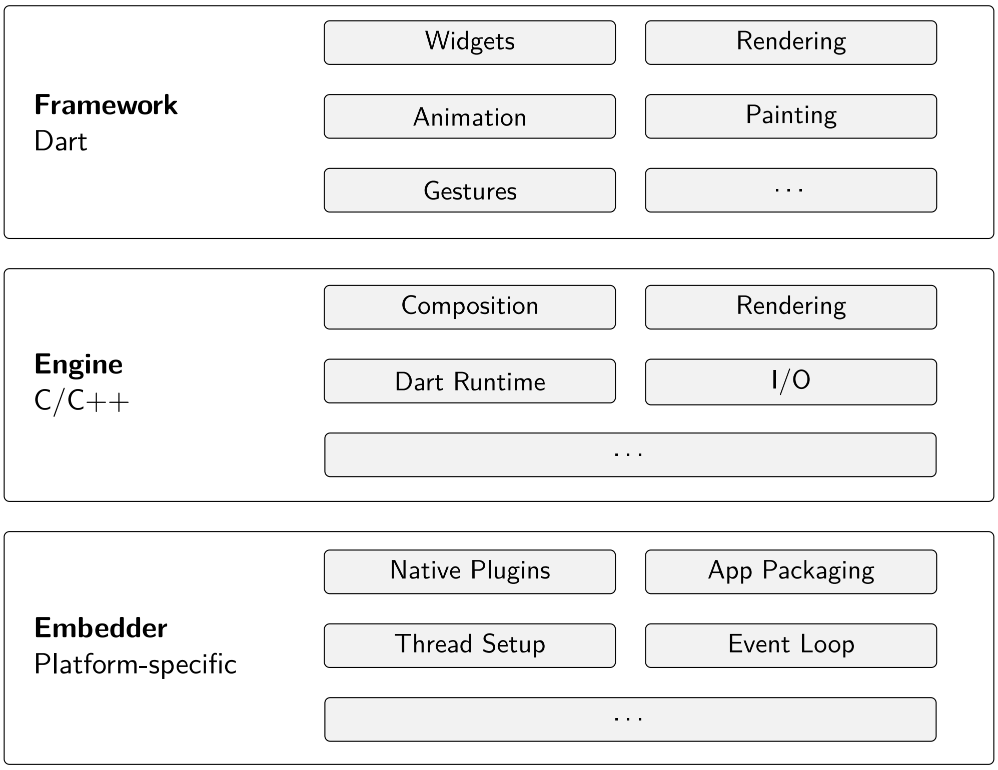


Figure 2.3: Main layers of Flutter (Flutter, 2021b)

2.2.2 Development

The Flutter UI is developed also with a declarative UI programming style like XAML, which means that we describe the state we want to draw on the screen. The component based approach also enables reusing widgets multiple times in the project (flutter, 2021).

```
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Conuter Sample',
      home: MyHomePage(),
    );
  }
}

class MyHomePage extends StatefulWidget {
  @override
  _MyHomePageState createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {
  int _counter = 0;

  void _incrementCounter() {
    setState(() {
      _counter++;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(widget.title),
      ),
      body: Center(
        child: Column(
          children: <Widget>[
            Text(
              'You have pushed the button this many times:',
            ),
            Text(
              '$_counter',
            ),
          ],
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: _incrementCounter,
        child: Icon(Icons.add),
      ),
    );
  }
}
```

Listing 2.3: Basic example in Flutter

A new Flutter project can be created with the command `flutter create <project name>`. The new project includes sample code, which the above code (see Listing 2.3) is a part of. All elements are seen as widgets, which are splitted in two main categories `StatelessWidget` and `StatefulWidget`. `StatelessWidgets` are used to display a widget, which can not be modified during runtime.The code sample in Listing 2.3 includes the `StatelessWidget Center`, which is used to center its child. The `Text`-widget instead is a `StatefulWidget`, which can be rebuilt in a new state using a new value. In our example, we use the method `setState` for the state management to update the variable `_counter`. We trigger this method by using the FloatingActionButton.

In contrast to XAMLs visual elements, layouts and even the `Padding` are widgets. The `Column` widget is used to display its children in a vertical order and the widget `Padding` is used to define the padding of another widget. In the application, all widgets together are ordered as a tree structure. For example Figure 2.4 shows our above code sample in Figure 2.3 in a widget tree.

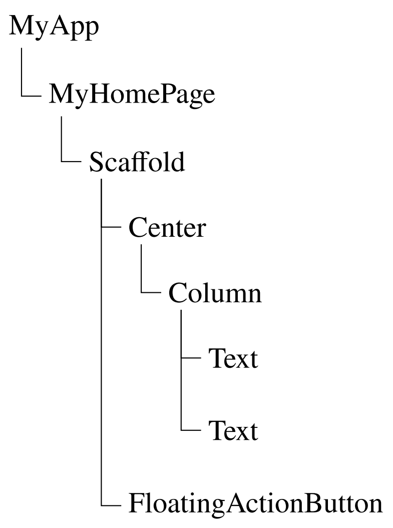


Figure 2.4: Code sample Figure 2.3 as widget tree

2.2.3 Dart

Flutter uses the object-oriented, client-optimized programming language Dart, which is developed by Google and it was released in 2011 (Google, 2021a). Dart also includes further features such as garbage-collection type inference, and improvements for better performance. It is used for Web, Desktop, Mobile, and even building server applications. The syntax of Dart is mostly influenced by C, C++, and JavaScript and can be compiled to native code or JavaScript (Wikipedia, 2021b).

The current version of Dart is version `2.13` and its latest major feature is supporting sound null safety, which was introduced in `version 2.12` (Dart, 2021; Google, 2021a). Null safety lets add a `"?"` the the type of variable, which means that this variable can set to null. By using null safty the most important advantage is, that the application won‘t crash anymore when this variable is set to null during runtime. Listing 2.4 shows some language features.

With the Dart package manager pub which is hosted on pub.dev, packages can be added to the Flutter project as new dependencies to extend the Flutter framework.

```
/// Represents a class of a person.
class Person{
  var name;
  String? surname;

  Person(this.name,{this.surname});
}

/// entry point of the program.
void main() {
  var person = Person('John',surname:'Doe');

  /// prints the name Max to the console.
  print(person.name);
}
```

Listing 2.4: Object Oriented example in Dart

Listing 2.4 is an example written in Dart. First, it shows how a class named `Person` can be created. It has two variables: a non-nullable variable called `name` and a nullable one called `surname`. These variables are set in the constructor of the class. The first parameter is the variable `name` which is required to be set. The second parameter is an optional named parameter which does not be set. Below the class, the code example shows the `main` method which is the entry point of the dart program. A new object called `person` will be created with the values `John` for the name and the optional `surname` parameter is set to `Doe`. The last line of the `main` method prints `John`, which is the variable name of the `person` object.

2.3 Related Technologies

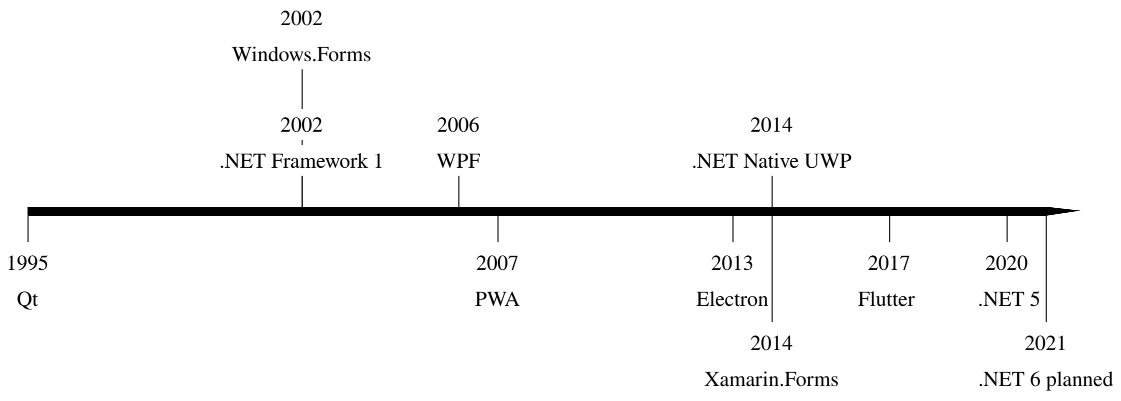


Figure 2.5: Chronology of framework releases

Figure 2.5 shows a brief timeline of the evolution and release dates of important frameworks which are used for Windows desktop applications. Above the timeline, the figure features frameworks which only target the windows operating system are shown. These frameworks will be discussed in more detail in section 2.1. Underneath the timeline different cross-platform solutions are shown.

In 2002, Microsoft developed the first version of .NET API with the **Windows.Forms** UI framework (Wikipedia, 2021c). In 2006 Microsoft developed WPF, which was shown in more detail in section 2.1. **Universal Windows Platform (UWP)** is also developed by Microsoft and was released in 2008. UWP was designed to have one application with one framework to target many different devices. The target devices include personal computers with Windows, HoloLens, XBox, Surface Hub and especially the former Windows Phone.

"**.NET Multi-Platform App UI (MAUI)** is a cross-platform framework for creating native mobile and desktop apps with C# and XAML. Using .NET MAUI, we can develop apps that can run on Android, iOS, macOS, and Windows from a single shared code-base." (GitHub, 2021c). MAUI is the next major step after Xamarin.Forms with more advantages and improvements in startup tracing, faster rendering and more performance (David, 2021).

Qt is another well-known cross-platform framework to mention, which was already developed in 1995 and some companies still use this framework. The application can run on many different platforms such as mobile phone, Desktop, and even in cars. On Windows, it is used by AMD and Intel (Wiki, 2019) for driver settings UI.

Progressive Web Application (PWA): Web based frameworks are a different approach developing Desktop applications. Those frameworks make it possible to run an application in almost every Web browser on any device. The most known Web frameworks are Angular, React Native and Vue. PWAs are close to native applications, they also have access to the operating system, they work offline and they also have native integration with features such as Desktop notifications (Khan, 2018).

Electron is an open source cross-platform framework running Web applications on a desktop environment. Electron consists of three components. Electron includes the Chromium‘s rendering librariy, the JavaScript library Node.js which is built upon the V8 JavaScript engine and an interface for common operating system operatons (Rieseberg, 2021). In addition it includes the package manager npm, which provides many additional functionalities such as testing or debug tools.

3 State of the Art

As mentioned in section 2.2, Flutter targets Mobile, Desktop, and Web. Even though the framework is already popular between developers, it has not reached much attention in research yet. It has not received much research despite being a popular framework amongst developers. WPF existed for a similar framework, developed in 2006 by Microsoft that also seems to lack extensive research. WPF has existed for a longer time. Nevertheless, there is not much research, which has been done about WPF. Reasons are likely that in universities Linux and Web technologies are preferred.

3.1 Related Work

The bachelor thesis from Müller [2021] covers a use-case study. The study compares a dictionary application written in Electron and in Flutter running on a Linux operating system. Since Flutter is a pretty new UI tool-kit, the goal is to find out how the Flutter application runs on a desktop computer and compare measured metrics. This experiment was limited to the Linux operating system. The hypothesis for this research was that Flutter as a Desktop application is not yet a viable choice for Desktop development, but still can be used for it with improvements and further developments. The research concludes that there are still missing features which are needed to work on a bigger project with Flutter. It further concluded that Flutter is not a revolutionary framework, but could be a viable choice in the future (Müller, 2021).

Another study for consideration in regards to Web frameworks for Desktop applications was conducted by Scoccia and Autili [2020]. "This study fills the gap that Web frameworks have not been empirically studied so far"(Scoccia et al., 2020). The goal of the study is to find out how the Web frameworks have an impact on their advantages and disadvantages during the Desktop applications development. For the methodology, they analyzed 453 desktop web applications published on GitHub and performed quality and quantity analyses to find traits and issues. As a result of the analyise they found out that developing Web applications with a team can work well. Obstacles can be lack of compatibility and bugs regarding the target platform. The study did not investigate any of requirements for the development itself and maintenance work (Scoccia et al., 2020).

Another bachelor thesis by Olsson [2020] compares the performance and look and feel between Flutter and native applications written in Kotlin for Android and Swift for iOS. She implemented an application to compare the metrics and came to the conclusion that the CPU performance is almost identical to native implementations. In addition, she created a survey to find out if people notice a difference between the Flutter UI and the native one. The survey concluded that Flutter mimics the native look, but still differences remain. Furthermore, she concludes that flutter performs well and is a good choice for Mobile development (Olsson, 2020) .

Finally, the bachelor thesis by Wu [2018] compares Flutter and React Native in its characteristics. He developed reference applications to compare them in different areas. Wu compared the applications in performance metrics such as disk space usage, scrolling in the app and frame per seconds (FPS). In addition, he compared the possible state management options. He came to the conclusion that React Native is an efficient framework and that it is a convenient platform to develop with. Wu also concluded that Flutter has good tooling support already and applications written in Flutter perform well (Wu, 2018).

3.2 Contribution

Since Flutter is a relatively new technology, there is not much research out there. Still, the studies above are covering almost all target platforms for which Flutter can be used. The studies cover the comparison to Mobile development, Web and Electron on a Linux Desktop using Web technologies. As a comparison tool they developed an application with which they use to collect data for different metrics. Most metrics seem to be CPU and RAM. In addition they use the code analysis to find out if there are many differences in line of code and code complexity. In our research we could not find any research about Flutter targeting Windows platform.

4 Reference Application Development

In order to answer our research questions, we implement a simple image gallery. An image gallery combines common features for many users. Figure 4.2 shows the first mock up of our implementation. In the next sections, we describe our features and important issues during our implementation in WPF and Flutter.

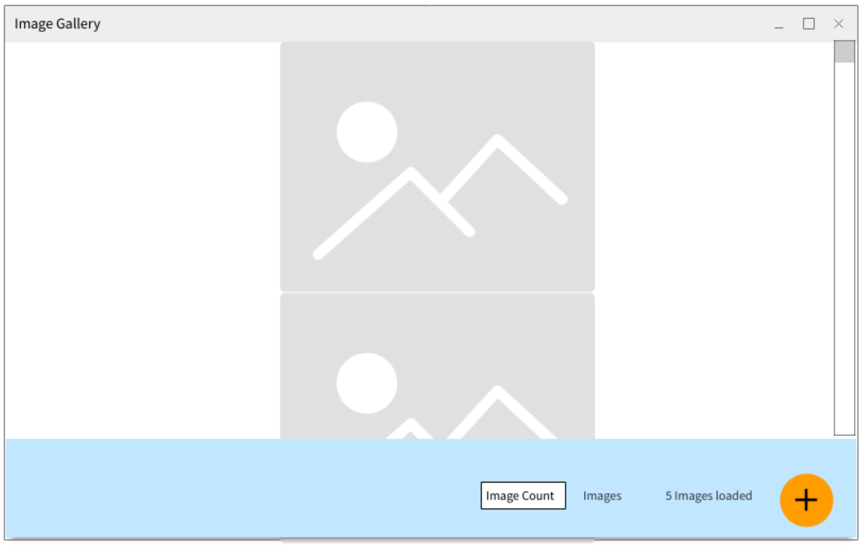
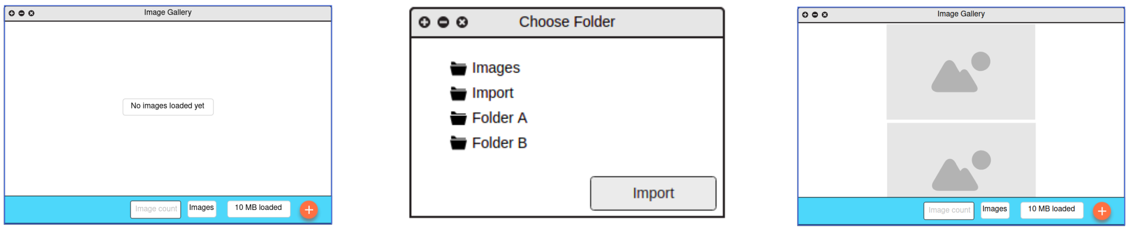


Figure 4.1: Mockup image gallery application

4.1 Features

For the reference application we found two features to be important for both the user and developer. The first feature is to import images (see 4.1.1) and the second is to display an image in the size of the application (see 4.1.2) with its details.

4.1.1 Import



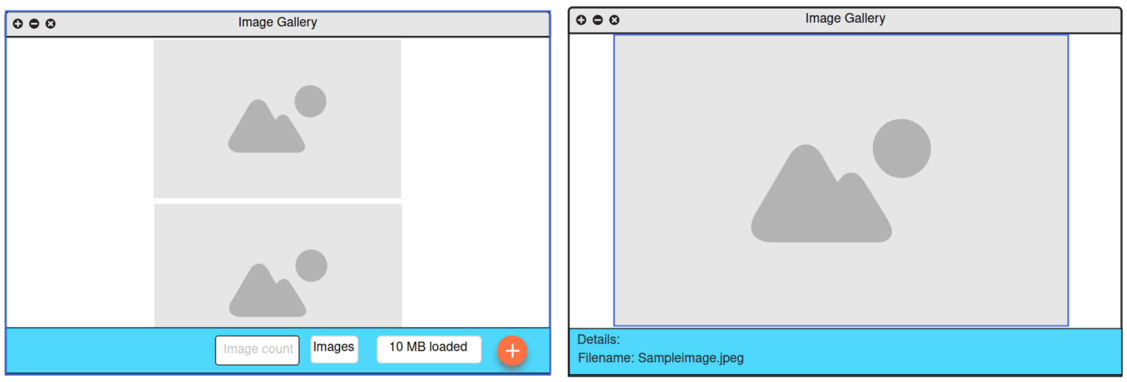
(a) Initial **(b)** Choose folder **(c)** Load images

Figure 4.2: Import procedure

Figure 4.2 shows the procedure on how we can import the images. In the initial state (a) no images are loaded yet. Before we import images, we can specify how many images should be loaded from the folder by typing the number in the textfield on the bottom of the application. For importing an image folder, we click the import button which opens up the import dialog where we can choose the folder to import. We accept the choice by clicking on the import button (b) and the application starts to import the images contained in the selected folder, asynchronously (c). When there are more images than what is displayed on the start page we can use the scroll view to see all of the images.

We added this feature to showcase some different features of the framework. First we use a native option to open a folder picker. Next, we are interested in how the frameworks performs importing images. In addition, we are scrolling through images, which enables us to see how well each framework perform scrolling through a larger list of images.

4.1.2 Detail View



(a) Initial loaded images **(b)** Detail page

Figure 4.3: Detail view Procedure

This feature is only active when images are loaded (a). We can click on an image to see its details. The image pops up in the size of the application window and displays its details below the image (b). We only display the most important information to illustrate where the details can be listed.

We added the second feature, because we wanted to find out how the loading in full size affect the metrics on each framework.

4.2 Implementation

For the implementation it is not only necessary to implement the features itself. It is also important to structure to code itself. We can structure the code with the Model-view-viewmodel (MVVM) state management pattern. In WPF this pattern is already best practice to use this pattern compared to Flutter there is not a best practice, but rather there are different options to choose. The MVVM pattern allows us to separate the UI and its logic and the data itself. MVVM consists mainly of four elements The user interacts, which is a passive view and without any logic included. Each view has a view model, where the logic of the view is implemented. It handles the user interaction and passes the data to different services and back to the view to update its state. The services provide the handling to external components such as a database or a web service and consumed by the view models.

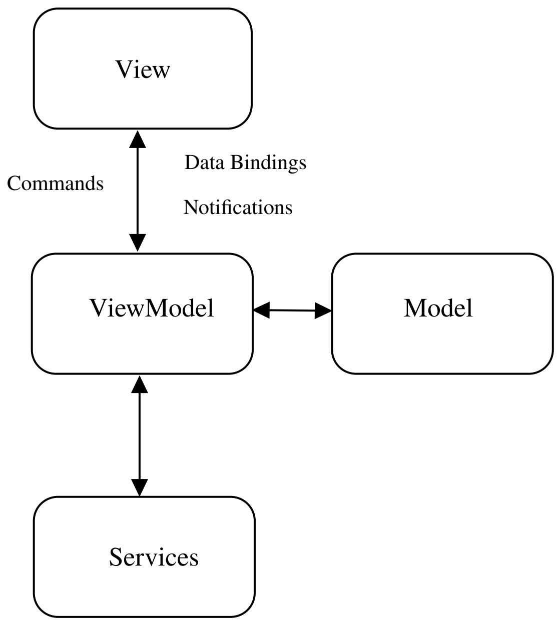


Figure 4.4: Model-view-viewmodel pattern

For the MVVM pattern there are not only advantages, but also disadvantages. In the following there are some of them listed (Bonniwell, 2018):

Advantages:

- Speed and performance enhancements
- Faster screen saving/loading
- Developer team can be split into front-end and backend developers for more efficiency development

Disadvantages: MVVM not only creates advantages, but also disadvantages, which can affect especially in small teams:

- It creates much overhead for simple applications
- High memory consumption in large applications
- Complexity of the application increases

For our implementation we designed our structure as shown in Figure 4.5. We added the view `GallerImagePage` which communicates with the view model `GalleryImagePageViewModel`, which includes the logic to import images from the folder explorer. The In both frameworks we could not face any problems to implement the pattern.Though we faced to some other obstacles, which we focus in the following sections 4.2.1 for WPF and section 4.2.2 for Flutter implementation. We also show which steps we took to configure our project first. In addition, we show how we implemented the first feature in both frameworks.

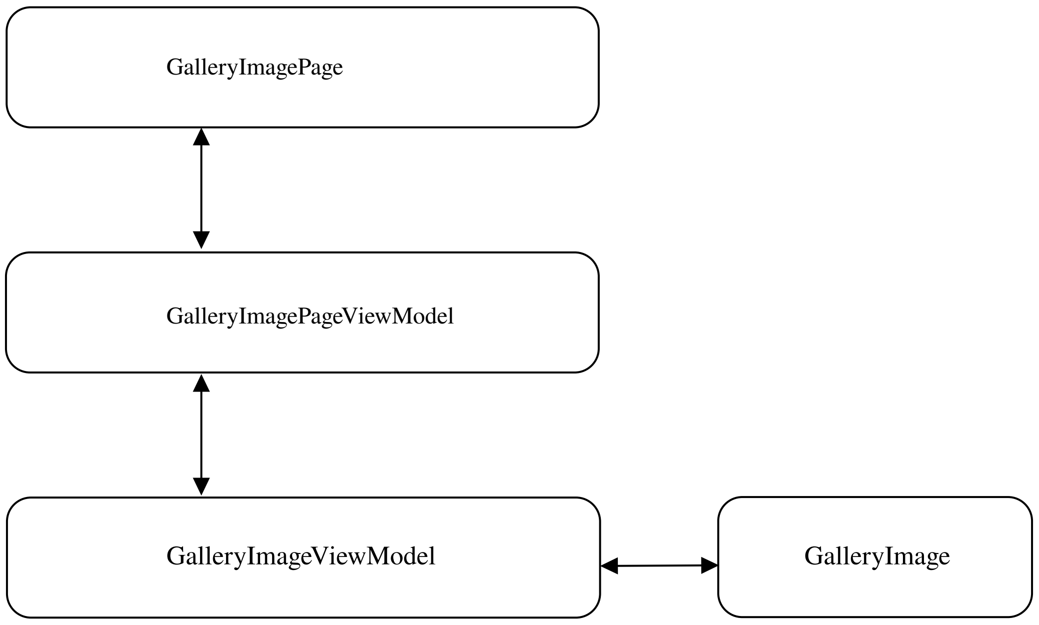


Figure 4.5: Structure of application

4.2.1 WPF

For our WPF implementation, we started to configure our project first. Listing 2.2 shows our configuration of our application, which is an updated version of the configuration we mentioned in the Background information chapter. First, we set our target Framework to `net5.0-windows10.0.19041.0`.

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>WinExe</OutputType>
    <TargetFramework>net5.0-windows10.0.19041.0</TargetFramework>
    <TargetPlatformVersion>7.0</TargetPlatformVersion>
    <UseWPF>true</UseWPF>
    <UseWindowsForms>true</UseWindowsForms>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="MahApps.Metro.IconPacks" Version="4.8.0" />
    <PackageReference Include="MaterialDesignThemes.MahApps" Version="0.1.6" />
    <PackageReference Include="Microsoft.UI.Xaml" Version="2.5.0" />
  </ItemGroup>
</Project>
```

Listing 4.1: Used Packages in WPF application

Next, we added the possibility to activate the usage of WPF itself. By default, WPF has no access to use the native file explorer. We were able through Windows-Forms to access the file explorer with the help of WindowsForms by adding the attribute `<UseWindowsForms>true</UseWindowsForms>`. At last, we added the dependency `MaterialDesignTheme.MahApps` and `MahApps.Metro.IconPacks` to be able to use `Google Material Design`, which is also used by our Flutter application. These updates were our first obstacle, because we have not used .NET 5 for a new project. When the configuration was done, we could start the implementation of our application. First, we started with the UI itself. Listing 4.2 shows a snippet from the start page. For our images, we used a simple list box, it is bound to the view model of our Images saved in the property `GalleryImages` and used in line 7 in Listing 4.2. To limit the images in the list view, we set the height to 300 pixels.

```
1  [...]
2  <ListBox
3      [...]
4      SelectionChanged="ListView_SelectionChanged"
5      ItemsSource="{Binding GalleryImages}">
6      <ListBox.ItemTemplate>
7          <DataTemplate>
8              <Image Source="{Binding GalleryImage}"  Height="300"/>
9          </DataTemplate>
10     </ListBox.ItemTemplate>
11 </ListBox>
12 [...]
```

Listing 4.2: List view of images in our WPF application

We implemented the structure mentioned in Figure 4.5. During development we did not face more issues than we did during the Flutter implementation.

4.2.2 Flutter

The development of a Flutter application for Windows is almost identical to developing in Flutter in Android, which we developed in Android Studio. The only difference is, that we had to specify the build target by using the command `flutter build windows` (Flutter, 2021a). Before we could start to implement the application itself, we added additional packages to the dependencies (see Listing 4.3) which are necessary to implement our features. We discussed them in section 4.1.

```
get_it: ^7.1.3
provider: ^5.0.0
file_selector: ^0.8.2
file_selector_windows: ^0.0.2
file_selector_platform_interface: ^2.0.2
url_launcher: ^6.0.3
path_provider: ^2.0.1
```

Listing 4.3: Used packages in our Flutter application

One of these packages is for accessing the file explorer. Since the package manager has lot of different packages, there are also similar packages which have the same goal but are developed by different developers. In our case, we first saw the package `File Picker`, but found it to not be available on Windows yet. We ended up using the package File Selector as indicated in in Listing 4.3. Because the File Selector was not fully implemented in one package, we needed to add two additional packages `file_selector_windows` and `file_selector_platform_interface`. Those packages enabled us to use a folder selector for our application. In chapter 7, we discuss more about packages and their limitations in the current stage of Flutter.

```
[...]
Scrollbar(
  isAlwaysShown: true,
  child: ListView(
    padding: const EdgeInsets.all(12.0),
    children: model.images.map((image) {
      return ImageView(image: image);
    }).toList(),
  ),
)
[...]
```

Listing 4.4: List view of images in our Flutter application

Next, we implemented the list view for our image in the start page of our application. Listing 4.5 is part of the implementation. During our simulation, we recognized the scroll viewer would not appear automatically. To fix this, we added a scroll bar widget that surrounded the list view, so that it is visible the whole time. While implementing the feature, we saw that the levels of widget trees grow very fast. The widget list view needs a child for the image to display. To follow a good style, we extracted images into a separate `StatelessWidget`, which we called `imageView` and used it in Listing 4.5 line 6. Listing 4.4 shows the code snippet of the image. For our feature to display the image in full size, we surrounded the image with the `GestureDetector` widget, which allows it to handle click events. We used the click event to go to the Detailpage, which displays the image in full and lists the image details underneath.

In chapter 2 Background Information we saw how the standard state management can be used by using the method `setState` in a `StatefulWidget`. In this method, the logic is integrated into the UI tree itself. This state management leads to some difficulties for testing the application and adding additional features to it. That is one more reason to use the MVVM pattern which we showed at the beginning of this chapter. Unfortunately, there is no best practice for implementing the MVVM pattern in Flutter, but rather there are different packages and we decided to implement the MVVM solution by FilledStack, since we could work with this implementation easily.

While experimenting with the the simulation script (see chapter 5) for the Flutter application, we noticed that the function for entering text automatically in a text field, which is provided by the Python library `PythonAutoGUI`, did not work. We could not find any solution on their GitHub page or in other sources on the Web. That caused a deviation in our process in our simulation tool. In the simulation of the scenario we left out typing the value in the text box automatically. We did not need to add any waiting time, because both simulations WPF and Flutter simulate the next steps at the same time. Instead we set the value of the text field to 30 by default. Other obstacles we faced during development were the error messages while we created the UI. The messages are unclear, which took us a while to understand and to figure out what went wrong.

Summarized, the implementation in Flutter was quite easy to finish.The structure of this declarative programming language enables a straightforward implementation of the widget tree, which also allows a fast realization of the applications. In total we needed more time to finish the Flutter version due to the above mentioned obstacles and problems we faced. At the end we could create both application, which look almost the same. We added screenshots of both implementations in Appendix A.1 and Appendix A.2.

```
ImageView{
[...]
GestureDetector(
        onTap: () {
          model.goToDetailPage(image);
        },
        child: Padding(
          padding: const EdgeInsets.only(bottom: 4.0),
          child: Container(
            height: 300,
            child: Image.file(
              File(model.galleryImage.path),
              fit: BoxFit.scaleDown,
            ),
          ),
        ),
[...]
}
```

Listing 4.5: Image widget from our Flutter application

5 Experiment Setup

After we developed the reference applications, we set up the experiment to measure our metrics. First we list hardware and software that we are using for the experiment and show distortions which can occur. Additionally, we define our test scenarios with aforementioned features in section 5.1.

In section 5.2, we show how we measure RAM usage, CPU usage, the startup time, packaged size and code size. In section 5.3 we describe how we run the simulation tool to retrieve results. For our experiment, we used one setup for all measurements:

Hardware	CPU	AMD Ryzen 5 2400 G
	GPU	Radeon Vega Graphics 3.600 MHz
	System Memory	32 GB
Software	Operating System	Windows
	Windows Version	10.0.19042.985
	Flutter Version	2.0.1
	Simulation script	

Figure 5.1: Testsystem configuration

Figure 5.1 shows our hardware and software configuration. We run our experiment only on Windows version `10.0.19042.985` for WPF and for Flutter we use version `2.0.1` with Dart `2.13`. We perform our scenarios with the release version of both frameworks. To measure our metrics, we used our developed simulation script in Python.

During the experiment, we recognized that there could be different situations which lead to values which can affect our results of the measurements. To avoid this, we tried to minimize the following distortions to get the best results. First RAM and CPU need to be cleared, so there are no caches stored from our application or running unexpected processes in the background. During each run, we measure RAM usage and CPU usage for the whole system. Additionally, Windows may start processes in the background which could influence our measured values. As a solution we first set up our test system in a way that there are no further programs which start after rebooting our test system for each test run.

5.1 Test Scenarios

For each feature (see section 4.1) we decided to create a test scenario. In each scenario, we we retrieve a value every 0.5 second, which seems a good intervall to evaluate the results.

Import: Table 5.1 describes the steps we took during our import scenario. As for the first step, we specify that we want to load 30 images (step 1) from a folder which we select in the folder explorer (steps 2 and 3). After selecting the folder with the images, the program loads all 30 images into the listview of our start page. The image size is between `95 Kilobytes` and `20 Megabytes` and `107 Megabytes` in total. For the loading process (step 4), we defined a waiting time for 60 seconds for a clear result, which we can see in chapter 6. When all images are loaded, we want to simulate, user behavior by scrolling down the scroll view within 7 seconds (step 5). For the last step we scroll up again within 2 seconds (step 6).

Step	Second	Action
1	1	Type 30 in textfield
2	2	Open folder explorer
3	6	Import image folder
4	7 - 67	Loading images process
5	68	Scroll view to bottom
6	75	Scroll view to top

Table 5.1: Action during import scenario

Detail view: In the first scenario, we were focusing on the import of images and how the frameworks differ in that regard. Table 5.2 describes our second scenario. We start the application where two images are already loaded (step 1) automatically with the size of `15.8 Megabyte` in total. After the loading process, we click on the first image to display the image in the size of the application with details placed below (step 2). When we click on the same image again, we go back to the start page (step 3). We repeat that for the second image (step 4 and 5). In the last steps, we want to observe if there is a difference when we repeat steps 2-5 again. For a better result, we added a waiting time for 3 seconds after each click.

Step	Second	Action
1	0 - 3	Import 2 images
2	3	Click on first image
3	6	Click on image to go back to start page
4	9	Click on second image
5	12	Click on image to go back to start page
6	15	Click on first image again
7	18	Click on image to go back to start page
8	21	Click on second image again
9	42	Click on image to go back to start page

Table 5.2: Actions during detail view scenario

In addition to both scenarios, we added a third scenario, which measures the idle state of our operating system. We define the idle state as follows: our application is not running. Only our Python script and Windows itself is running.This scenario allows us to understand how much the operating system needs without our application running. Therefore, we can subtract the idle values from the values of our first scenario to get the exact values on how much CPU and RAM our application uses.

5.2 Metrics

In our research we are focusing first on existing tools which we could use to measure the metrics CPU, RAM. But we could only find profilers which are used for only one programming language. For example the built-in profiler in Visual Studio for .NET is used for WPF but can not be used for our Flutter application. We came to the conclusion that we need our case the best solution to develop our own Python scripts. This way, we can also design the scripts specially to target our application.

Before we can compare our applications we need to define metrics. Each metric needs to be specific, relevant, and measurable while our scenario is running. In our Python script, we made use of the cross-platform library `psutil`, which allows us to measure RAM and CPU usage.

5.2.1 RAM Usage

In our Python script, we use the method `psutli.virtual_memory()` which returns the RAM usage value in bytes back (Psutil, 2021). The amount of RAM we measure describes how many bytes are being used on the operating system. It includes the operating system itself and any programs which are running at that moment.

5.2.2 CPU Usage

The CPU usage describes how much of the CPU‘s total capacity is used within a specific time (Wikipedia, 2021a). We use the method `psutil.cpu_percent(interval = 0.5 seconds)` to measure the CPU and collect the value with an interval of 0,5 second. This metric is relevant to see how efficient the framework works in our application.

5.2.3 Startup Time

We define startup time as the following: We start our stopwatch the time when we start our application and end when all components are loaded for the first time and we can interact with the application.

The startup time metric is especially interesting for the developer cycle, because the application needs to restart often to verify and test the updated code. For the user itself it is negligible, because the application usually gets started once a day. Also nowadays the personal computer has enough capacity to start a program without a large lack of performance.

5.2.4 Packaged Size

On Windows there is not a general way to package an application so we decided to use create a `zip` which contains the compressed executable and its necessary binaries for the dependencies. We then compare the size of the archive. The packaged size it is relevant for example to upload the package into the cloud where it can be distributed to different clients.

5.2.5 Code Size

The line of code can play a big factor in bigger projects. When a feature can be implemented with less code, it might lead to less maintainable code, which also implies less bugs. We then used the open source program Sloc Cloc and Code (scc) to count all code lines of each project (Boyter, 2021). It was necessary to execute the executable in the terminal `./scc.exe` inside our project. The tool automatically generates a table displaying how many lines of code are used grouped by programming languages.

5.3 Test Execution

```python
1  def run(args):
2      we = MeasureRun(args)
3      we.start_app()
4      we.wait_for(seconds=5)
5      we.start_measurements()
6      we.run_scenario()
7      we.stop()
8      we.wait_for(seconds=8)
9      we.stop_app()
```

Listing 5.1: Simulation script in Python

For our simulation script, we use the Python library `PyAutoGUI`. This library enables us to simulate our scenarios, which we defined in section 4.1. The simulation provides accurate results for a representative comparison. Before we can start the simulation, we need to make sure the applications are centered on the screen. Listing 5.1 shows the routine of our simulation script to measure our metrics for both applications. We summarize our routine as follows: First we start the application (line 3), second we start to measure the usage statistics (line 5). Line 6 starts the simulation we have specified in our script all (see Figure 5.2). When the scenario is over, we stop measuring and end the application. Our script receives three parameters. The first parameter `--framework` specifies for which framework we run the simulation: `wpf` or `flutter`.The next parameter `--operation` specifies which scenario we want to measure, which allows the values `import` and `detailimage`.

The last parameter `--attempt` stands for at which run we are at. We decided to repeat our experiment twenty times, as it seems to be a good repetition to get a broader view of the results, in which we can calculate meaningful median of our result.

```
python measure.py --framework wpf --operation import --attempt 1
```

Figure 5.2: Command in our test script

6 Experiment Results

After we executed our test twenty times, we got our raw data from each run. We can compare the results from our scenarios (see section 5.1), we needed to process the data first. We recognized the idle values from our third scenario were not always zero. It seems that on our test system, Windows needed to execute programs in the background. First we thought we made a mistake so we started to measure it a second time, which did lead to the same results. So we decided to subtract all values from the averages of our results and the idle values. As a result, we had some negative values, because some values from the idle state were greater than we measured in our scenarios. We knew that the values of the idle scenario should be zero, which led us to replace all negative values with the value zero. The results then seemed more realistic for further investigation.

To enclose our results is to mention that they can not be seen as a general performance test for the whole framework, but rather it is a specific view of how the frameworks perform processing and displaying images to the screen. We have also seen that scrolling in Flutter works much smoother than in our WPF application. For a more detailed performance check we need necessary to verify more parts of the frameworks. Further possible investigations are illustrated including the scroll view in chapter 8.

Additionally we could not type in the amount of images in the text field from the Flutter application in our simulation automatically, which meant that we had to wait so that the timeline is parallel to our WPF version. As a result, our values might not be from the same exact time step. In the following, we relate to the steps in the scenarios, which we described in Table 5.1 and Table 5.2.

6.1 RAM Usage

Figure 6.1 shows the RAM usage for our first scenario, and Figure 6.2 the second scenario. We described them in detail in Table 5.1 and Table 5.2. In both figures the red line represents WPF‘s RAM usage statistics and the blue dotted line Flutter‘s RAM usage. In the both scenarios there is in common that WPF needs more RAM after the application has started, during importing, and displaying the images.

Focused on each scenario, we need to mention that the first scenario shows that WPF needs up to `584 Megabytes` for importing the images (see steps 2 Table 5.2) which is about twice as much as Flutter needs. We also see that WPF needs around five seconds more time to process the images until they are imported and displayed to the screen. At the end of the scenario during the steps 4 and 5, we recognize that using the scroll viewer in Flutter is less efficient in contrast to WPF. The

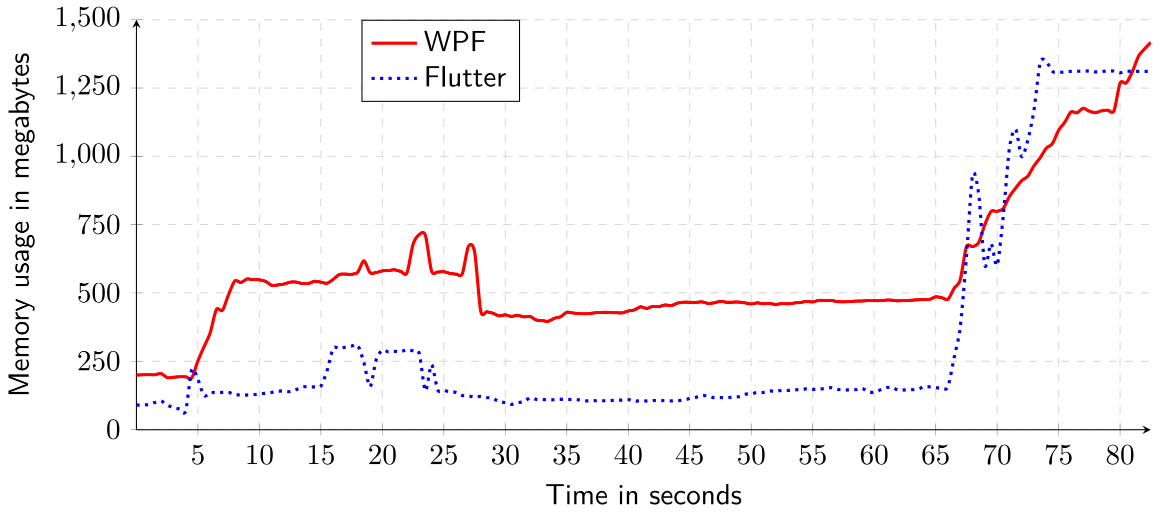


Figure 6.1: System memory usage during import images scenario

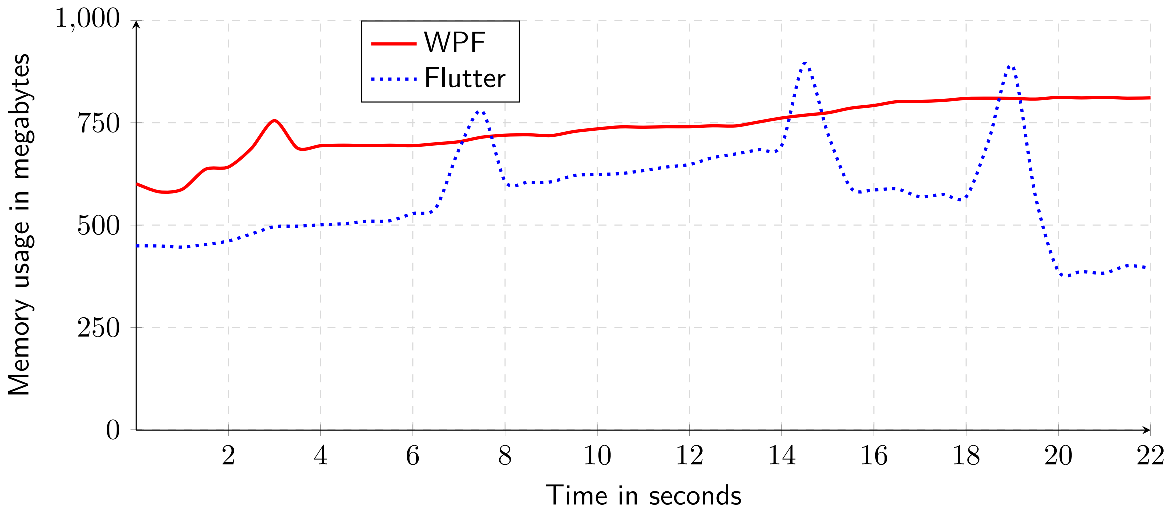


Figure 6.2: System memory usage during detail view scenario

RAM usage in WPF increases steadily as expected, because we loaded all images at once and not when they are seen on the screen. Surprisingly, our WPF implementation uses more RAM than in our Flutter version, when the scenario is done.

The results of our second scenario look as expected. As mentioned in subsection 4.2.2, we navigate to a new page to display the image in full, which leads the Flutter framework to reload the images after we go back. In our scenario, we can see this effect in time steps 3,7, and 9, which represent the peaks in the graph.

In conclusion, Flutter consumes in both scenarios less RAM compared to WPF for the majority of the time. One reason could be that Flutter processes and uses the RAM in a more efficient way. The image process can be affected by the framework itself or our implementation.

6.2 CPU Usage

Figure 6.3 and Figure 6.4 show the CPU usage for each scenario. In both figures, the red line represents the measured values of the WPF application and the blue dotted line represents the Flutter CPU usage statistics. We have seen in the previous section that WPF needs more RAM most of the time, in contrast our result in CPU usage shows that Flutter needs more CPU in both scenarios almost through the whole scenarios.

In the first scenario, it is significant that after loading the images, Flutter still needs 3% and 18%, even when the images are already imported. During the scroll view in step 5 and 6, Flutter needs up to 50% which is 40% more than WPF.

In the second scenario, it seems Flutter loads the images much more efficiently than WPF, which we can see at the beginning of the scenario in step 1. It is also remarkable, that Flutter and WPF needs only 2% CPU to display the first image in full screen for the first time, but in step 6 Flutter needs up to 23% CPU for the same scenario. The other peaks in steps 3, 5, 7, and 9 occur when the Flutter application goes from the detail view back to the start page. We assume the higher distinction, because of the difference in implementation (see subsection 4.2.2).

In conclusion Flutter uses more CPU compared to WPF. The higher CPU usage indicates that the Flutter Framework processes more for handling the images to reach the lower RAM usage we have seen in the previous section.

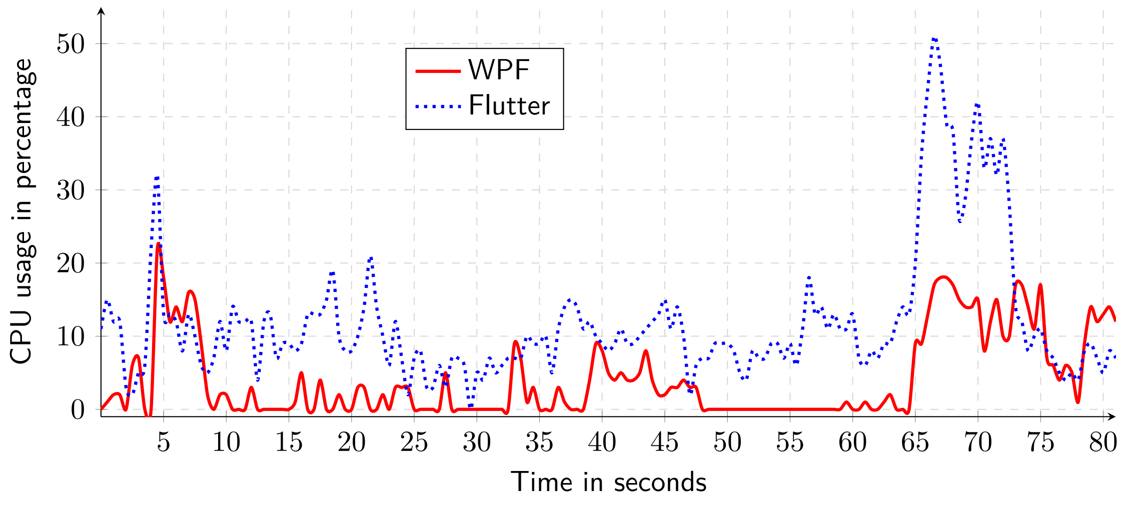


Figure 6.3: CPU usage during import scenario

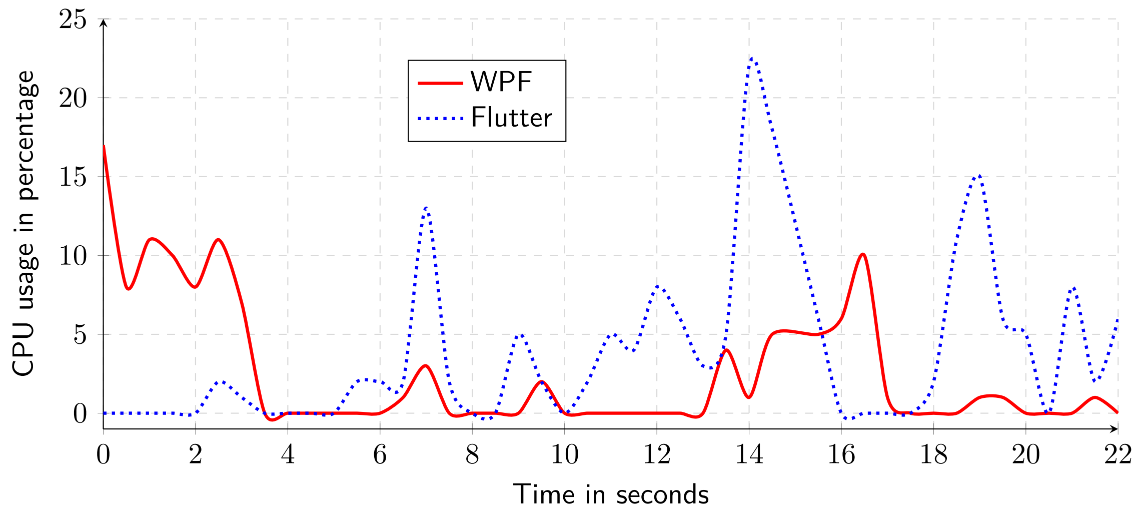


Figure 6.4: CPU usage during detail view scenario

6.3 Packaged Size

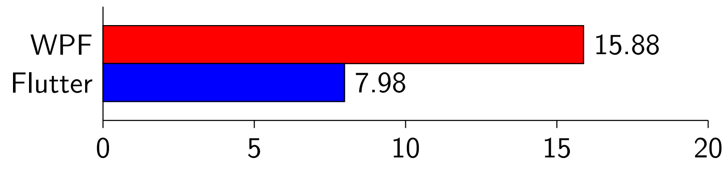


Figure 6.5: Packaged size of applications in megabytes

The Figure 6.5 shows the size of the zipped applications. Our Flutter application is almost 50% smaller, which relates to the fact, that we needed less code for the application (see section 6.4 and we assume that the Flutter libraries are smaller. These sizes are relevant to know for distributing the application. The only thing which is different in both applications is that we added the `Material Design` to the WPF application. So without these dependencies the WPF application is `6.231 Megabytes` which is then 22% larger than the Flutter package.

6.4 Code Size

Table 6.1 and Table 6.2 show the static code analysis of our applications. For our result we are only interested in the code lines which we were coded by ourself. For the WPF we only used C# and XAML, which in sum we used `1.508 lines of code` to implement our application. Compared to the Flutter we used only Dart with `832 lines`. The other programming language are auto generated and are needed to be able to execute the application on Windows. For the Flutter application it seems significant that it needs several more files to create the underlying Win32 application which starts our Flutter application.

Programming Language	Files	Lines	Code
C#	20	1260	978
XAML	5	248	240
JSON	1	10	10
Total	26	1518	1228

Table 6.1: Code analyse for WPF application

Programming Language	Files	Lines	Code
C Header	42	3205	1670
Dart	20	832	692
C++	12	1351	1052
XML	7	115	64
CMake	5	250	32
Shell	1	13	2
YAML	1	80	21
Total	88	5846	3533

Table 6.2: Code analyse for Flutter application

6.5 Startup Time

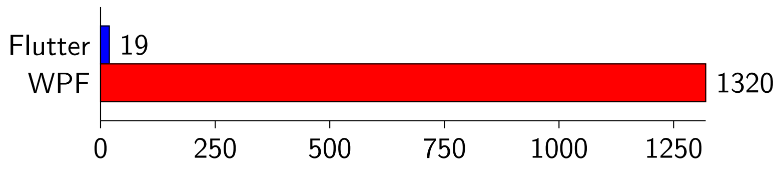


Figure 6.6: Average startup time in ms

In our experiment we used the executable `.exe` file to start our application. The above Figure 6.6 shows the average startup time for both application Flutter and WPF. We see surprisingly that WPF needs `1320 ms` in average until the application is started. In contrast Flutter needs `19 ms` on average.

7 Flutter Feature Completeness

7.1 Packages

The website for the packages is pub.dev. On its landing page you can see first the favorite packages which are being used for flutter. During our research the three popular packages are `image_picker`, `shared_preferences` and `flutter_cache_management`. Flutter targets different platforms so we can filter the packages by the platform and how popular the packages are.

Figure 7.1 shows a short overview of how many packages from the first 100 most popular packages are available for developing each platform. Since Flutter has started to target mobile platforms Android and iOS, there are the most available packages with 99 out of 100. For Windows there are already 75 of them available. As previously mentioned, Flutter for Windows in beta stage is quite remarkable since there are already so many popular packages available. The most popular package available for every platform is a `path` package that allows us to work with filepath operations. The package `http` provides basic high-level functions to consume Hypertext Transfer Protocol (HTTP) resources (Pub.dev, 2021a,b). Since the Flutter framework consists of minimal functionalities we need to use many dependencies bigger projects. In our research we found that there are conflicts between the used versions of a package. An example is to use the package with version `http 1.01` and an other package with version`1.12.0` is not possible. To resolve this conflict the package needs to be updated, so that all packages and the project itself use the same package version. It can be much work especially when a bigger project which needs many packages. To prevent this conflict it also helps to verify who develops the package and how often it gets updated. In addition to these issues, it is important to make sure the package is available on the platform the application should run on. The development of packages is open-source, so every developer can develop a package and publish it on the pub.dev website. So it the completeness of the functionality is not guaranteed.

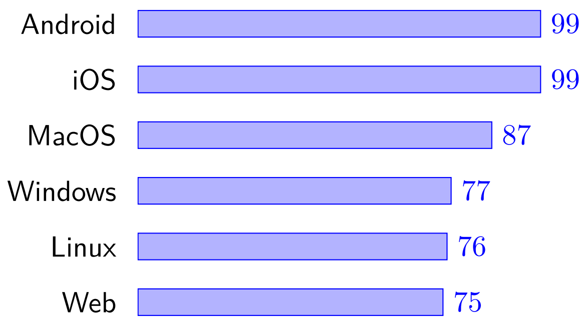


Figure 7.1: Availability of the top 100 most popular Dart packages by platform (Müller, 2021)

Package Name	file_picker 3.03	filesystem_picker 1.0.4
Target Platform	Android, iOS, Web	Android, iOS, Linux, MacOS, Windows

Figure 7.2: Example of similar packages with different target platforms

Figure 7.2 shows two corresponding packages for providing the same solution to a file picker in the application with different target platforms. For example, the package `file_picker` does not target Windows. Therefore, it is also necessary to verify the target platforms before it can be used in the application. Since Flutter is in an early stage, it is likely that many packages will add more target platforms in the future.

7.2 Features

Since Flutter is open-source and is developed on GitHub, it is possible to participate in new features which will be implemented into the Flutter framework. During our research, there were more than 5.000 issues open, which includes improvements and new features. For Windows in particular, there are 283 issues open. We found the following issues the most important. The following features are already listed in Github and some are already work in progress:

- **Resize window with Flutter**: In our app we changed the window size before its initialization to fit our needs. While researching, we could not find a package which worked for our case to size the window. We were required to change the window size in the native C++ Code directly. The configuration for the window size is in the `main.cpp`. This file is located in the runner folder.
- **Screen center position**: In Flutter it is not possible to start the application window in the center of the screen on Windows. It would have helped us for our experiment where it was necessary to center the application first. For this reason, we also updated the start position of our application in the `main.cpp` file (GitHub, 2021a).
- **Textfield shortcuts**: This feature would enable the use of shortcuts in text fields. For example `Shift+End` would set the cursor to the end of the text inside the textfield (GitHub, 2021d).
- **UWP support in plugins**: Our application will be launched from an underlying Win32 application to run it on Windows. With this feature it is possible to replace the Win32 application with an UWP application (Github, 2021d).
- **Window fullscreen mode**: Currently Flutter it is not possible to set the application window to fullscreen (Github, 2021c).
- **Context menu**: A context menu is not available on the Desktop. On GitHub the developers are still discussing which features it should include and how it is the best way to implement it (Github, 2021e).
- **System tray icons for desktop**: The system tray icons are important on the Desktop to display which applications are running in the background. For now, it is undecided whether it will be added to the Flutter framework or offered as separate package (Github, 2021a).

- **Application appears always on top:** Currently, it is not possible to configure the Flutter application, that the window stays always on top of all applications. As an example on GitHub it is mentioned the Task manager (Github, 2021b).

7.3 Widgets

For fast application development, it is necessary to have a variety of APIs which help developers to implement the desired features. One way is to use third party packages like discussed in the previous section. Disadvantages are that they need to rely on that the packages get updated for example, that they fit to breaking changes in the Flutter Framework and works in a proper way and the plugin needs to be complete. A more secure way for the developer is that the Flutter Framework provides the features. The Flutter team has many more developers who develop, test, and periodically deploy new updates to the framework. This allows the developers to be sure that the features are working properly and get improved. Accordingly, Flutter is a new and small framework with still missing features. We want to illustrate the widget menu bar and the Fluent Design which are attractive and rich the framework for more desktop development with Flutter.

Menu bar: For most Desktop applications, a menu bar is essential. Figure 7.3 shows a simple example of what a menu bar looks like in WPF. The menu bar is not yet available in the Flutter Framework, but we found a prototype on GitHub where developers are working on implementing this feature. They are not sure yet, if it will be included in the Flutter Framework or it is available as a external package.

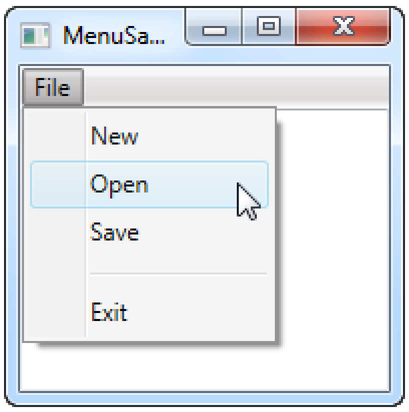


Figure 7.3: Simple menu bar in WPF (wpf-tutorial, 2021)

In the current state of Flutter, it is not possible to add a menubar to the application.

Fluent Design: For the user, it is not only important that all features are available to use, but it is also essential that the user enjoys the look and feel while he works with the application. For a better and more integrated look and feel, developers created different design guidelines for each platform. Flutter already implements these guidelines for Android (Material Design) and iOS (Cupertino), but at this stage, Fluent Design is not yet implemented. Figure 7.4 shows an example of a side menu in Fluent Design. As a step further, there is already an open issue on GitHub.

Nevertheless, we found a prototype on GitHub, which is considered as a new feature for the Flutter Framework or it will get published as a new package (Google, 2021c). Additionally, we found the unofficial package fluent_ui, which already includes most of these guidlines (pub.dev, 2021).

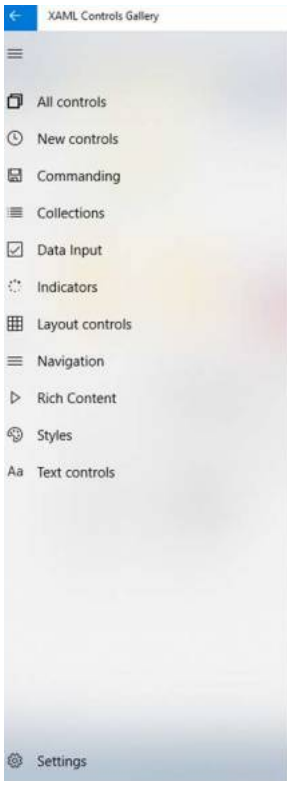


Figure 7.4: Menu in WPF Fluent Design

8 Conclusion and Outlook

Flutter is a new cross-platform UI tool-kit. It is a light-weight framework which already targeted mobile devices, the Web, and also recently Windows Desktop. We think it is a promising framework which made us curious to find out how well it already works on Windows compared to the longer existing WPF UI tool-kit developed by Microsoft. In addition, we wanted to find out which features are still missing to use this framework for a productive application. We decided to implement a simple image gallery in both frameworks for a use case study to answer our research questions which allows us to take our results and adapt them to many different other applications which are working with images. For the comparison, we chose the metrics RAM and CPU usage. We used our simulation script to collect the metrics during two different scenarios. The first scenario is about importing images into a list view and using its scroll view. The second scenario is to display an image in full size. In both scenarios, we discovered that WPF consumes more RAM compared to Flutter. In contrast, Flutter needs more CPU for the majority of the time. During scrolling, Flutter uses up to 50% CPU which is a remarkable result to observe that also tells us that Flutter is not fully optimized yet. The results of our package size, implementation size, and startup time are also surprising. The Flutter application requires 55% less code, and its packaged size is about `7.98 Megabytes` which is about 50% smaller compared to WPF. Application startup times are also 70 times faster on Flutter.

In our second part, we were researching additional packages for Flutter and verify if there are features for Desktop missing. First, we found out that Windows has 75 packages out of 100 of the first popular packages on pub.dev. The most important features for developing Desktop applications are already available. On the contrary, we recognized that significant features are still missing. The most useful feature would be to create a context menu which will be used in many applications. In addition, a widget for a menu bar, which is an important widget for Desktop applications, is still missing.

After evaluating our results, we can answer our research questions:

Research Question 1:
Is Flutter an alternative compared to Desktop application framework WPF with .NET 5?
Research Question 2:
What are the current limitations of Flutter?

As far as our first research question is concerned, we think that new developers can prefer Flutter over WPF to develop simple applications. In bigger projects, the higher CPU usage can play a bigger role which could lead to performance issues. Nevertheless, our performance results show that Flutter already performs well regarding RAM usage. Because Flutter is only in the beta stage, we are sure that Google and many other developers will improve the framework to reach more companies who are willing to use Flutter in their projects. The answer to our second research question confirms our previous answer. We think there are some limitations as mentioned above, but Flutter can be a good choice for simple applications. If special features are required, it should be verified that the desired functionality is available either in the framework or as a package.

In a nutshell Flutter is a great UI tool-kit which has potential for growth. After our research, we think it is important to first specify the features the application will need. Second, this choice is very individual and the developers' knowledge must also be taken into consideration.

At long last, it is to mention that we only could work on some functionality of the framework which gave us a specific view to answer our research questions. For further estimation and understanding of the performance and feature completeness of Flutter and even other functionalities of the framework it needs more research in different aspects.

There are many further features which help to understand how the framework is processing images. In our reference application, we loaded all images at once into the list view, which causes the high RAM usage we have seen in our results. For a further comparison, the implementation can be modified so that the images are loaded when they need to be displayed on the screen. This modification especially changes the RAM usage, and can be compared perfectly with our implementation. Our reference applications can also be extended with new features, which allow new use cases. New features could be editing images or importing videos. To rate these features, we found a suitable metric FPS in the thesis written by Wu (see section 3.1). This metric is not only suitable for importing videos in our gallery but also for our existing feature using the scroll view. It is especially interesting if the framework can hold a stable FPS or if it is fluctuating. It is not only relevant to extend the application but also use different test systems, because we could not measure Graphical Processing Unit (GPU) usage because of the missing libraries for AMD graphic cards. It is a good expansion for our experiment to execute it on different hardware, especially on a test system with much less RAM and a weaker CPU to find out if Flutter works with the same performance or if there are significant differences.

We think there are many relevant research topics, which can profit from our experiment and its results. The research will especially help developers to understand and improve Flutter in the future.

Bibliography

[Bar21] Baron. *Cross-platform mobile frameworks used by software developers worldwide in 2019 and 2020*. 2021. URL: https://www.statista.com/statistics/869224/worldwide-software-developer-working-hours (cit. on p. 18).

[Bon18] Bonniwell. *The 5W's of MVVM*. July 2018. URL: https://www.sagitec.com/blog/the-5ws-of-mvvm (cit. on p. 30).

[Boy21] Boyter. *scc*. 2021. URL: https://github.com/boyter/scc (cit. on p. 38).

[Dar21] Dart. *Sound null safety*. 2021. URL: https://dart.dev/null-safety (cit. on p. 21).

[Dav21] David. "The New .NET Multi-platform App UI David Ortinau". In: (2021). URL: https://devblogs.microsoft.com/xamarin/the-new-net-multi-platform-app-ui-maui (cit. on p. 23).

[Dil21] Dilan. *Toyota's Flutter-based in-vehicle user experience*. May 2021. URL: https://dilan.me/toyotas-flutter-based-in-vehicle-user-experience (cit. on p. 18).

[Flu21a] Flutter. *Desktop support for Flutter*. 2021. URL: https://flutter.dev/desktop (cit. on p. 32).

[Flu21b] Flutter. *Flutter architectural overview*. 2021. URL: https://flutter.dev/docs/resources/architectural-overview (cit. on p. 19).

[Flu21c] Flutter. *Flutter on Github*. 2021. URL: https://github.com/flutter/flutter (cit. on p. 18).

[Flu21d] Flutter. "User interface". In: (2021). URL: https://flutter.dev/docs/development/ui (cit. on p. 18).

[flu21] flutter. *Introduction to declarative UI*. 2021. URL: https://flutter.dev/docs/get-started/flutter-for/declarative (cit. on p. 19).

[Geo17] D. George. *WPF Architecutre*. Mar. 2017. URL: https://docs.microsoft.com/en-us/dotnet/desktop/wpf/advanced/wpf-architecture?view=netframeworkdesktop-4.8.

[Geo18] D. Georg. *Get started (WPF)*. Jan. 2018. URL: https://docs.microsoft.com/en-us/dotnet/desktop/wpf/getting-started/?view=netframeworkdesktop-4.8 (cit. on p. 15).

[Geo21] D. George. *XAML overview (WPF .NET)*. Dec. 2021. URL: https://docs.microsoft.com/en-us/dotnet/desktop/wpf/xaml/?view=netdesktop-5.0#what-is-xaml (cit. on p. 16).

[Git21a] GitHub. *[Windows] Consider adding initial screen center position #86244*. 2021. URL: https://github.com/flutter/flutter/issues/86244 (cit. on p. 48).

[Git21b] GitHub. *Flutter Engine*. 2021. URL: https://github.com/flutter/engine (cit. on p. 18).

[Git21c] GitHub. *Introducing .NET Multi-platform App UI (MAUI)*. 2021. URL: https://github.com/dotnet/maui (cit. on p. 23).

[Git21d] GitHub. *TextField: Missing keyboard shortcuts for Windows #77020*. 2021. URL: https://github.com/flutter/flutter/issues/77020 (cit. on p. 48).

[Git21e] Github. *[Desktop] Add support for system tray icons #81644*. 2021. URL: https://github.com/flutter/flutter/issues/81644 (cit. on p. 48).

[Git21f] Github. *[Desktop] Allow "always on top" like Windows Task Manager or Android Emulator #79840*. Apr. 2021. URL: https://github.com/flutter/flutter/issues/79840 (cit. on p. 49).

[Git21g] Github. *[window_size] Allow toggling full screen #679*. 2021. URL: https://github.com/google/flutter-desktop-embedding/issues/679 (cit. on p. 48).

[Git21h] Github. *Add UWP support to plugin template #85836*. 2021. URL: https://github.com/flutter/flutter/issues/85836 (cit. on p. 48).

[Git21i] Github. "Proposal: desktop context menu fidelity". In: (2021). URL: https://github.com/flutter/flutter/issues/74255 (cit. on p. 48).

[Goo21a] Google. *Dart*. 2021. URL: https://dart.dev (cit. on p. 21).

[Goo21b] Google. *Fuchsia*. 2021. URL: https://fuchsia.dev/fuchsia-src/contribute/roadmap/2021/flutter_on_fuchsia_velocity (cit. on p. 18).

[Goo21c] Google. *menubar*. 2021. URL: https://github.com/google/flutter-desktop-embedding/tree/master/plugins/menubar (cit. on p. 50).

[Kha18] Khan. *The Pros and Cons of Progressive Web Apps*. 2018. URL: https://clutch.co/app-developers/resources/pros-cons-progressive-web-apps (cit. on p. 23).

[Lan20] R. Lander. *Announcing .NET 5.0*. Nov. 2020. URL: https://devblogs.microsoft.com/dotnet/announcing-net-5-0 (cit. on p. 17).

[Mic21a] Microsoft. *dotnet/core*. Ed. by GitHub. 2021. URL: https://github.com/dotnet/core (cit. on p. 17).

[Mic21b] Microsoft. *Nuget*. 2021. URL: https://www.nuget.org (cit. on p. 15).

[Mil20] M. Milanović. *A Brief Walk Through .Net Ecosystem*. 2020. URL: https://milan.milanovic.org/post/a-brief-walk-through-net-ecosystem (cit. on p. 17).

[Mül21] E. Müller. "Web technologies on the desktop: an early look at Flutter". Bachelor thesis. 2021. URL: https://elib.uni-stuttgart.de/handle/11682/11515 (cit. on pp. 25, 47).

[Ols20] Olsson. *A Comparison ofPerformance and LooksBetween Flutter and NativeApplications*. June 2020. URL: https://www.diva-portal.org/smash/get/diva2:1442804/FULLTEXT01.pdf (cit. on p. 25).

[Psu21] Psutil. *psutil documentation*. 2021. URL: https://psutil.readthedocs.io/en/latest (cit. on p. 37).

[Pub21a] Pub.dev. *Http 0.13.3*. 2021. URL: https://pub.dev/packages/http (cit. on p. 47).

[Pub21b] Pub.dev. *Path 1.8.0*. 2021. URL: https://pub.dev/packages/path (cit. on p. 47).

[pub21] pub.dev. *fluent_ui 2.2.1*. June 2021. URL: https://pub.dev/packages/fluent_ui (cit. on p. 50).

[Ric19] Richard. "Introducing .NET 5". In: (May 2019). URL: https://devblogs.microsoft.com/dotnet/introducing-net-5 (cit. on p. 17).

[Rie21] F. Rieseberg. *Chapter 1. What Is Electron?* 2021. URL: https://www.oreilly.com/library/view/introducing-electron/9781491996041/ch01.html (cit. on p. 23).

[SA20] Scoccia, Autili. "Web Frameworks for Desktop Apps: an Exploratory Study". In: (Oct. 2020). URL: https://dl.acm.org/doi/pdf/10.1145/3382494.3422171 (cit. on p. 25).

[Sel20] Sells. *Announcing Flutter Windows Alpha*. 2020. URL: https://medium.com/flutter/announcing-flutter-windows-alpha-33982cd0f433 (cit. on p. 19).

[Sne21] Sneddon. *First Look: Ubuntu is Working on a Brand New Installer*. Feb. 2021. URL: https://www.omgubuntu.co.uk/2021/02/ubuntu-is-working-on-a-brand-new-installer (cit. on p. 18).

[Sus15] Sushil. *Overview of Windows Presentation Foundation (WPF) Architecture*. Mar. 2015. URL: https://www.c-sharpcorner.com/UploadFile/819f33/overview-of-windows-presentation-foundation-wpf-architectu (cit. on pp. 15, 16).

[Wik19] Q. Wiki. *About Qt*. 2019. URL: https://wiki.qt.io/About_Qt (cit. on p. 23).

[Wik21a] Wikipedia. *CPU time*. June 2021. URL: https://en.wikipedia.org/wiki/CPU_time (cit. on p. 37).

[Wik21b] Wikipedia. *Dart (programming language)*. 2021. URL: https://en.wikipedia.org/wiki/Dart_(programming_language (cit. on p. 21).

[Wik21c] Wikipedia. *Windows Forms*. June 2021. URL: https://en.wikipedia.org/wiki/Windows_Forms (cit. on p. 22).

[wpf21] wpf-tutorial. *The WPF Menu control*. 2021. URL: https://wpf-tutorial.com/common-interface-controls/menu-control (cit. on p. 49).

[Wu18] Wu. "React Native vs Flutter, cross-platform mobile application frameworks". Mar. 2018. URL: https://www.theseus.fi/bitstream/handle/10024/146232/thesis.pdf?sequence=1 (cit. on p. 26).

[WWTW16] Warren, Wagner, Turn, Wenzel. "What is "managed code"?" In: (June 2016). URL: https://docs.microsoft.com/en-us/dotnet/standard/managed-code (cit. on p. 15).

If not otherwise specified, all links were last followed on Aug 2, 2021.

A Screenshots of the Reference Application

A.1 Flutter Version

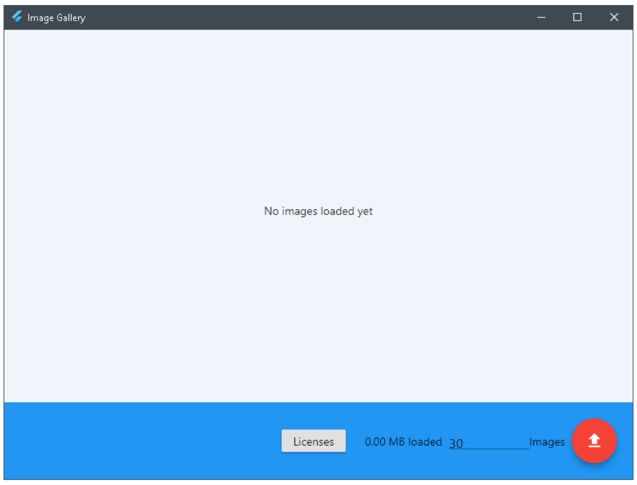


Figure A.1: Start page without loaded images

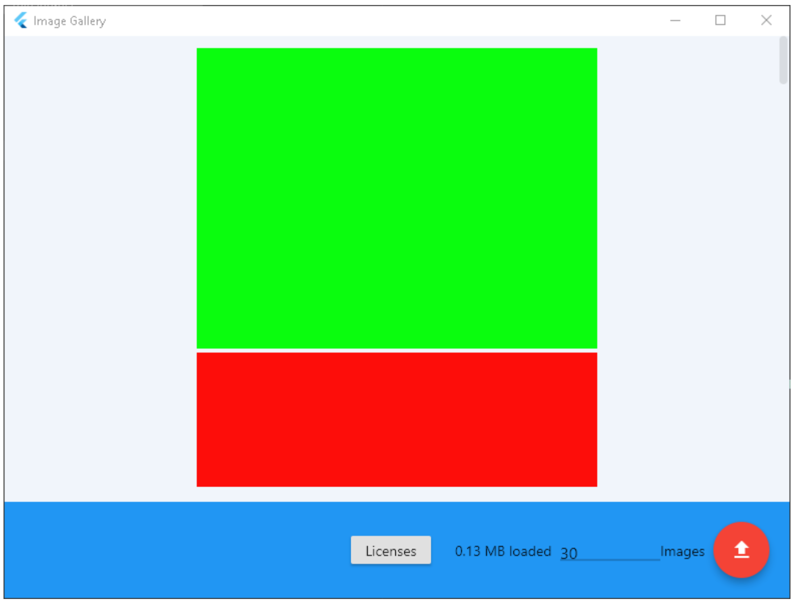


Figure A.2: Start page with loaded images

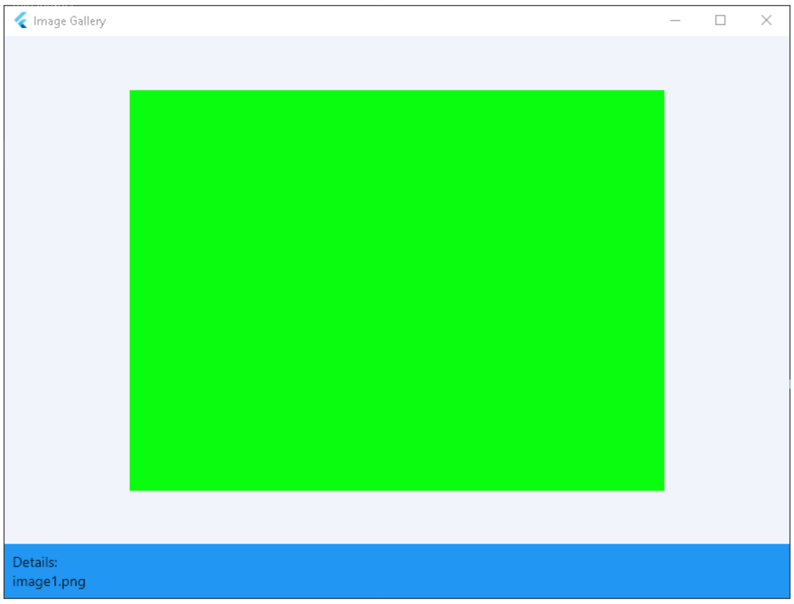


Figure A.3: Detail view

A.2 WPF Version

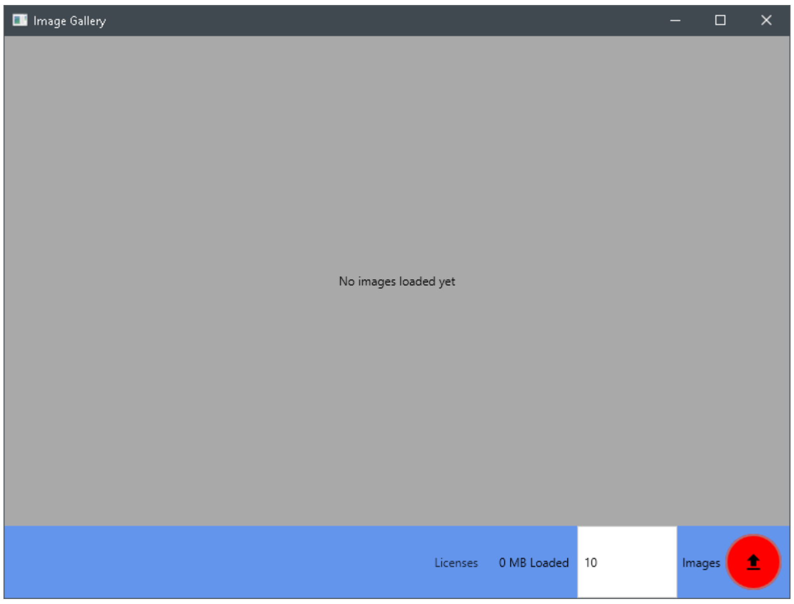


Figure A.4: Start page without loaded images

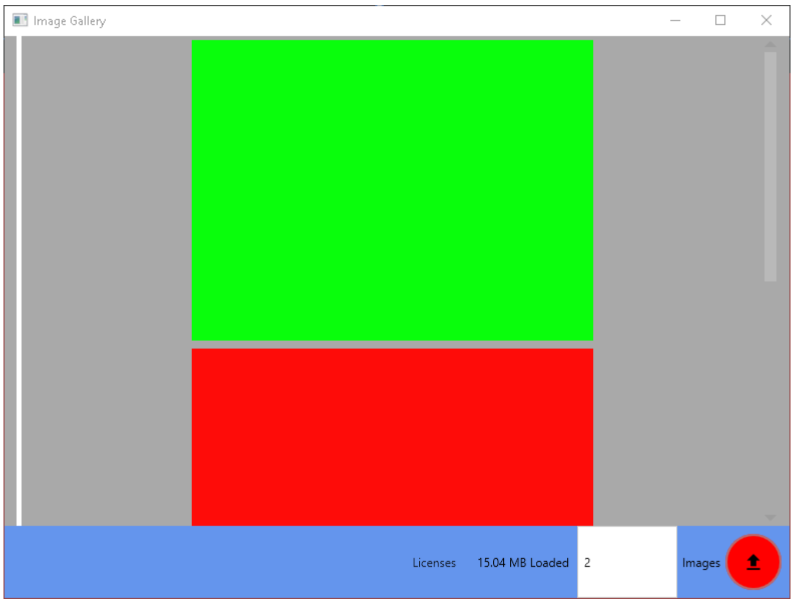
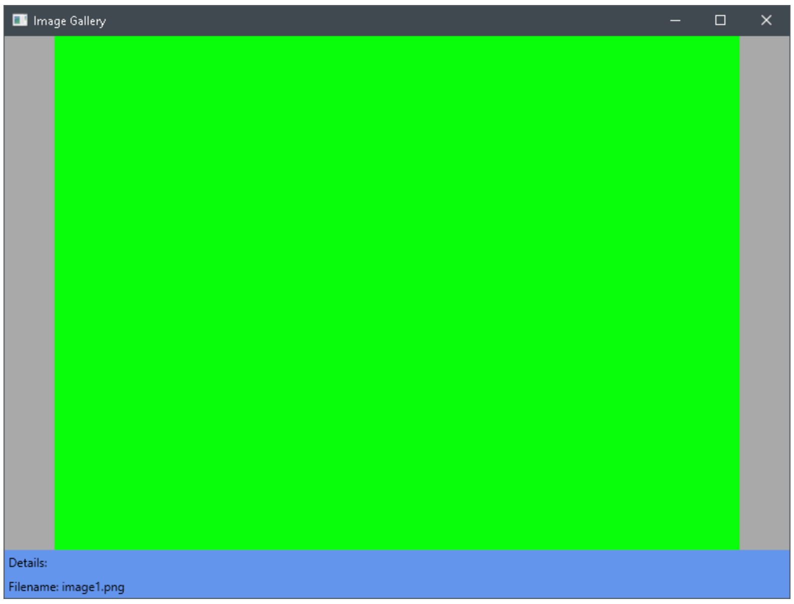


Figure A.5: Start page with loaded images



Details:

Filename: image1.png

Figure A.6: Detail view

Declaration