Sérgio Gabriel Pacheco de Aguiar

A viabilidade de Dart em desenvolvimento full stack: um estudo de caso

Dart's viability in full-stack development: a case study

Universidade de Aveiro
2021

Sérgio Gabriel Pacheco de Aguiar

A viabilidade de Dart em desenvolvimento full stack: um estudo de caso

Dart's viability in full-stack development: a case study

Dissertação realizada por parte de Sérgio Gabriel Pacheco de Aguiar no âmbito de obtenção de grau de Mestre em Engenharia de Computadores e Telemática, conferido pela Universidade de Aveiro, sob a orientação científica do Professor Doutor José Maria Amaral Fernandes, professor auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro.

Dedico este trabalho à minha mãe.

O júri / The jury

Presidente / President

Prof. Doutor Carlos Manuel Azevedo Costa
Professor Associado com Agregação da Universidade de Aveiro

Vogais / Examiners committee

Prof. Doutora Inês de Castro Dutra
Professora Auxiliar do Departamento de Ciência de Computadores da Faculdade de Ciências da Universidade do Porto

Prof. Doutor José Maria Amaral Fernandes
Professor Auxiliar da Universidade de Aveiro

Agradecimentos/ Acknowledgments

Ao meu orientador, Professor Doutor José Maria Amaral Fernandes, pela sua constante disponibilidade, explicação de tecnologias e exploração de ideias que facilitaram o estudo em questão, bem como pela ajuda com organização temporal e flexibilidade em termos de tema de dissertação, muitíssimo obrigado.

Aos meus colegas, Fábio Daniel Ribeiro Alves e Ricardo Rodrigues Azevedo, por me acompanharem no meu percurso e estarem sempre prontos a ajudar quando necessário.

Aos meus restantes colegas, que, mesmo que a um menor grau, também me acompanharam e suportaram durante os passados anos.

À minha família, em particular à minha mãe, Sandra Marisa Pereira Pacheco, que tornaram possível o meu ingresso nesta universidade e me suportaram durante os meus estudos.

Palavras-Chave Dart, Flutter, Desenvolvimento full stack; Spring Boot, Prova de conceito.

Resumo

Em 2012, Google lançou a linguagem Dart que, mais recentemente, devido ao Flutter, recebeu um impulso em popularidade e é muitas vezes referida como uma linguagem / ecossistema full stack adequado para o desenvolvimento de soluções front end e back end. No entanto, além do Flutter para dispositivos móveis, o uso de Dart ainda é muito baixo quando se trata de desenvolver soluções de nível corporativo.

Nesta dissertação, tentamos investigar a adequação do uso de Dart para desenvolver uma solução full stack com foco especial no seu suporte de back end. Com isso em mente, foi estabelecido um cenário típico envolvendo um front end móvel e um compatível com web, em que ambos comunicam com um servidor back end por meio de um endpoint REST. Para comparação de desempenho, implementamos um servidor back end equivalente desenvolvido usando Spring Boot, uma solução popular baseada em Java, que foi usada como referência.

O resultado principal foi que um sistema full stack pode ser desenvolvido com apenas um ecossistema Dart / Flutter e, no nosso cenário, o desempenho desse sistema ultrapassou o Spring Boot. Do ponto de vista do programador, soluções assíncronas incorporadas Dart prontas para uso (por exemplo, streams, Futures, etc.) são claramente uma melhoria em relação a mecanismos semelhantes em Java / Spring Boot devido a evitarem soluções Java típicas, nomeadamente configurações assíncronas e anotações. No entanto, apesar de alguns projetos interessantes surgirem, ao excluir os pacotes / recursos desenvolvidos pela própria Google, a maioria dos pacotes desenvolvidos por terceiros usam dependências desatualizadas devido a problemas de compatibilidade ou foram abandonados inteiramente - isso teve um impacto durante a fase de desenvolvimento, pois levou a restrições não planeadas na escolha de pacotes e / ou frameworks usados.

Keywords Dart, Flutter, Full-stack development; Spring Boot, Proof of concept.

Abstract

In 2012, Google released the Dart language which, more recently, due to Flutter, has received a boost in popularity and is being often referred to as a full-stack language / ecosystem suitable for developing front-end and back-end solutions. However, aside from Flutter for mobile, Dart usage is still quite low when it comes to developing enterprise level solutions.

In this dissertation, we tried to investigate the adequacy of using Dart to develop a full-stack solution with special focus on its back-end support. With that in mind, a typical scenario involving both a mobile and a web-supported front end, where both communicate with a back-end server via a REST endpoint, was established. For performance comparison, we deployed an equivalent back-end server developed using Spring Boot, a popular Java-based solution, which was used as reference.

The main result was that a full-stack system can be developed with just a Dart / Flutter ecosystem and, in our scenario, this system's performance surpassed Spring Boot's. From a developer's perspective, off-the-shelf Dart embedded asynchronous solutions (e.g., streams, Futures, etc.) are clearly an improvement over similar mechanisms in Java / Spring Boot due to avoiding typical Java solutions, namely asynchronous configurations, and annotations. However, despite some interesting projects arising, when excluding Google's own developed packages/resources, most third-party packages are either using out-of-date dependencies due to compatibility issues or have been abandoned entirely – this had an impact during the development stage as it led to unplanned constraints when choosing packages and / or frameworks used.

Contents

List of figures

List of tables

List of abbreviations and acronyms

API – Application Programming Interface

CCM – Crypto Currency Manager

CD – Continuous Deployment

CI – Continuous Integration

CORS – Cross-Origin Resource Sharing

DB – Database

DBMS – Database Management System

HTML – Hypertext Markup Language

HTTP – Hypertext Transfer Protocol

IDE – Integrated Development Environment

JDBC – Java Database Connectivity

ORM – Object-Relational Mapping

POM – Project Object Model

REST – Representational State Transfer

SW – Software

TSDB – Time Series Database

UI – User Interface

URI – Uniform Resource Identifier

VM – Virtual Machine

XML – Extensible Markup Language

1 Introduction

Modern services all tend to follow a similar system architecture: a front end comprising either a mobile application, a website or both; and a back-end server, which can be distributed, that handles the requests sent by the front end.

When developing a full-stack system, there are multiple options, namely languages and frameworks, to pick from. Dart [1] is a language option created by Google in 2011 and then released in 2012 [2]. It has been designed to create clients/user interfaces (UI), such as those present in web and mobile applications. Despite not being its original purpose, it also provides developers with means to build servers as observed from the variety of frameworks developed for that exact purpose.

Nowadays, Dart has had an increase in visibility due to being the main language adopted for Flutter [3], an open-source UI software (SW) development kit created by Google in 2017 and then released in 2018 [4]. Flutter is used to develop cross platform applications, such as Android, iOS, and Web ones, from the same codebase. This aspect of having a shared codebase has proven to be so appealing that many companies have started investing into it by changing their projects into Flutter-built ones. Examples of companies using Flutter are BMW and Tencent [5].

The success of Flutter in front-end development [6] and the added support for Dart back-end development led to a natural assumption that Dart can be used to develop an entire full-stack system.

However, regardless of, in theory, Dart being usable as the main, if not only, language to develop an entire full-stack system, there is little evidence backing it as a reasonable option in real world situations. Developers tend to gravitate towards other options, such as Java's [7] Spring Boot [8], that have demonstrated immense community approval and support, as well as countless successful project releases.

At same time, many Dart frameworks exist (e.g. Aqueduct [9] and Angel [10]), which seem to support Dart as a viable alternative but still lack clear proof of concepts/systems to support it.

1.1 Objective and Scope of the study

It was hypothesized that Dart is powerful enough of a language to be able to create competent back-end servers, especially given the number of available frameworks, even if performance-wise it ends up being worse than current state-of-the-art solutions. Given Dart's focus on asynchrony, operations revolving around this concept were hypothesized to be handled efficiently.

In this context, questions as the ones below are natural:

- Is Dart a suitable language to create a back-end server in?
- Compared to state of the art, how does Dart fare when being used to create back ends?

- Is Dart a viable alternative to current state-of-the-art solutions?

The main objective of this dissertation is to explore if Dart is already a viable option to develop a full-stack solution comprising the presentation/front-end layer and back-end layer.

To achieve the results, a set of objectives were also set:

- Explore existing Dart solutions to deploy a system – a cryptocurrency exchange service to handle requests incoming from front-end applications was selected.
- Compare the Dart system with a state-of-the-art solution based on Java's Spring Boot, using the same system specification.

The comparison between Dart and Spring Boot solutions will focus on the actual performance of the system and on a subjective critical assessment from the perspective of the developer. The performance assessment will be based on monitoring both back-end servers using timing logs; and the subjective from the developer perspective, namely system development considerations and opinions regarding the development process.

1.2 Thesis structure

In the "Full-stack development in Dart/Flutter ecosystems" section, an overview of Dart and Flutter's main aspects and advantages will be covered, as well as information on some existing frameworks.

In the "Crypto Currency Manager (CCM) scenario" section, the scenario conceived to model both developed back ends after will be explained. This includes a generic view of the back-end server's architecture, a listing of the exposed REST (Representational State Transfer) interfaces and reasoning behind the choice made for the reference system.

In the "CCM (Crypto Currency Manager) implementations" section, the logic behind both CCM implementations will be explained, such as regarding used technologies and packages/libraries. An overview of the remaining systems used will also be done, going over the front-end applications and the reverse proxy.

In the "The logging system" section, an overview of the logging solution developed for the study will be given, including the logged data destination, why it was chosen and how log data reaches it.

In the "Result gathering and discussion" section, the environment conditions used during metric collection, how they were collected, and the technology used to treat them will be explained. Additionally, the obtained results will be presented, described, and discussed.

In the “Conclusions” section, the questions initially posed will be answered, the hypotheses weighed against the previously obtained results and thoughts on future steps presented.

2 Full-stack development in Dart/Flutter ecosystems

Dart [1] was created by Google in 2011 and then released in 2012 [2] with the purpose of creating best-in-class clients/user interfaces (UIs), such as those present in web and mobile applications. It's a language that was optimized for UI creation, possesses syntax that is familiar and easy to understand due to its similarity to other known languages, and bolsters features such as hot reload. It can run on any platform due to its virtual machine architecture, the Dart Virtual Machine, through native platform virtual machine implementations, also supporting the usage of ahead-of-time compilers to allow for operating systems such as Android and iOS. Compiled Dart code is also browser friendly as it provides direct mapping to JavaScript.

Currently, Dart is capable of being used for both front-end and back-end development. One of the main objectives of this work is to assert if Dart is already able to provide a good full-stack development option.

Full-stack development is the act of developing both the front end and back end of a system. Front-end development revolves around creating user-friendly interfaces that allow the client to use the available services. Back-end development revolves around creating servers to manage said services by receiving and handling client requests. Back-end servers do so by opening communication channels usable by front-end applications and storing data in persistent storage, such as databases.

2.1 Asynchronous code

Dart was built upon the idea of having a language specialized in asynchronous programming, something that is highly needed when dealing with UI, Dart's focus. So, Dart native libraries have access to a vast amount of additional asynchronous programming resources that can be useful when also addressing back-end service implementation.

Dart's asynchronous aspect goes even a step further. Rather than the typical usage of shared-memory threads running concurrently, Dart employs isolates [11], a light process-like execution unit with its own memory heap, to run its code - it uses an isolate-based concurrency approach. This makes it so all cores are still taken advantage of yet simplifies the code and eliminates a great possibility of errors, due to no isolate being able to alter another's state.

Code written using Dart, no matter the platform, will always be taking advantage of Dart's strengths as an asynchronous programming language, simply because these aspects are native to it. Dart also provided the concepts of Stream [12] and Future [13] from its inception unlike other languages such as Java where they were overlaid. In Dart, two keywords as simple as "async" and "await" ensure a clear asynchronous model with system-wide native support. There is no need for

manual handling of threads due to its isolate-based concurrency principles nor for depending on run-time server containers such as Tomcat.

Example: synchronous functions

```
String createOrderMessage() {
  var order = fetchUserOrder();
  return 'Your order is: $order';
}

Future<String> fetchUserOrder() =>
    // Imagine that this function is
    // more complex and slow.
    Future.delayed(
      const Duration(seconds: 2),
      () => 'Large Latte',
    );

void main() {
  print('Fetching user order...');
  print(createOrderMessage());
}
```

```
Fetching user order...
Your order is: Instance of '_Future<String>'
```

Example: asynchronous functions

```
Future<String> createOrderMessage() async {
  var order = await fetchUserOrder();
  return 'Your order is: $order';
}

Future<String> fetchUserOrder() =>
    // Imagine that this function is
    // more complex and slow.
    Future.delayed(
      const Duration(seconds: 2),
      () => 'Large Latte',
    );

Future<void> main() async {
  print('Fetching user order...');
  print(await createOrderMessage());
}
```

```
Fetching user order...
Your order is: Large Latte
```

Figure 1 – Code: synchronous versus asynchronous function comparison [13].

In Figure 1, the same basic program to simulate fetching a user's drink order is represented in a synchronous and asynchronous version.

In the synchronous version, the initial output is the hard-coded print on the main() function's first line. It is then followed by the print resulting from the second line which returns the string conversion function, toString(), relative to a Future of type String. This is the case due to the createOrderMessage() function being called inside the print and returning a String with a variable resulting from the fetchUserOrder() function. This fetchUserOrder() has a return type of Future<String>, so the final output makes sense from a programming standpoint. The issue here is the fact that the returned value is always the same and not the intended one, the user's order, which results from the Future.delayed() call. Since the code is synchronous, the createOrderMessage() call has its lines of code executed in order and, as soon as the Future.delayed() call is performed, starting the two second delay prior to the correct output being returned, the function is exited. Due to this, the intended value will never be obtained in the synchronous version because the result print will happen before the Future can complete.

In the asynchronous version, a few changes were made: the main() and createOrderMessage() functions are now asynchronous, returning a Future instance rather than their previous void and String types, respectively, and are signed with the async keyword. Additionally, they both now await on a function call (createOrderMessage() in the main() function's case, and fetchUserOrder() in the createOrderMessage()'s).

Signing a function with the "async" keyword defines a function as asynchronous, allowing for the use of "await" keywords, and changing the return type to a Future of the previous type, such as from a String to a Future<String>, means that the function will return the given type, when the Future eventually completes. In the case of the given example, the synchronous version had one main issue, which was the fact that the fetchUserOrder() function was returned from before the Future.delayed() call had completed. This is now solved by using the "await" keyword on that function call, meaning that the function will wait for the Future to complete before moving on with the code, guaranteeing the existence of the desired output.

These keywords (and the Future type) are the backbone of Dart asynchronous programming and what allows it to be so powerful when compared to other languages.

```
Future<int> sumStream(Stream<int> stream) async {
  var sum = 0;
  await for (var value in stream) {
    sum += value;
  }
  return sum;
}
```

Figure 2 – Code: reading stream values via an "await for" iteration [12].

The final main aspect of Dart's asynchronous properties is the existence of streams. A stream is a sequence of asynchronous events and can be iterated over to obtain its values. Contrary to other structures, where something would be given when asked for, streams inform when new values exist to be consumed.

Values sent to a stream can be iterated over by using the "asynchronous for loop", commonly known and referred to as "async for", as shown in Figure 2. Due to requiring the usage of the "await" keyword, its function must be signed with the "async" keyword. An "async for" iterates over all existing values in a stream and, once finished, waits for either a new value to be added or for the stream to be closed. As for the code present in Figure 2, the "await for" iterates over each value present in the Stream<int> object passed as an argument, cumulatively adding them to the sum

variable. Once every existing value from the stream object is consumed, the loop will pause and wait for either a new value or the stream's closure before moving onto the following line of code, the sum's return statement.

```
Stream<int> countStream(int to) async* {
  for (int i = 1; i <= to; i++) {
    yield i;
  }
}
```

Figure 3 – Code: generating a stream using an async* function [12].

For values to exist in a stream though, they must be added in some way. In Figure 3, the most basic and common way to do so is shown: "async*" functions. Just as signing a function with the "async" keyword, signing with "async*" turns the function into an asynchronous one, allowing for the usage of, for example, "await" keywords. The main difference between these two is the fact that while an "async" function returns, eventually, a Future, an "async*" function returns a Stream, which can effectively return multiple Futures over time. Values are returned via using the "yield" keyword rather than the conventional "return" one.

In the example shown in Figure 3, a "for" loop iterates a number of times equal to the value passed through the "to" argument. If this variable were to have the integer value 3 passed onto it, the loop would execute for the "i" variable values of 1, 2 and 3, given its increment at the end of each loop iteration. Within the loop's body, only a "yield" instruction is present. This instruction returns the current "i" value, meaning that, throughout the loop's entirety, the values 1, 2 and 3 would be yielded back, given the previously stated value of 3 for the "to" argument. Given that the function then finishes so does the Stream close making it so any iterating over this function's result comes to a stop after receiving the three yielded values, rather than waiting for a new one.

2.2 Null Safety

Null Safety [14] is the guarantee that, by default, all objects in an object-oriented language will have a non-null reference. Under some circumstances, a developer may desire for an object to have a null reference, in which case can be specified.

```dart
var i = 42; // Inferred to be an int.
String name = getFileName();
final b = Foo();
int? aNullableInt = null;
```

Figure 4 – Code: three non-nullable variables and one nullable variable in Null Safe Dart [14].

In Figure 4, some Dart variables are shown. The first three cannot be null when Null Safety has been opted into, while the fourth can. This is due to the "aNullableInt" variable having "?" added to its declaration.

Null safety is important due to its ability to reduce runtime exceptions which might be caused by unexpected null references during execution. Dart, being a statically typed language, allows for code errors to be found in compile-time which, when combined with Null Safety, allows for issues of this nature to be found before deployment.

```dart
bool isEmpty(String string) => string.length == 0;

main() {
  isEmpty(null);
}
```

Figure 5 – Code: how a runtime exception may occur in Null unsafe Dart [15].

In Figure 5, a function to verify whether a String object is empty or not, via checking if its length is equal to 0, is present. This function is then called in the main() function, which passes "null" into the String argument. In Dart without Null Safety, this does not generate a compiler error meaning the "NoSuchMethodError" exception is only thrown when running the code, which is an issue when developing UIs, one of Dart's main focuses. However, with Null Safety, the issue is made readily known, especially due to the power of current integrated development environments, or IDEs, due to the String variable not being able to have a null reference.

Prior to Null Safety, Null was treated as a subtype of all other types. This, however, caused a lot of issues such as the one stated previously. Despite it being a subtype of String, for example, it has access to none of its methods or attributes, such as length. Due to this, it could be used in operations such as additions or subtractions between int variables without any compiling errors being found, yet then causing an exception during runtime. To fix this, Null was removed from being a subtype of all other types after Null Safety was introduced, as shown in Figure 6.

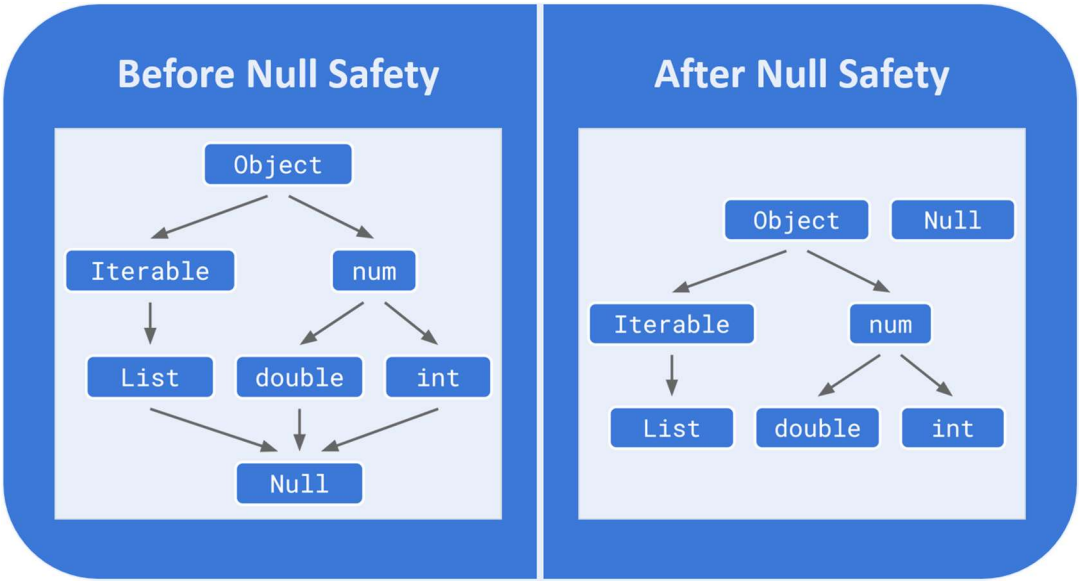

Figure 6 – Dart primitive type hierarchy before and after Null Safety [15].

On the other hand, due to Null still being important when it comes to programming, nullable types, such as String?, were added as supertypes to other types. When specifying that a given function takes a String? as an argument, both Null and a String can be passed, and the compiler will not raise any error notice. This allows for the creation of methods with optional parameters, requiring them to either be nullable or have a default value. Additionally, with the addition of Null Safety, Dart also added the ability to specify mandatory named parameters by using the "required" keyword, be it for nullable or non-nullable types.

2.3 Flutter: Dart's front end of choice

In terms of front end, Dart has Flutter as its main option. Flutter [3] is an open-source UI software (SW) development kit, created by Google in 2017 and then released in 2018 [4], and is used to develop cross platform applications, such as Android, iOS, and Web ones, from the same codebase. Flutter's widgets [16] abstract how each native system handles critical UI differences, such as scrolling and navigation, providing developers with a great option for developing solutions for multiple platforms using the same codebase.

Flutter's widgets take inspiration from React [17], a JavaScript [18] library for building UIs, in the way that they are built using structures that describe what they should look like, the widgets. When a widget has its internal state changed, the description used to build how it looks, it is rebuilt and compared to the previous iteration to efficiently conduct the minimal changes required on the underlying render tree. In Flutter, widget code (UI or not) integrates in the same overall code to achieve both UI and behavioural descriptions using Dart language constructs to avoid the need for a

separate declarative layout language such as XML used, for instance, when developing native Android applications.

```
FutureBuilder(
  future: http.get('http://awesome.data'),
    builder: (context, snapshot) {
    if (snapshot.connectionState ==
        ConnectionState.done) {
    return AwesomeData(snapshot.data);
  } else {
    return CircularProgressIndicator();
 }
}
```

Figure 7 – Code: typical structure for a FutureBuilder widget (without error checking) [19].

This allows Flutter widgets to have access to all of Dart's asynchronous capabilities, however this does not mean it does not build onto it. Flutter features two main async widgets: the FutureBuilder [20] and the StreamBuilder [21].

The FutureBuilder is a widget that builds itself based on the last snapshot of interaction with a Future, where a snapshot pertains to the data existing at a specific instance in time. It is useful because a UI needs to be able to display placeholders when waiting for specific data, resulting from a Future, to be usable.

In Figure 7, the main logic employed when working with FutureBuilders is shown. A Future to monitor is passed onto the "future" argument, and an AsyncWidgetBuilder [22] is passed onto the "builder" argument. In the case of the given example, the Future passed onto the argument was an HTTP (Hypertext Transfer Protocol) GET request to the address "http://awesome.data" and the AsyncWidgetBuilder was one with logic to handle both a state of completion and of lack thereof. Whenever the connection's state is checked for the relevant snapshot, if it contains the "done" value then it should also be checked for errors via the "snapshot.hasError" bool attribute. If no error occurred, then the data present in "snapshot.data" is the intended result for the given Future. If the connection's state has the "waiting" value then the desired data is not yet available, hence being unable to show it to the user via the interface. Instead of it, something like a loading screen can be

used to not only fill in the vacant space but also inform the user that the application is handling data obtaining.

```
StreamBuilder<int>(
  stream: _myStream,
  builder: _myBuilderFunction,
);
```

Figure 8 – Code: simplified overview of a StreamBuilder widget implementation structure [23].

The StreamBuilder shares the same previously explained logic as the FutureBuilder, however snapshots are based on interaction with a Stream. As shown in Figure 8, a stream can be passed to the "stream" argument, and an AsyncWidgetBuilder to the "builder" parameter. Just as with the FutureBuilder, the snapshot can be accessed to check the connection state and the existence of data in the snapshot, as well as errors. Additionally, the "initialData" argument can be used to set the widget's initial data. Both the StreamBuilder and the FutureBuilder widgets follow the builder creational design pattern [24].

```
class ColorInfo extends InheritedWidget {
  ColorInfo({this.colors, Widget myChild})
      : super(child: myChild);
  final List<Color> colors;
  bool updateShouldNotify(
      ColorInfo oldWidget) =>
          oldWidget.colors != colors;
  static ColorInfo of(BuildContext context) =>
      context.inheritFromWidgetOfExactType(
          ColorInfo) as Storage;
}
```

Figure 9 – Code: extending InheritedWidget to propagate colour information [25].

Aside from the async widgets enunciated, Flutter is composed of many others that contribute towards creating fluid UIs, but some could also be used to assist in the creation of Flutter-based back-end servers. The main example of this is the InheritedWidget [26].

The InheritedWidget is a widget that allows for data to be propagated down a widget tree. In Figure 9, an example on how to create a class to propagate colour information is shown. In this example, there are three noteworthy aspects to it: the class extends InheritedWidget (to obtain all the base characteristics that make up an InheritedWidget), it declares a list variable named "colors" (the data to propagate), and it implements both the updateShouldNotify() and of() methods.

The updateShouldNotify() method is used to notify widgets that depend on the propagated data if they should be redrawn when the data suffers changes. In the case of the given example, if the "colors" variable is changed with new colour information, then the widgets must be redrawn, due to being different than the previous one.

The of() method allows for the child widgets to access the propagated data easily and, when used, causes any child widget consuming its data to be rebuilt when the InheritedWidget itself suffers a relevant change in state.

```
Widget build(BuildContext context) {
   var colors = ColorInfo.of(context).colors;
   // Do something with colors
 }
```

Figure 10 – Code: accessing data propagated by the InheritedWidget [25].

In Figure 10, an example on how to use the of() method to obtain the propagated data in a child widget is given. By calling the method with the current context as the argument, the nearest instance of the ColorInfo InheritedWidget is obtained, thus allowing for the "colors" attribute to be accessed.

Due to its ability to propagate data throughout an entire widget tree, it makes the InheritedWidget be universally used when it comes to propagating services, such as database access. For larger projects though, it can increase code complexity due to the amount of bloat-code, thus recommending the use of state management packages such as provider [27] or scoped_model [28].

2.4 Dart back-end frameworks

In SW development, frameworks are higher level technological support structures that aim to make tasks, such as HTTP request handling through REST (Representational State Transfer) API

(Application Programming Interface) [29] support, more convenient and less of a nuisance for developers. Back-end frameworks are those located server side which assist in servicing requests from client applications.

In Dart's case, several web frameworks exist: the two most popular ones, Aqueduct [9] (now Liquidart [30]) and Angel [10], and some less commonly used ones, such as Jaguar [31], Start [32], Shelf [33], Vane [34], and Alfred [35], among others that may exist yet were not found. Of these, Aqueduct, Angel and Jaguar stand out as the most complete ones in terms of supporting libraries, while the remaining ones offer more simplified approaches, often just focusing on HTTP request handling.

The study's focus was mainly on Aqueduct and Angel, the most popular and mature Dart frameworks. Aqueduct is a Dart HTTP web server framework for building REST applications. It features a statically typed ORM (Object-Relational Mapping) and database migration, OpenAPI 3 [36] integration, integrated testing libraries and an OAuth 2.0 [37] server. Not only does it provide a lot of solutions akin to current state of the art in the back-end server area, but it also has simple to understand and use syntax. Despite its popularity, the Aqueduct development team has announced they would be halting development on the project. Since then, a group of developers has taken over from where Aqueduct left off, resuming development under the Liquidart framework name.

Angel is a production-ready back-end framework for Dart. It is a simple framework designed with extensibility in mind so that newly needed features can easily be developed and over three dozen packages to extend it already exist, to allow for the usage of important tools, such as ORMs or authentication. It is certainly a lot more lightweight than Aqueduct was but that can also bring advantages, depending on the project.

Among the several available solutions, such as ASP.NET Core [38] and Express.js [39], Java's Spring Boot [8] was chosen as a reference language-framework combination to run comparisons on. This was because Java is both still a popular language and like Dart in a variety of aspects, such as syntax and VM existence. By comparing a Dart-based implementation to a popular and widely used state-of-the-art solution, a better grasp of Dart's capabilities could be obtained.

2.4.1 Current situation

Spring Boot is a java-based open-source framework for developing back-end servers, built on the principle of trying to minimize implementation dependencies, whose compiled code can run on any platform, given the existence of the Java VM. Despite, according to polls run by StackOverflow [40], Java dropping from 45.3% popularity in 2018 [41] to 41.1% in 2019 [42] and to 40.2% in 2020 [43], Spring [44] remains relevant by having its "Loved" rate above 50% [43]. With Spring Boot being an

extension of the Spring framework, which handles a lot of configurations and reduces overall boilerplate code, it is one of the main reasons keeping Spring competitive.

Within the Java development community, specifically, Spring Boot is the most used application framework, according to JRebel's 2021 Java Developer Productivity Report [45], standing at 62%. Despite this being lower than the previous year's 83% [46], it still stands as the clear pick for a Java-based reference system.

Spring Boot makes use of dependency injection, which is the usage of one object within another in a way akin to a service, simplified integration of other frameworks and tools, such as JPA [47]/Hibernate [48] ORMs (Object-Relational Mappings), which are used to convert data between incompatible type systems by creating virtual databases, and Spring Boot MVC [49], a web framework based off the Model-View-Controller SW design pattern [50].

It has the advantage of being easy to understand and use, given little knowledge of its code concepts, such as dependency injection, aspect-oriented programming, and proxies. The fact that it is well documented and handles a lot of the configurations itself also helps to increase its ease of use.

Despite being so popular and having a lot of advantages, it has some aspects that are not as pleasant to a lot of developers: developers with less experience, or even completely new, can have a tough time using it due to a lot of crucial details pertaining its code being omitted, such as how dependency injection actually works; and the fact that it relies too much on HTML solutions for its front end, an aspect that turns a lot of developers away from it, and, often, towards Flutter.

Even though Java is a popular language, a lot of the community has, over time, increasingly fled from it. Alternatives to Spring Boot should be studied in the hopes of finding a new one to consider as state of the art.

Not every back-end developer enjoys working with HTML (Hypertext Markup Language). Dart, with the assistance of the Flutter development kit, provides them with a more fluid and code-like way to design UI, increasing productivity.

Google has been investing in Dart as of late by moving a lot of resources into Flutter development with Dart even being used as one of the languages to develop their new mobile operating system, Fuchsia [51]. With Dart growing in importance soon it becomes imperative to begin evaluating what possibilities are brought to the table, as well as to inform those who are still in the dark.

2.4.2 HTTP request handling

It is crucial to be able to support HTTP request handling and both Aqueduct and Angel provide an API for easing the HTTP endpoints declaration and deployment.

In Aqueduct, controllers handle HTTP requests, and the most important type of controller is the router. Routers evaluate request paths and send them to other controllers to handle the actual request

operations, with the type of request being available by adding an argument of type “Operation” to a method’s signature.

```
@override
Controller get entryPoint {
  final router = Router();

  router
    .route('/heroes')
    .link(() => HeroesController());

  router
    .route('/example')
    .linkFunction((request) async {
      return Response.ok({'key': 'value'});
    });

  return router;
}
```

Figure 11 – Code: Aqueduct's basic routing using routers and controllers [52].

In Figure 11, the router object has two different routes added to it: “/heroes”, which forwards the request towards the HeroesController controller, and “/example”, which links a function controller to return a 200 response, represented by the Response.ok() method.

```
class HeroesController extends Controller {
  final _heroes = [
    {'id': 11, 'name': 'Mr. Nice'},
    {'id': 12, 'name': 'Narco'},
    {'id': 13, 'name': 'Bombasto'},
    {'id': 14, 'name': 'Celeritas'},
    {'id': 15, 'name': 'Magneta'},
  ];

  @override
  Future<RequestOrResponse> handle(Request request) async {
    return Response.ok(_heroes);
  }
}
```

Figure 12 – Code: extending Controller in Aqueduct to return a list of heroes [52].

Once a request with path "/heroes" reaches the router, it gets forwarded to the HeroesController that is shown in Figure 12. This class extends Controller, has a list of hero identifiers and names, and an instantiation of the Controller class's handle() method, where a 200 response is returned alongside the present list of heroes.

Despite other frameworks heavily relying on annotations, Aqueduct takes a different approach that still manages to remain simple and easy to build on.

In Angel, controllers can also be used to handle HTTP requests. Just as in Aqueduct, a class that extends a Controller class is created. In the case of Angel, as shown in Figure 13, the class is annotated with @Expose to define the route requests must take to reach it. This annotation is then also used in class methods to further specify the route, such as "/todos/:id" to have the request handled by the getTodo() method or "/todos/login" to have it handled by the login() one.

```
@Expose("/todos")
class TodoController extends Controller {

  @Expose("/:id")
  getTodo(id) async {
    return await someAsyncAction();
  }

  @Expose("/login")
  login() => auth.authenticate('google');
}

main() async {
  Angel app = Angel(reflector: MirrorsReflector());
  await app.configure(TodoController().configureServer);
}
```

Figure 13 – Code: Angel's HTTP routing using controller logic [53].

To be able to use annotations or any other form of reflection, the server's instantiation, using the Angel class, must have a suitable reflector passed as an argument, such as the MirrorsReflector present in the example or a static reflector variant. Finally, controllers must be added to the server's configuration via the Angel.configure() method.

Despite Angel's state-of-the-art-like options, its controllers, being more than adequate to fulfil current HTTP request handling needs, Angel allows its users to handle requests in a separate way: its basic routing interface.

```
app.get('/todos/:id', (req, res) async => {'id': req.params['id']});
```

Figure 14 – Code: Angel's basic routing [54].

Angel's basic routing is performed by adding routes using the AngelApp.addRoute() method. This method, however, has four different wrappers, get(), post(), patch() and delete(), that can be used to not have to specify the HTTP request method. Rather than specifying a route with, for example, "app.addRoute('get', '/route/path', requestHandler)", the same can be achieved via "app.get('/route/path', requestHandler)".

The requestHandler mentioned above is an asynchronous method with two arguments: a RequestContext, with data such as headers and body relative to the request, and a ResponseContext, with the data being returned in the response once done. In Figure 14, an example of a HTTP GET request using Angel's basic routing interface is shown. In the specified path, the "id" is prefaced with ":" which makes "id" be a path parameter, allowing for different resources to be requested using the same path specification. If a GET request were to be performed on the path "/todos/1", the returned body would be a key-value pair with "id" as the key and 1 as the value, given the requestHandler's access to req.params to obtain the value passed onto the "id" parameter.

```
@RestController
public class HelloController {

	@GetMapping("/")
	public String index() {
		return "Greetings from Spring Boot!";
	}

}
```

Figure 15 – Code: Spring Boot's HTTP request handling using controller logic [55].

In Figure 15, a simple Spring Boot controller is shown. Unlike seen in Aqueduct and Angel, created controllers do not extend a Controller class nor are they referenced in code by the programmer in any way to be configured. Instead, the @RestController annotation marks the class as a controller. In a comparable way to Angel, controller methods are annotated to set the path requests must take to reach the relevant request-handling method. While in Angel, the @Expose annotation has the type of HTTP method passed through the "method" argument and is the only existing request handling annotation, Spring Boot allows for the usage of different ones for different purposes, such as @GetMapping for HTTP GET requests or @POSTMapping for HTTP POST requests. These derive

from the @RequestMapping annotation which, as in Angel, allows for the HTTP method to be specified via a "method" argument.

```
@SpringBootApplication
public class Application {

	public static void main(String[] args) {
		SpringApplication.run(Application.class, args);
	}
}
```

Figure 16 – Code: Spring Boot's basic application class [55].

As previously stated, no additional setup beyond annotation usage is required to use a controller in Spring Boot. As shown in Figure 16, the only requirement is to create a class and annotate it with @SpringBootApplication. This handles some basic configuration for the server and, once the SpringApplication.run() method is called, the server starts running.

Finally, to improve learnability and maintenance, it is important to document every portion of a developed system and API endpoints are no exception. The current state-of-the-art solution for API endpoint documentation is the OpenAPI specification, or OAS, previously known as Swagger specification, up to 2015 [56]. The OAS defines a language-agnostic interface to use as a standard for both creating and accessing RESTful APIs. Due to being language-agnostic, a single API specification can be used to create APIs with the same interface characteristics in different languages, which led to the appearance of generation tools, such as the Swagger toolset [57], to automatically generate RESTful API code based off an inputted specification. Additionally, a specification can be obtained from existing APIs to best describe them, which serves as documentation.

2.4.3 ORM support

Object-relational mapping [58], or ORM, is a programming approach that bridges object-oriented data and relational database engines, such as MySQL [59]. It allows for data to be stored in a database's tables through the usage of a developer-focused API, following a previously specified object-to-table mapping, while keeping its properties in such a way that objects with the same data can be created by obtaining data from the table.

ORMs allow for developers to abstract themselves from the database management aspect of the system and focus on the data itself as if it were stored in a code variable, forming a virtual database structure. Data is managed through the implementation of models, classes which get mapped to

tables, where each attribute corresponds to one of the table's columns. Due, in part, to this advantage when it comes to development facilitation, back-end frameworks often incorporate it in some way.

```
@Table(name: "ArticleTable")
class _Article {
  @primaryKey
  int id;

  String contents;

  @Column(indexed: true)
  DateTime publishedDate;
}
```

Figure 17 – Code: defining a database data model in Aqueduct [60].

In Aqueduct, the available ORM uses PostgreSQL [61], requiring the developer to maintain a local instance of the database. Once basic setup has been completed, data modelling can be done, and annotations are used.

The main annotation in Aqueduct is @Column, allowing for each attribute, a table column, to have a variety of properties commonly found in relation databases set, such as if the column is a primary key, nullable, unique, indexed or incremented automatically. Additionally, a wrapper for primary key configuration exists via the @primaryKey annotation (equivalent to annotating with @Column(primaryKey: true, databaseType: ManagedPropertyType.bigInteger, autoincrement: true)). In Figure 17, a database table for storing article data is shown being mapped onto an Aqueduct data model. Article id values are the table's primary key while the article publish dates are indexed. Finally, the class is annotated with the @Table annotation which allows for the database table's name to be manually set, rather than the default, using the class's name.

```
final query = Query<User>(context)
  ..values.name = "Bob"
  ..values.email = "bob@stablekernel.com";

final user = await query.insert();
```

Figure 18 – Code: running an insert query via Aqueduct's ORM to add to a User table [62].

Aside from modelling data, ORMs also provide developers with an interface to query databases. An example on how to insert data into a User table in Aqueduct's ORM is shown in Figure 18. Initially, a query object is created using the context. Then, using the cascade notation to access multiple attributes, the data to be inserted can be specified (in this case, "Bob" as the name and "bob@stablekernel.com" as the email). Finally, once the query is fully specified, the method relative to the query operation, in this case the insert() method, is called on the query object. This being an operation over an external system, a database, it can take some time, making this method be asynchronous and, as such, possible to be awaited on. Once the operation concludes, the object that's equivalent to the data inserted into the table is stored into the "user" variable.

Dart Type	General Column Type	PostgreSQL Column Type
`int`	integer number	`INT` or `SERIAL`
`double`	floating point number	`DOUBLE PRECISION`
`String`	text	`TEXT`
`DateTime`	timestamp	`TIMESTAMP`
`bool`	boolean	`BOOLEAN`
`Document`	a JSON object or array	`JSONB`
`Any enum`	text, restricted to enum cases	`TEXT`

Figure 19 – Aqueduct ORM's Dart code to PostgreSQL column type equivalence [60].

As stated previously, ORMs make sure data keeps its properties through conversions in both directions. In Aqueduct's case, all type conversions are fixed and are shown in Figure 19.

Unlike in Aqueduct, Angel allows for different databases to be used through migrations [63]. Migrations allow for databases to be altered and take in different connection classes for different databases, such as a PostreSQLConnection object when using a PostgreSQL database.

When it comes to modelling data, Angel makes use of abstract classes to define its models and, just like Aqueduct, annotations to describe properties a column should have in the database table.

```
@orm
@serializable
abstract class _Todo {
    bool get isComplete;

    String get text;

    @Column(type: ColumnType.long)
    int get score;
}
```

Figure 20 – Code: defining a database data model in Angel [64].

The main annotation used is @Column and it allows for setting properties such as a column's SQL type, index type and whether it is nullable. Each attribute also has an implicit getter method from the usage of the "get" keyword. As for class-wide annotations, @serializable marks the class for serialization, which is required by Angel to proceed with the class-to-table mapping, and @orm allows for setting the table's name, just as Aqueduct's @Table annotation, as well as specifying if migrations should be generated by Dart's builder. An example is shown in Figure 20.

As mentioned in the last paragraph, SQL column types can be specified in Angel's ORM, unlike in Aqueduct's. While in Aqueduct an int variable always maps to either an INT or a SERIAL SQL column type, Angel allows for more types to be used, such as BIGINT, SMALLINT, TINYINT and more. This increases developer freedom, yet also increases the chances of bugs occurring.

```
Future<void> markAsComplete(Todo todo, QueryExecutor executor) async {
    var query = TodoQuery()
        ..where.id.equals(todo.idAsInt)
        ..values.isComplete = true;

    await query.updateOne(executor);
}
```

Figure 21 – Code: running an update query via Angel's ORM [64].

The way data is queried in Angel is remarkably like Aqueduct's. Initially, a query object is also created, followed by building the query itself through cascade notation usage for increased legibility and finalized with an asynchronous call to the respective database operation which is a single row update in the case of the example given in Figure 21, via the updateOne() method.

```java
@Entity
public class City implements Serializable {

    @Id
    @GeneratedValue
    private Long id;

    @Column(nullable = false)
    private String name;

    @Column(nullable = false)
    private String state;

    protected City() { }

    public City(String name, String state) {
        this.name = name;
        this.state = state;
    }

    public String getName() {
        return this.name;
    }

    public String getState() {
        return this.state;
    }
}
```

Figure 22 – Code: defining an entity in Spring Boot [65].

Just as in Angel, Spring Boot allows for multiple databases to be used, such as MySQL and PostgreSQL, due to using the JPA, or Java Persistence API, library and JPA databases are created automatically when using embedded databases, such as H2 [66] and HSQL [67].

JPA provides object-relation mapping through entity describing by annotating a class with @Entity, so that JPA can become aware of its existence. Classes must include a constructor with no arguments, ideally protected from direct use, and a primary key specification, which can be done via the @Id annotation being used in a class attribute.

In Figure 22, an example on how to create an entity to store data related to cities is shown. As explained in the previous paragraph, the class is annotated with @Entity so it can be recognised and features an empty protected constructor, as well another constructor for convenient object creation. The class also extends Serializable so that the data can be sent outside of the Java Virtual Machine environment to a database. In terms of attributes, the “id” is annotated with both @Id, marking it as a primary key, and @GeneratedValue, an annotation that allows for setting the generation strategy for a simple primary key. The remaining attributes, “name” and “state” are annotated with @Column,

which works similarly to its counterpart in both Aqueduct and Angel, being able to set whether a column is nullable or unique, amongst other properties. Finally, getter functions for the "name" and "state" attributes are defined.

```
public interface CityRepository extends Repository<City, Long> {

    City findByNameAndStateAllIgnoringCase(String name, String state);
}
```

Figure 23 – Code: extending a data repository interface in Spring Boot [65].

The way to access data in Spring Boot is through the repository interfaces in which queries are created based off the names given to methods. A simple example of this is shown in Figure 23 where a CityRepository interface is created by extending Repository [68], one of the most used interfaces (with another being CrudInterface [69]). In this interface, a findByNameAndStateAllIgnoringCase() method is specified and it generates a query that returns a row based off the case-insensitive "name" and "state" values passed into the function as arguments. This row is then mapped to a City object and returned by the function.

3 Crypto Currency Manager (CCM) scenario

The study's main objective was to assert the adequacy of Dart-based solutions in providing a full-stack software (SW) solution. To accomplish this objective, a testbed scenario providing a typical architecture for enterprise SW solutions was defined. The scope was cryptocurrency and the goal to create a Crypto Currency Manager (CCM), a simple scenario that covers all the way from the top of the stack, the front end comprised of a mobile/web User Interface (UI) down to the back-end servers, for cryptocurrency quotation administration.

CCM is a system that tries to emulate current standard service back ends. Users can not only perform the expected session requests, such as registering and signing in, but also both sell and purchase different cryptocurrencies, the main service being provided. This scenario was expected to both be simple to implement and to act similarly enough to an actual system in terms of the data circulating to not compromise its integrity. The system should provide a front end via web and mobile UIs, and a REST (Representational State Transfer) API (Application Programming Interface) for data access. From an intra-system perspective, it should rely on a database repository and provide logging functionality for system monitoring and performance analysis.

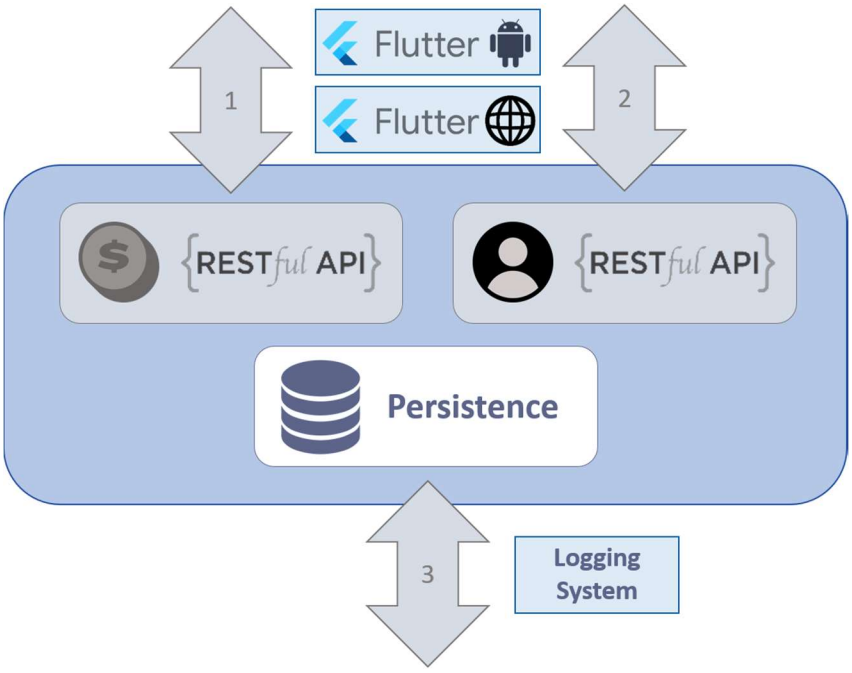

Figure 24 – Generic CCM scenario overview.

By providing a service for users, these must interact with it in some way, that being through the front-end applications. These applications forward their requests through either RESTful API controller present, be it the currency (1) or the user (2) one depending on the request, as is shown in

Figure 24. The CCM system then services the received request, accessing and/or editing the persistent storage database as necessary, and sends a request of its own to the external logging system.

3.1 CCM RESTful interface

Representational state transfer, or REST, is an architectural style used in SW development to forward external HTTP (Hypertext Transfer Protocol) requests into a system so that data can be accessed and modified. As shown in Figure 24, it is present in the CCM solution and is used to communicate with the front-end applications, via GET and POST requests.

Request Path	Request Type	Request description
/crypto/currencies	GET	Request that returns a list of all existing currencies. An "OK" message is returned in the "OPStatus" JSON field and the list of currencies is returned in the "Currencies" JSON field.
/crypto/currencies/:name	GET	Request that returns the information pertaining to the specified currency, if valid. If the specified currency is valid, an "OK" message is returned in the "OPStatus" JSON field, as well as the currency information in the "Result" JSON field. Otherwise, "INVALID_CURRENCY" is returned instead if the specified currency is invalid.
/crypto/exchange/sell	POST	Request that allows a user to sell a specified amount of currency, if given a valid input. If the sale is successful, an "OK" message is returned in the "OPStatus" JSON field. Otherwise, "INVALID_SALE" is returned if the sale could not be completed, for whatever reason.
/crypto/exchange/purchase	POST	Request that allows a user to purchase a specified amount of currency, if given a valid input. If the purchase is successful, an "OK" message is returned in the "OPStatus" JSON field. Otherwise, "INVALID_PURCHASE" is returned if the purchase couldn't be completed, for whatever reason.

Table 1 – CCM Currency language-independent RESTful API paths.

In Table 1, the currency related RESTful API paths required to simulate a realistic service are described. These paths allow for a simplified cryptocurrency system to be deployed and were inspired by a RESTful API created by NiceHash [70], a Slovenian cryptocurrency platform for mining and

trading with an integrated marketplace where buyers and sellers of both cryptocurrencies and hashing power can interact [71].

Request Path	Request Type	Request description
/user/register	POST	Allows a user to register, if given a valid input. If the user is successfully registered, an "OK" message is returned in the "OPStatus" JSON field. Otherwise, "REGISTRATION_ERROR" and "ALREADY_REGISTERED" messages are returned instead if a database error occurred or if the user is already registered, respectively.
/user/signin	POST	Allows a user to sign in, if given a valid input. If the user successfully signs in, an "OK" message is returned in the "OPStatus" JSON field. Otherwise, "SIGN_IN_ERROR" is returned instead if the sign-in failed, for whichever reason.
/user/signout	POST	Allows a user to sign out, if given a valid input. If the user successfully signs out, an "OK" message is returned in the "OPStatus" JSON field. Otherwise, "SIGN_OUT_ERROR" is returned instead if the sign-out failed, for whichever reason.
/user/currencies	POST	Fetches the amount of each currency a user has, if given a valid input. If the data is successfully fetched, an "OK" message is returned in the "OPStatus" JSON field, as well as the user's current amount of money in the "Money" JSON field and the amount of each different owner/previously owned currency in the "Currencies" JSON field. Otherwise, "ACCOUNT_ERROR" is returned instead in the "OPStatus" field if a database query failed.

Table 2 – CCM User language-independent RESTful API paths.

In Table 2, the user related RESTful API paths required to keep track of each user's information are described. Front-end applications for most types of services require users to create sessions through the usage of secret information, such as username and password combinations, to assert that the handled requests are coming from a legitimate source. Information pertaining to each individual user is stored in persistent storage, such as databases, and require the previously mentioned secret information to be able to access/modify it (often simply requiring it when signing into an account).

3.2 Spring Boot CCM – The comparison reference

It was considered relevant to have some reference to evaluate/compare the Dart-based CCM system. As such, it was decided that a CCM with the same requirements should be implemented and deployed. Given its popularity, Java's Spring Boot was used.

Flutter has already settled into the front-end scene as a competent competitor [6] due to its ability to develop for multiple platforms while still obtaining satisfactory performance, meaning that it was more than fit to be used to develop the front end for this system. This led to the decision of sharing UI solutions developed in Dart. It was also decided to share a logging solution between CCM systems.

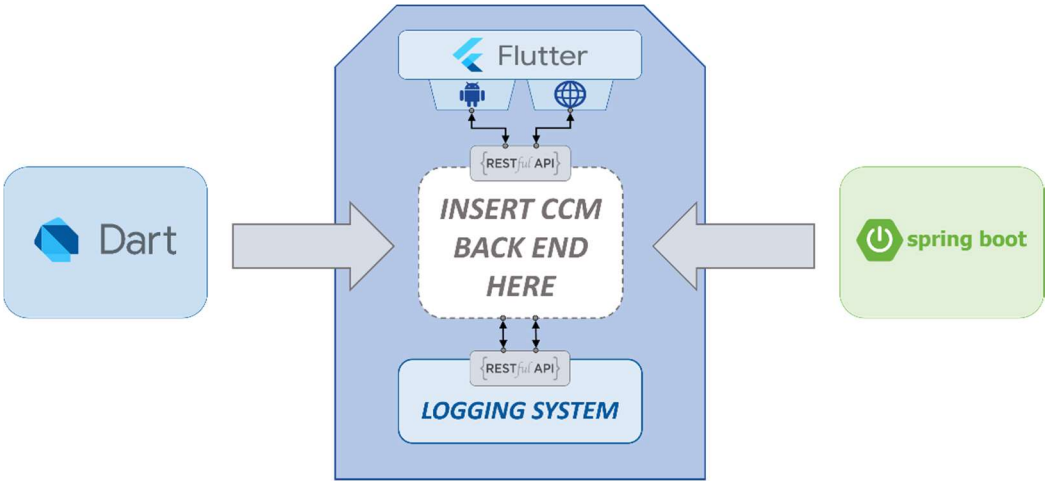

Figure 25 – Full system overview with modular back-end CCM servers.

The back-end implementation was decided to be the study's primary focus – besides being the most unexplored side of Dart it also possessed the highest potential performance bottleneck as it would provide both UI and API support for the CCM systems. An overview of the overall architecture can be found in Figure 25 where a Dart and Spring Boot back-end implementation case is shown.

4 CCM (Crypto Currency Manager) implementations

In this chapter, the implementation process for both full-stack systems, the Dart Angel CCM and Spring Boot CCM versions, will be explained. A simplification of how the study's full system was built is shown in Figure 26.

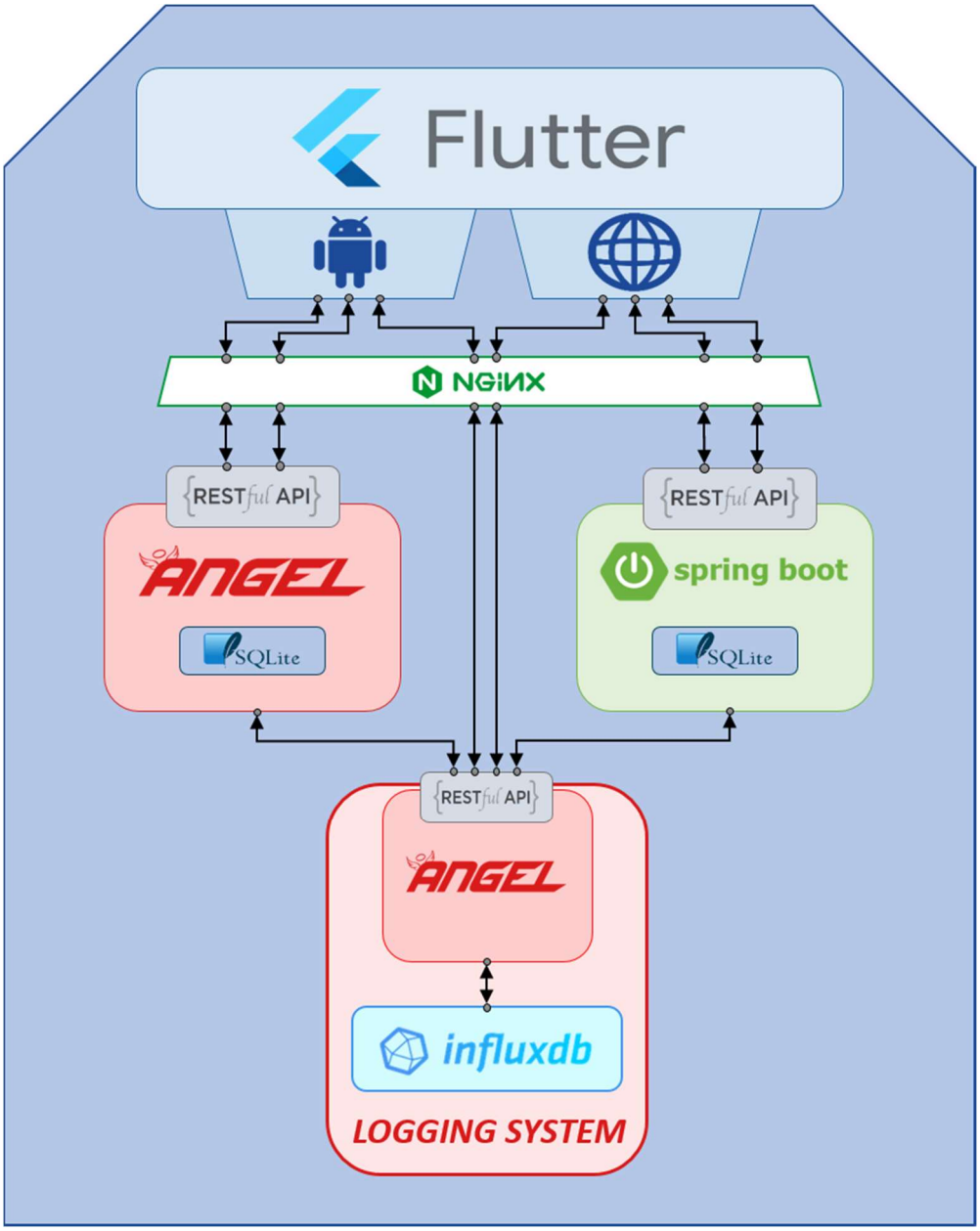

Figure 26 – Full study case with parallel Dart Angel CCM and Spring Boot CCM back ends.

4.1 Dart back-end CCM server

Just as with other programming languages, even if Dart has the capability to be used to develop back-end servers, the process can always be made better in terms of productivity by using frameworks. These handle a lot of configurations, which would otherwise be cumbersome, and provide extra tools to create an overall better experience for the developer. This being the case, the first step taken was to conduct some research on the available options. The two frameworks that immediately stood out, due to being the most popular, were Aqueduct [9] and Angel [10], with the former being the most popular and robust looking.

There were several other options available, yet all of them less popular. The ones encountered were Jaguar [31], the most robust looking, Start [32], Shelf [33], Vane [34] and Alfred [35]. These were not explored too much, but most of them seem to employ similar syntaxes when treating logical REST (Representational State Transfer) [29] constructs.

Aqueduct was an extremely robust Dart HTTP web server framework for building REST applications that employed a lot of solutions similarly to current state of the art. Due to this, it was the initial choice to develop the CCM system with, however, a few weeks into the study, the Aqueduct development team announced they would be halting development on the project, with the last commit onto the project's GitHub having been on August 31st of 2020 [72]. This made it so the main option available was Angel, the framework used throughout the study.

Angel is a simpler production-ready back-end framework developed for Dart with extensibility in mind. At this study's inception, Angel, or, more appropriately, Angel2 [73] at the time, had version 2.1.1 as the most recent release, dating February 5th of 2020 [74], half a year prior to Aqueduct's last one, which was already extremely outdated package-wise. This led to a variety of issues throughout development, related to package compatibility due to more recent packages, ones kept up to date, sharing dependencies with outdated ones, but different versions, which would lead, in most cases, to running older versions of the up-to-date packages, just to accommodate the outdated ones, due to the lack of a suitable replacement.

After March 3rd of 2021, issues started to escalate further as Dart's version 2.12 had a stable release [75] that brought to it Null Safety [14]. Null Safety is an especially important aspect any programming language can have, and it had always been highly requested. Given such interest from the community, as well as importance code-wise, it is no surprise that most packages would start to update to be able to support it. As great of a milestone as this might have been for Dart, it was also a huge hindrance in another regard: there were, and still are, an incredible number of abandoned packages which could still function while outdated, given the programmer knew what had to be forfeited to make them work, that due to sharing dependences with packages that transitioned into

Null Safety no longer could (efficiently). As newer packages started being created, ones with no version prior to Null Safety's release, less and less options to use in developing for outdated frameworks became available. Eventually, despite all the problems, a usable and compromise-filled system was developed for the study's purposes.

On May 14th of 2021, Angel3 officially released [76] under a different developer who had taken over the project on February 14th of 2021 [77]. It released with version 4.0.0, however due to being a new project, would not yet be prominently favoured by search algorithms. This new Angel would only be adopted into the study post version 4.1.0's release which happened on the 22nd of June 2021 [78]. Angel3 would now require a minimum Dart SDK version of 2.12 to function and be fully Null Safety compatible, enabling for this study's system to be updated and remove most of its constraints instantly. All struggles had during this study, both before and after the Angel3 migration, will be mentioned throughout this section.

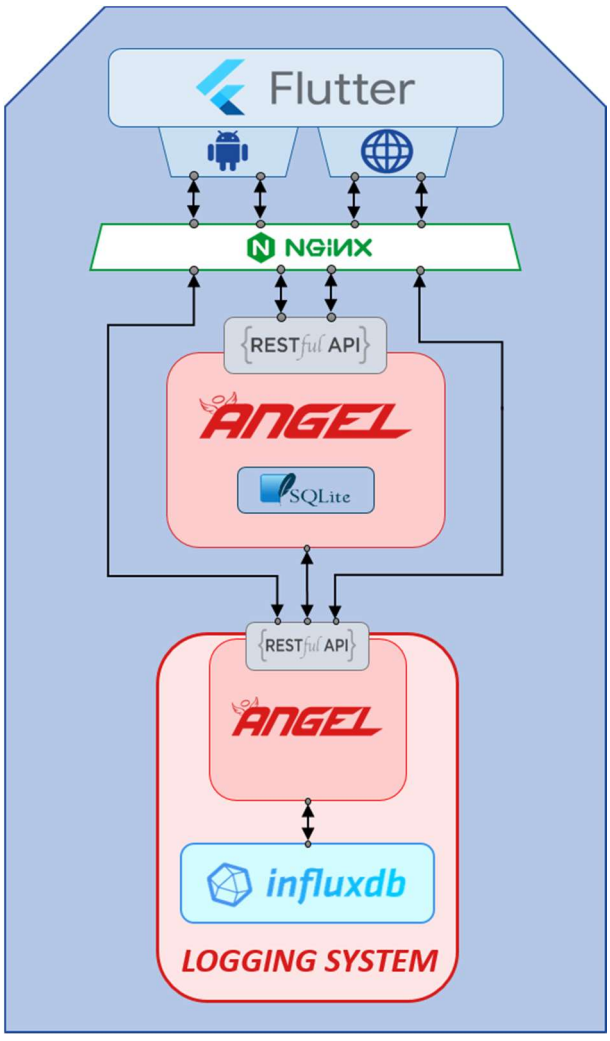

Figure 27 – System overview with Dart Angel back-end server.

In Figure 27, a post-development overview of the Dart Angel CCM system is shown. When compared with the one in Figure 25, the main difference concerns the fact that the Dart back-end server is now an Angel one, due to the choice of framework, and is now incorporated into the system. Another apparent difference is the elaboration of the logging system by incorporating an Angel subsystem, which will be further explored later, and an InfluxDB (Influx Database) [79] instance. Due to the lack of logging options available to Dart, a solution based off the usage of a time series database (TSDB), an increasingly popular approach, was developed. By doing this, it was expected to have the system make more readily available any logged metrics for better system diagnosis and maintenance.

Since the start of development with Angel, a problem that made itself felt throughout the entire process manifested itself: package version incompatibilities, as stated previously. A variety of packages chosen to be used due to the advantages they provided, while also working in a comparable way to state-of-the-art solutions, created issues while attempting to be fetched. Despite Angel's packages being compatible between themselves due to their dependencies either being maintained by themselves or the Dart team, that fact does not mean they are with external packages which are, often, required to develop competent back-end servers due to the number of different technologies required.

One of the main dependency issues was between the angel_orm package [80], which provides runtime support for Angel's ORM (Object-Relational Mapping), and the Influxdb_client [81] package, which allows for data to be efficiently sent to an InfluxDB instance. The fact that there were no version combinations to have both included in the same system led to the creation of a secondary system to handle the InfluxDB operations. This new system within the same back end would henceforth be referred to as the logging subsystem due to its function being to log the information sent to it (post the sent message to the standard output and send the metrics given to the InfluxDB database). Later, it was removed from the Dart Angel back end and added into the external system, used by both CCM implementations.

The Influxdb_client package had always caused issues prior to Angel3's release, making it so other libraries would require downgrading to function, and even being fully incompatible with the angel_orm package, as previously stated. Post Angel3 migration though, it became fully unusable. Due to this, an alternative that would still allow to send information to the InfluxDB instance had to be formulated. It was decided to send the information directly to the client through HTTP POST requests. This transition presented issues when transitioning both CCM implementations away from bridge libraries, however the Spring Boot one was the most impacted by the change. In terms of issues had developing the Angel CCM, they were mostly minor and related to the headers being sent. The main issue would be related to setting the data point's timestamp. When specifying the

timestamp precision and using the DateTime class's method that returns the milliseconds since epoch, the request would simply fail with the following return message: "failure writing points to database: partial write: points beyond retention policy dropped=1". This being returned means that the given timestamp, despite being the current time when sending, was beyond the retention policy [82] set for the client (the window of time where points of data are valid). The solution to this issue was simply not stating the postage timestamp and letting InfluxDB handle it, since the relevant timestamps were being sent as actual values to be stored within it. More info regarding the final iteration of the external logging system can be found in a later section.

Despite tools for API (Application Programming Interface) code generation existing for Dart and using them for both CCMs possibly removing yet another variable that could influence timings, both the amount of bloat code and the incompatibility with Dart Angel (endpoints not controlled nor accessible by the AngelApp instance) made them not be a viable option to use, given the simplicity of the problem at hand. Due to this and to attempt to minimize API differences with the Spring Boot implementation, the Swagger Inspector Extension [83] for Google Chrome/Microsoft Edge was used to obtain a specification to follow when developing the other implementation.

The CCM system handles external requests. This is done via receiving HTTP requests which contain the required information to proceed with the internal operations. Every request also triggers the sending of four POST requests to the logging system via REST. These requests were used to gather metrics and gauge system performance. Given the fact that they were handled asynchronously, they did not impact the overall system's performance in a negative way.

4.1.1 Persistence and ORM usage

Due to most of the CCM operations involving interacting with local data to maintain a correct functionality state for the service and the advantages ORM bring to developers in this area, Dart ORMs were explored, starting off with the framework's own ORM package, angel_orm [80]. This package required a large time investment and ended up not fully functional, missing crucial elements required to be functional. The issues encountered while using this package were as follows:

- The pub.dev installation page for angel_orm only mentioned the angel_orm dependency addition being required, while its GitHub readme file [84] also included the need for the angel_orm_generator [85] and build_runner [86] developer dependencies.
- Both the pub.dev page and the GitHub readme file failed to mention the required build_runner command required to generate the ORM files.
- The angel_orm GitHub readme file mentioned that the ORM worked better when using the package angel_serialize [87] yet did not give example on how to create a vanilla project.

- The basic functionality page for angel [64] specified that angel_serialize is the foundation for angel_orm, meaning it was not option, as would be assumed from the GitHub readme.
- The pub.dev installation page for angel_serialize only mentioned the angel_serialize dependency addition being required, while its GitHub readme [88] file also included the need for the angel_model [89] dependency and the angel_serialize_generator [90] and build_runner developer dependencies.
- Due to the angel_serialize package requiring angel_model, the angel_orm page failed to mention it also required it to function.
- Once all the (known) issues were taken care of, the ORM files were generated. Not every file that was supposed to, according to the documentation, was.

Post migrating to Angel3, the ORM was given another chance due to the possibility of having been worked on by the new developer. The same issues as before were found, plus new additional ones, such as the documentation page for angel3_serialize [91] having no info except a hyperlink to the "Documentation", which proceeded to take the user to the very same page (fixed since then).

Other ORM alternatives were explored yet ended up being dependent on their own frameworks, calling for a different type of solution: direct SQLite [92] implementation. Despite a lot of developers not being too fond of writing SQL queries or even using DBC (database connectivity) libraries/packages, despite optimally written queries outperforming ORM usage, not only was it easier but also less time expending than attempting to use an ORM in Dart, meaning that, as with other ORMs, Dart's options also bring with them an increase in overhead. Add to this the fact that most of the database actions are driver dependent, drivers which are shared by implementations across multiple languages, and it minimizes the impact the database operations have on the different CCM timings.

Despite it being both fast and easy to implement SQLite support for a project running on native Dart, the sqlite3 [93] package only provides support for atomic operations, not having a way to handle transaction logic, something important for corporate level services. Due to this, alternatives were researched and one looked promising: Moor [94], since then renamed to Drift [95].

Drift is a reactive persistence library that allows for queries to be both written in SQL and Dart, fulfilling the role of an ORM when it comes to increasing efficiency and handling what some developers dislike about working with SQL. An attempt to use it was performed yet resulted in an unexpected issue when attempting to get the dependencies: it stated that the Flutter SDK was required. Upon closer inspection, despite the Drift website stating in its main header that the package is "for Dart" and that the package's pub.dev page is tagged with "Dart Native", the package is

intended to be used in a Flutter environment. This, once again, shows a case where third-party Dart documentation is lacking in quality.

Continuing the search for other options, the only other package worth mentioning is sqflite [96], yet, once again, it requires a Flutter environment to work, which resulted in the system not employing transaction logic in its Dart CCM.

4.1.2 Currency manager implementation

The Dart Angel CCM, being an implementation of the CCM scenario, must follow the same functional objective. Due to this, a simple currency value fluctuation simulator was developed to provide the front-end applications with realistic-looking data.

For simplification purposes, only three currency types are managed, these being named after well-known precious metals: GoldCoin, SilverCoin and BronzeCoin. At the time of system initiation, all these currencies have their values set to 15.0 and are configured to fluctuate pseudo-randomly as time goes on.

```
_updateTimer = new Timer.periodic(Duration(seconds: 3), (timer)
{
  for (String c in _currencyValues.keys)
  {
    _currencyValues[c] = _currencyValues[c]
        + Random().nextDouble() / 10 - 0.05;
    if (_currencyValues[c] < 0) _currencyValues[c] = 0;
  }
});
```

Figure 28 – Code: currency value fluctuation over time using a Dart timer.

As shown in Figure 28, a repeating timer was instantiated through the usage of a Timer.periodic() constructor [97]. The first argument passed to the constructor was a Duration with the "seconds" argument being set to "3", meaning the timer would repeat itself with intervals of three seconds between callback invocations. The second argument passed was the callback method just stated, in which the value variability for each currency was pseudo-randomly generated. A value could not increase nor decrease by over 0.05 per invocation, nor could it ever become negative.

Due to Dart's focus on asynchronous code and the number of available native solutions for a wide variety of asynchronous problems, the overall implementation process was very simple, only requiring an efficient data structure to store the currency/value data, such as a HashMap, a timer

instance, an initialization method to set the initial state of the manager and the timer configuration, and some simple getter methods for the data to be available to the Dart Angel controllers.

4.1.3 Package specification

The packages used to develop the final system changed over the course of the study, not only being replaced by others but also updated (changed to a more recent version of the same package). Due to Dart's own "pub get" command handling all package management, there was no need for an external build manager in Dart.

The packages present in the final iteration of the system are shown in Table 3 and any outdated ones were simply not updated due to releasing after the study's system being considered finalized and used to obtain the results.

Package	Version Used	Latest
angel3_framework **[98]**	4.1.1 (8th of July, 2021)	4.2.1 (4th of October, 2021)
http **[99]**	0.13.3 (3rd of May, 2021)	0.13.4 (4th of October, 2021)
logging **[100]**	1.0.1 (25th of March, 2021)	1.0.2 (14th of September, 2021)
sqlite3 **[93]**	1.1.2 (28th of May, 2021)	1.3.1 (19th of October, 2021)
synchronized **[101]**	3.0.0 (17th of February, 2021)	3.0.0 (17th of February, 2021)

Table 3 – Dart Angel CCM package versions.

The angel3_framework package contains all the necessary angel dependencies to have the framework function correctly. There are additional add-on packages to further complement the base one, such as angel3_orm, yet they are not essential to it functioning.

The http package allows for HTTP requests to be forwarded, thus being able to interact with RESTful APIs. It provides developers with high-level functions that act as wrappers for Dart's native options. Prior to migrating to Angel3, version 0.12.2 had to be used due to subsequent ones (>= 0.13.0) being incompatible with angel_framework (due to an incompatibility between http_parser versions), thus being outdated by four versions.

The logging package provides the programmer with a Logger that asynchronously handles logged information in any way implemented in its handler. Prior to migrating to Angel3, version 0.11.3 had to be used due to subsequent ones (>= 0.11.4) being incompatible with angel_framework, thus being outdated by three versions.

The sqlite package provides Dart bindings to SQLite. With it, interaction with SQLite databases is possible and even allows for shipment of custom SQLite versions, rather than just using the OS version.

The synchronized package provides basic lock mechanisms to prevent concurrent access to asynchronous code.

4.2 Spring Boot CCM back-end server

The Java Spring Boot language/framework combination was chosen to be used to develop the reference back-end server due to it being currently considered state of the art in the way that it deals with most service-related operations.

By developing a system using state-of-the-art solutions to act as a reference, benchmarks can be set to better judge how competent any compared system is at dealing with each given situation. As such, the same exact scenario used in developing the Dart-based CCM back-end server was used in developing the Spring Boot one, resulting in the system shown in Figure 29, which is the exact same as the system shown in Figure 27 but with a different back-end server.

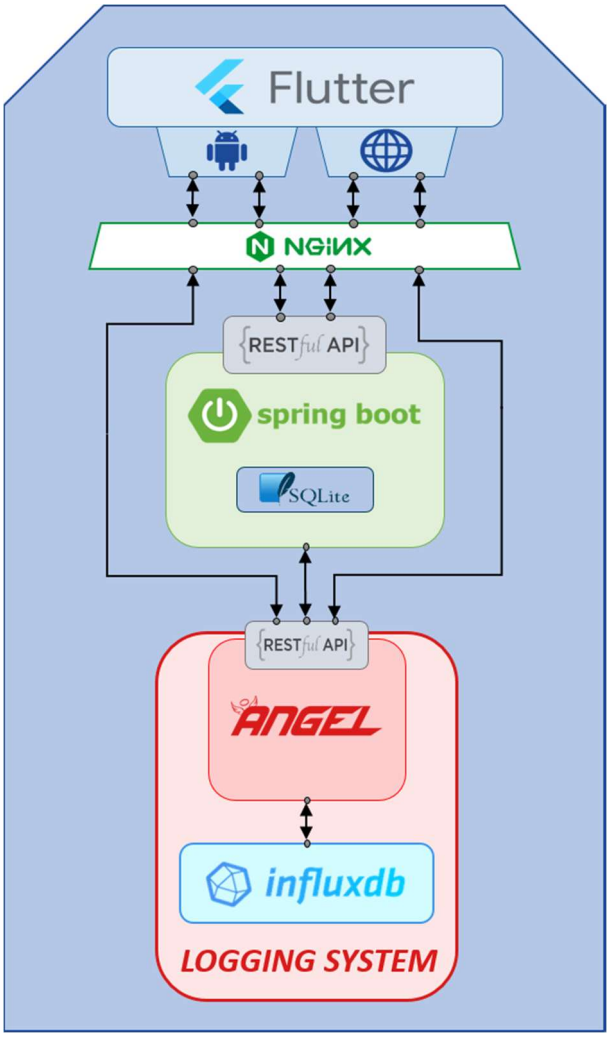

Figure 29 – System overview with Spring Boot back-end server.

The development process was straight-forward without any issues arising. The only real issue occurred when swapping from using the influxdb-client-java dependency [102] to sending all information to the logging system's Angel subsystem to reduce the study's variables and obtain a better comparison. While in the Dart-based solution the primary issue was related to the data being sent, namely the timestamp, the issue encountered in the Spring Boot implementation was a lot more bizarre. Initially, the information was being sent by using Java's HttpURLConnection [103] class, a typical Java solution, however the requests reaching the logging system would always have an empty body. This was discovered using Wireshark [104], a network protocol analyser, to conduct package sniffing in the network between the Spring Boot CCM and the Dart Angel logging subsystem. The response packets contained an error message, not available in the CCM console due to only an exception being thrown, that led to believe that the information was being sent incorrectly, as a value was arriving to be used as null. When looking at the packets being sent by the CCM, they were found to have been sent correctly (with a body consisting of the correct information), meaning that the CCM was performing its operations correctly. This situation was irregular since the logging subsystem was functioning perfectly when the Dart Angel CCM was the one sending the requests.

After multiple attempts at using different header values, it was decided to try out other forms of sending the HTTP requests, with Java's HttpClient [105] being the next candidate. Unfortunately, the result would be the same: an empty body arriving at the logging subsystem. In this case though, rather than the error only being available through packet sniffing it was also made available in the CCM console. Finally, it was decided to try Spring Boot's own solution, the RestController [106] class. By using this class, the issue was solved and it, once again, just shows how efficient Spring Boot is at solving developer's problems, especially since the time it took to implement the RestController solution was around half the taken in the previous implementations.

Just as in the Dart Angel CCM, the Spring Boot CCM system handles external requests via receiving HTTP requests which contain the required information to proceed with the system's internal operations. Every request triggers the sending of the same four POST requests as in the Dart Angel implementation to the logging system via the same REST endpoint. While Java lacks in terms of asynchronous logic when compared to Dart, Spring Boot possesses a solution to the problem in the form of the @Async annotation, which was taken advantage of to send the requests with minimal temporal repercussions. By annotating the method used to send the information with @Async it indicates to the system that the method should be run on a different thread and the system could know of this by annotating the class the previously annotated function belongs to with @Service to make it so Spring's component scan is aware of it existing. Finally, the application class must be annotated with @EnableAsync to enable Spring's ability to run @Async methods.

Due to having used SQLite for persistence in the Dart Angel implementation, the same was used in this one, as well as the same data structures (table declarations) and operations/queries.

Finally, Spring Boot possesses a solution for API documentation in the form of SpringFox [107], which automates API documentation and even provides a user interface (UI) to browse through it. To minimize lines of code, or LoC, metrics and avoid any possible impact on performance, it was excluded from the project to better resemble the Dart Angel implementation as it isn't imperative to functionality.

4.2.1 Currency manager implementation

The Spring Boot CCM, just as the Dart Angel CCM, is an implementation of the CCM scenario and, as such, must implement the same functional objective. Due to this, another simple currency value fluctuation simulator was developed to provide the front-end applications with realistic-looking data.

As explained regarding the Dart Angel CCM's currency manager, only three currency types are managed (GoldCoin, SilverCoin and BronzeCoin), all being initialized with the value of 15.0.

```
@Scheduled(fixedDelay = 3000)
public void updateCurrencyValues()
{
    for (String c : currencyValues.keySet())
    {
        currencyValues.put(c, currencyValues.get(c) + Math.random() / 10 - 0.05);
        if (currencyValues.get(c) <= 0) currencyValues.put(c, 0.0);
    }
}
```

Figure 30 – Code: currency value fluctuation over time using a Spring Boot scheduled task.

As shown in Figure 30, a scheduled task was used to vary currency values over time. A scheduled task is created by annotating a function with @Scheduled [108], specifying either a "cron", "fixedDelay" or "fixedRate" argument. For the developed scheduled task, a value of "3000" was passed onto the "fixedDelay" argument, making it so the task is executed every 3000 milliseconds, the same three seconds used in the Dart Angel CCM version of the manager. The annotated method itself is tasked with varying the currency values whenever called, applying the same logic shown in Figure 28. Finally, to be able to schedule the task to execute, the class containing the function annotated with @Scheduled must be annotated with @EnableScheduling, to enable the function's detection.

Despite Java also having a Timer class [109] that can be used in an analogous way to Dart's, it involves a higher degree of bloat code as well as thread management, which is prone to errors, time consuming and generally avoided by developers. Due to this, Spring Boot's own solution, which solves both the issues, was used.

Aside from the usage of Spring Boot annotations, the remaining code logic was analogous to the Dart Angel counterpart, be it the usage of a HashMap to store the currency/value data, the initialization method, or the existing getter functions.

4.2.2 Dependency Specification

To not only facilitate the development process but also to make sure state-of-the-art solutions were being employed, Maven [110], a software (SW) project management and comprehension tool, was used to manage the project's build, using it to better access external libraries through dependency specification in Maven's Project Object Model file, or POM [111] file. It was also used due to Java not natively possessing a solution analogous to Dart's "pub get" command.

The dependencies used to develop the final system remained mostly unchanged throughout development and are shown in Table 4. The only exception was the sqlite_jdbc dependency, which updated a couple of times with minor changes.

Dependency	Version Used	Latest
spring-boot-starter-data-rest **[112]**	2.5.4 (19th of August, 2021)	2.5.6 (21st of October, 2021)
spring-boot-starter-web **[113]**	2.5.4 (19th of August, 2021)	2.5.6 (21st of October, 2021)
spring-boot-starter-log4j2 **[114]**	2.5.4 (19th of August, 2021)	2.5.6 (21st of October, 2021)
sqlite-jdbc **[115]**	3.36.0.3 (30th of August, 2021)	3.36.0.3 (30th of August, 2021)
spring-boot-maven-plugin **[116]**	2.5.5 (23rd of September, 2021)	2.5.6 (21st of October, 2021)

Table 4 – Spring Boot CCM library versions.

The spring-boot-starter-data-rest dependency is a staple is Spring Boot development, facilitating the creation of REST endpoints by handling a lot of the otherwise cumbersome boilerplate code.

The spring-boot-starter-web dependency is also a staple in Spring Boor development, allowing for the usage of Tomcat, an embedded server, as well as other tools. It was not used in its totality, excluding the spring-boot-starter-logging dependency.

The spring-boot-starter-log4j2 dependency adds log4j support to a project, also being a staple of Spring Boot projects alongside Logback (via spring-boot-starter-logging). It is also the reason for the spring-boot-starter-logging dependency being excluded from spring-boot-starter-web.

The sqlite-jdbc dependency allows for SQLite databases to be accessed via the JDBC (Java Database Connectivity) API. This was the only dependency receiving updated over the course of the study. The version initially used was 3.34.0, meaning the dependency was updated a total of six times over the entirety of the study.

The spring-boot-maven-plugin dependency allows for executable jar/war files to be created and was used to facilitate deployment into a testing environment.

4.3 Flutter web/mobile applications

The CCM application was developed using Flutter [3], allowing for it to be run on both mobile devices and conventional desktop browsers. The application was treated as proof of concept focused on testing and using the CCM back-end servers rather than on the final user experience. It featured two main interfaces – one for testing through metric collection, shown in Figure 31, and one with a basic UI for prospective CCM users focused on crypto currency, shown in Figure 32.

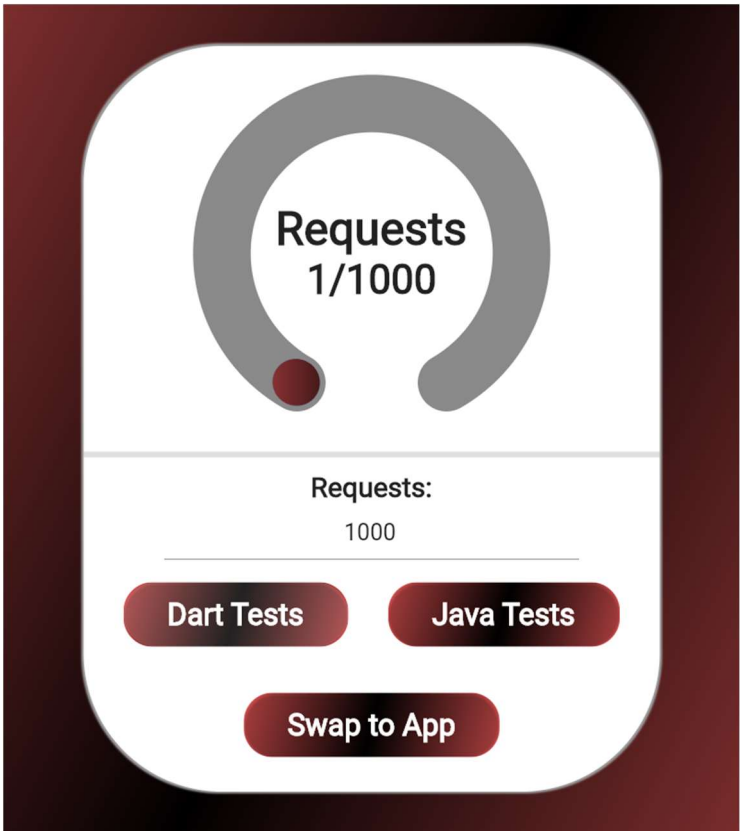

Figure 31 – Application testing screen for metric collection.

The testing screen, shown in Figure 31, displays a circular progress bar (made using the sleek_circular_slider package [117]) which, accompanied by a textual aid at its centre, tracks completion for a testing session; a text form field to input the desired number of requests to be run; and three buttons which allow for sending requests to the Dart Angel back-end server, sending requests to the Spring Boot back-end server, or changing the screen over to the application one.

Although the testing screen is clearly out-of-scope for a CCM user, it was critical to collect, log and measure the complete temporal scope of specific requests between the mobile application and the CCM back ends. Despite logging requests being sent from each back end to the logging system these only provide information regarding events from within the back end itself, not considering timings related to the API request handling process. For both the "Dart Tests" and the "Java Tests" buttons, the same request sending logic was used, simply to different endpoints. More information on specifics related to the sent requests and reasoning behind the choices opted for can be found in a future section.

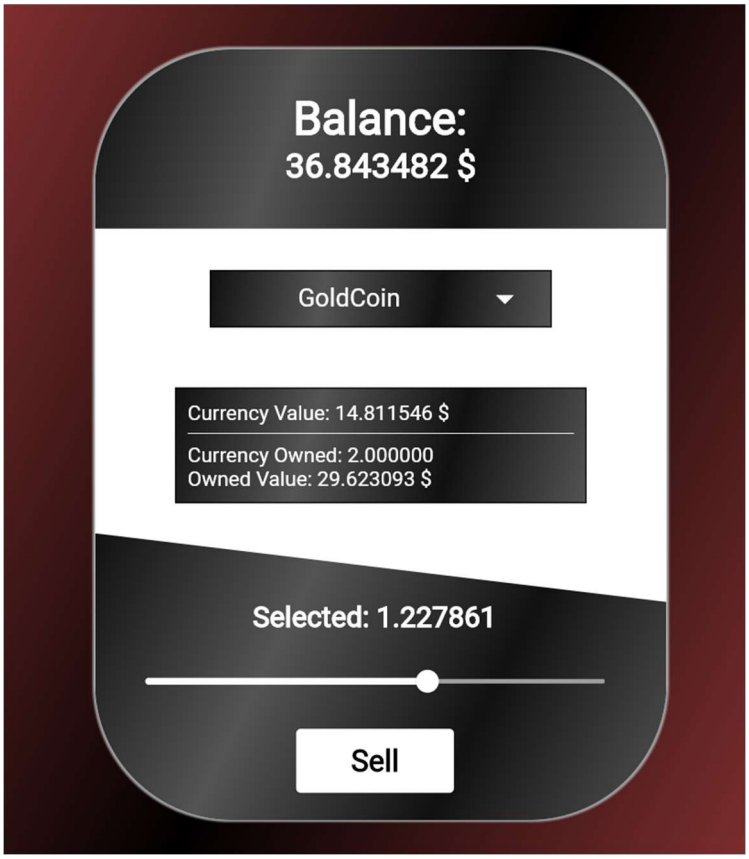

Figure 32 – Currency selling screen.

The CCM UI, shown in Figure 32, exemplifies one of the types of operations that a typical front-end application made to service the developed CCM back-end servers can implement. The screen is

divided into three main sections: the balance section at the top which displays the user's current account balance; the currency information section at the centre which allows for choosing which of the available currency info (the current currency value, how much the user owns and the user's total currency value) should be displayed through the usage of a dropdown menu; and the sale section at the bottom which allows for the user to select how much of their currency they wish to sell, then being able to do so through the click of a button. This screen was implemented merely for proof-of-concept and demonstration reasons.

4.4 Dealing with Cross-Origin Resource Sharing (CORS)

Although not a focal point of the present system, dealing with CORS [118] is a common task in back-end development and working on the CCM implementations was not an exception.

Cross-Origin Resource Sharing, or, as it is commonly known, CORS, is a security mechanism that allows for resources that are restricted to outside a web page's domain to be requested from within a different one. This is done through the addition of request/response headers:

- Origin: Request header that specifies where the request originated from. Used to lift restrictions based off the sender.
- Access-Control-Allow-Origin: Response header that specifies whether the resource is restricted to the given origin. The use of the "*" wildcard means any origin can access the resource.
- Access-Control-Request-Headers: Request header to indicate which HTTP (Hypertext Transfer Protocol) headers might be sent during the resource-related request. This header is used when issuing a pre-flight request.
- Access-Control-Allow-Headers: Response header to indicate which HTTP headers can be used during the resource-related request. This header is used in response to a pre-flight request containing the "Access-Control-Request-Headers" header.
- Access-Control-Allow-Methods: Response header that specifies which HTTP request methods can be used to access the resource. This header is used in response to a pre-flight request.
- Access-Control-Expose-Headers: Response header that specifies which response headers the requester has permissions to access.

In terms of implementation, controller logic was used to forward requests performed to the paths specified in Table 1 and Table 2. All request handling was done using Angel and Spring Boot's own packages/libraries.

```
@Expose("/foo")
class FooController extends Controller {
  @Expose("/some/strange/url/:id", as: "bar")
  someActionWithALongNameThatWeWouldLikeToShorten(int id) async {
  }
}

main() async {
  Angel app = new Angel();

  app.get("/some/path", (req, res) async =>
    res.redirectToAction("FooController@bar", {"id": 1337}));
}
```

Figure 33 – Code: routing a request using named controllers in Angel [53].

In Angel, it is possible to route requests from the basic routing function bodies to controllers using the ResponseContext class's redirectToAction() method. As shown in Figure 33, a name can be passed to the "as" argument present in the @Expose annotation. This name can then be called in the redirectToAction() method's first argument following the "controller@name" format, followed by the annotated method's argument passing. This allows for the "req" and "res" variables relating to the RequestContext and ResponseContext, respectively, to be passed into the controller to be used as normal. The main issue with this approach is related to scalability, as it implies the creation of as many different CORS configurations as required (when route grouping is not possible), all done via Dart code in a maintenance-unfriendly way. An even worse solution is to not use controllers at all and simply have their method bodies in the basic routing methods, which adds confusion to the scalability issue due to having configurations, request rerouting and functional code described in the same codebase.

The ideal solution would be to not have to deal with CORS configurations being handled in Dart code at all, which is what using NginX [119] achieved, as explained in the following section.

```
@CrossOrigin(origins = "http://domain2.com", maxAge = 3600)
@RestController
@RequestMapping("/account")
public class AccountController {

        @GetMapping("/{id}")
        public Account retrieve(@PathVariable Long id) {
                // ...
        }

        @DeleteMapping("/{id}")
        public void remove(@PathVariable Long id) {
                // ...
        }
}
```

Figure 34 – Code: enabling CORS in Spring Boot controllers [120].

In Spring Boot, handling CORS is a lot more straightforward. The @CrossOrigin annotation can be added to either a controller class or method to enable CORS in a localized way. As shown in Figure 34, it is added to the AccountController class with the origin specified to be "http://domain2.com", meaning that only requests originating from that domain can access the resources supplied by this controller, as well as the "maxAge" being set to the value 3600 to specify how long a pre-flight request can be cached for before having to send another one (in this case, 3600 seconds).

Due to the usage of NginX, as stated previously, no CORS configurations remained present in the final iteration of the study's CCM systems.

4.4.1 Handling CORS with NginX

Despite the existence of solutions to deal with CORS in both Dart and Spring Boot the level of verbosity and inconsistency of the in-code solutions led to the adoption of a very common solution at an industry level – the usage of NginX.

NginX is a free open-source HTTP server and reverse proxy characterized by its low resource consumption, high performance, and simple configuration. Rather than threads, it uses a highly scalable event-driven architecture that allocates a small amount of memory to each request being handled asynchronously, making it so even fewer active systems can still take advantage of the architecture's benefits.

NginX was used with the primary goal of handling CORS related issues without having to perform configurations when it comes to codebase. Aside from this though, NginX provides multiple other

advantages when a full-stack system is considered, such as the possibility of multiple back-end servers existing that require load balancing to be performed.

```
http {
    server {
        location / {
            proxy_pass http://localhost:8080;
        }

        location /images/ {
            root /data;
        }
    }
}
```

Figure 35 – Code: basic NginX configuration file [121].

In Figure 35, a simple NginX configuration file is shown. Server directives are specified within an http context, and each can have an ip address and port set to listen from, using the "listen" keyword, so that NginX can decide which server processes an incoming request. Servers then have location specifications, which set the configuration to use depending on the Uniform Resource Identifier (URI) found in the request with more specific locations taking precedent when multiple are hits. In the example, two locations are specified and if a request were to be made using, for example, the "/images/pie.png" URI then the second location would be used and the given URI would be added to the path specified using the "root" keyword, resulting in "/data/images/pie.png". If a request were to be made using the "/route" URI, then the first location would be used and the proxy_pass instruction run, forwarding the request to, in the case of the given example, "http://localhost:8000/route" due to the proxy_pass not having a URI specified, while if it did the URI would simply be replaced.

In terms of the implemented reverse proxy, three server specifications were user: two similar ones (for both CCM implementations, the Angel and Spring Boot ones) with two locations, for requests to be routed towards the correct user or currency controller, and a third one with a single location to service external requests to the logging system, such as from a web or mobile application.

5 The logging system

In programming, logging is the act of keeping a record, an event's timestamped documentation, about what happens within a system, such as data input/output or state management. It allows for developers to not only be able to observe the state of a running system, but also to identify malfunctions or simply irregular behaviours. Logging enables access to historical data, via past logs, to conduct both debugging and quality assurance. Due to the growing complexity in systems and tools over the years, logging has also increased in both utility and demand to the point where it became imperative to use in all corporate-level applications.

Due to its importance, a multitude of logging solutions, both internal and external to systems, exist. In Java, for example, two have settled in as state-of-the-art solutions: Logback [122] and log4j [123]. These have been perfected over time, adding support for extra tools, and even porting to other languages [124], and have already been proven to be efficient. In Dart, however, the same can't be said. There are no logging frameworks that stand out, with the main option being the "logging" package [100] which simply handles a lot of the bloat code programmers would have to deal with to implement their own version of it. In contrast with Logback and log4j, Dart's "logging" package does not provide a plugin architecture (usually called appenders [125]) that allows for extending to be able to write event data to a destination, such as to an external metrics system or database.

At the start of the study, the usage of a logging system was already envisioned in the high-level architecture (in the "Crypto Currency Manager (CCM) scenario" section) although details on its implementation had not yet clearly been defined. As the CCM system evolved, InfluxDB (Influx Database) [79] was considered as a viable option to act as the final log destination for both the Angel (Dart based) and Spring Boot (Java based) implementations. InfluxDB is a time series platform that empowers monitoring and analysis solutions. Due to its capability of ingesting millions of data points per second it pairs well with typical full-stack solutions which tend to generate a high number of logs asynchronously. Another option considered was remote logging onto something like logz.io [126] which could be achieved through the usage of the logging_appenders package [127] (a standalone package which depends on the "logging" package and solely provides support for logz.io, a paid service, and Grafana loki [128], not only very early into development at the start of this study but also requiring a sizeable amount of unjustified work for the purposes required).

In the Spring Boot CCM, log4j was used and data was being asynchronously sent to the InfluxDB instance through a bridge library, while the Angel CCM had a logging subsystem it would asynchronously send data to, which then both would post information to the system's standard output and send data to the InfluxDB instance, using a bridge package. The different integration with InfluxDB approaches led to question the impact of the logging option on the overall efficiency of

either system, hindering the study's primary goal of assessing Dart's ability to be used to develop back-end servers. As such, logging was entrusted to a common external logging system, communicated with via an HTTP (Hypertext Transfer Protocol) API (Application Programming Interface).

The external logging system was responsible for sending data to InfluxDB. This common logging system was Dart based and made use of the Angel logging subsystem. Sharing the same logging system allowed for the elimination of variability caused due to CCM language-specific variability in how logging integrations functioned which led to the study being fairer when assessing both CCM's performance. Additionally, by providing an open REST (Representational State Transfer) endpoint in the logging subsystem itself, it enabled for the sending of logs from the web/mobile applications as well.

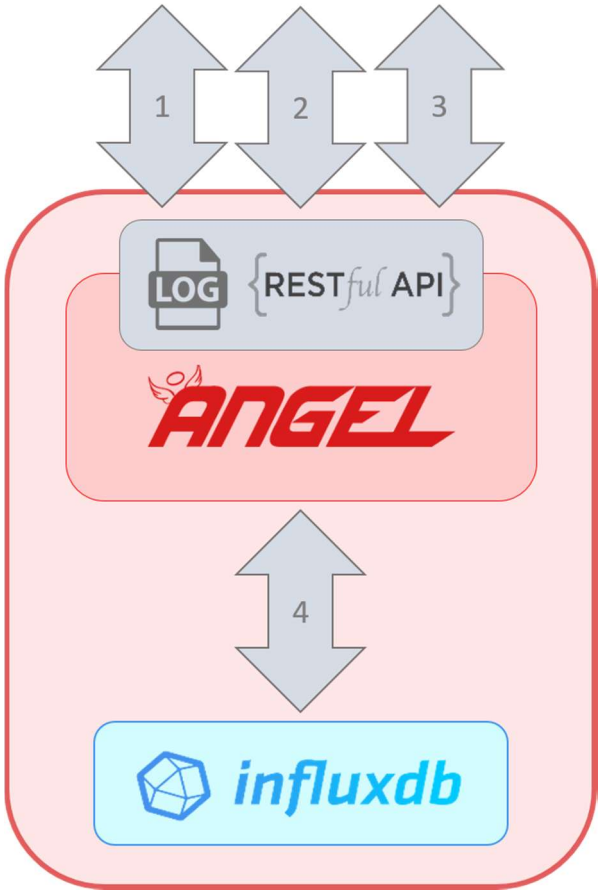

Figure 36 – Logging system overview.

In Figure 36, an overview of the logging system's final iteration is shown. It is composed by the previously mentioned logging subsystem, developed using Dart's Angel framework, and by an InfluxDB instance. The only way to communicate with the logging system was through the Angel

subsystem's RESTful interface, which could be used by both CCM implementations and by the web/mobile applications, as exemplified by the arrows marked with numbers 1, 2 and 3. The subsystem would then send the appropriate data to the InfluxDB platform through HTTP POST requests, as exemplified by the arrow with number 4, being the only way to do so in the entire system.

5.1 Dart Angel logging subsystem

The Dart Angel logging subsystem is the logging system's means of communication with the exterior. A RESTful interface was used by both CCM implementations and by the mobile/web applications to send logging requests. Using the same logging system allowed the logging times obtained to merely account for CCM specific times, namely when receiving/handling requests and performing database operations.

```
static final Logger _logger = Logger('InfluxDBConnection');

static Future log(String message, String measurement, String tagKey,
    String tagValue, String fieldKey, int fieldValue) async
{
  Uri uri = Uri.http(_url.substring('http://'.length), '/api/v2/write',
      {'precision': 'ns', 'bucket': _bucket, 'org': _org});

  Map<String, String> headers = {};
  headers['Content-Type'] = 'text/plain; charset=utf-8';
  headers['Authorization'] = 'Token ${_token}';
  headers['User-Agent'] = 'SC4ALLLogger/1.0.0';
  headers['Content-Encoding'] = 'identity';

  String data = '${measurement},${tagKey}=${tagValue} ${fieldKey}=${fieldValue}';

  http.Response response = await http.post(uri, headers: headers, body: data);

  if (response.statusCode == 204) _logger.fine(message);
  else _logger.severe('Could not send message.');
}
```

Figure 37 – Code: Dart Angel logging subsystem's logging solution implementation.

In Figure 37, most of the developed InfluxDBConnection class is shown, only excluding from the image some variable declarations with information such as the InfluxDB bucket and orgs to use, via the variables "_bucket" and "_org", and the class's enveloping scope.

```
Logger.root.level = Level.ALL;
Logger.root.onRecord.listen((event)
{
  print('${event.level.name}: ${event.time}: ${event.message}');
});
```

Figure 38 – Code: Dart Angel logging subsystem's log handler implementation (using the logging package).

Initially, a Logger is instantiated, passing onto it the class name for code localization reference. Any usage of a logging method, such as Logger.severe() or Logger.fine() generates an event (the package uses a StreamController instance [129]) which is caught by the Logger's customizable handler's listener, as shown in Figure 38. When configuring a project's logger, a developer can set what log levels are displayed, as well as the log handler's body.

The developed log() method, shown in Figure 37, receives multiple arguments: "message" is fed into the previously specified logger to describe the system's state over time; while "measurement", "tagKey", "tagValue", "fieldKey" and "fieldValue" are the data being sent to the InfluxDB instance and refer to the different type of information necessary to create a valid data point. In the method's body, preparations to send the information to the InfluxDB instance are made by creating a Uniform Resource Identifier (URI) instance with the data's target information, followed by the request's header specification, which includes user authentication through the "Authorization" header. The data to be sent is then modified in the way required by the InfluxDB API and the HTTP POST request is sent using the previously specified URI, headers, and body (data). This request, due to being handled externally to the logging subsystem and returning a response with vital information, must be waited on. Once the response arrives, the status code received is accessed and, if the value is 204, then the request was sent and handled successfully, with the data point added to the InfluxDB instance's database.

5.1.1 RESTful interface

The logging subsystem's RESTful interface is comprised of a single controller with a single path to send requests to, as is show in Table 5. These sent requests must specify a variety of fields in its body which are then either sent to the InfluxDB instance or used by the standard output logger.

Request Path	Request Type	Request description
/log	POST	Request that allows for an application to request a logging action to happen, if given a valid input. If the action is successful, an "OK" message is returned in the "OPStatus" JSON field. Otherwise, "LOGGING_ERROR" is returned if the action could not be completed, for whatever reason.

Table 5 – Dart Angel Logging subsystem RESTful API Paths.

5.1.2 Package Specification

Due to having been developed using a Dart environment, the packages used suffered multiple changes and updates throughout development, just as in the Dart Angel CCM back-end server. The final packages and respective versions used, as well as the latest at the time of writing, are shown in Table 6.

Package	Version Used	Latest
angel3_framework **[98]**	4.1.1 (8th of July, 2021)	4.2.1 (4th of October, 2021)
http **[99]**	0.13.3 (3rd of May, 2021)	0.13.4 (4th of October, 2021)
logging **[100]**	1.0.1 (25th of March, 2021)	1.0.2 (14th of September, 2021)

Table 6 – Dart Angel logging subsystem package versions.

The angel3_framework package contains all the necessary angel dependencies to have the framework function correctly. There are additional add-on packages to further complement the base one, such as angel3_orm, yet they aren't essential to it functioning.

The http package allows for HTTP requests to be forwarded, thus being able to interact with RESTful APIs. It provides developers with high-level functions that act as wrappers for Dart's native options. Prior to migrating to Angel3, version 0.12.2 had to be used due to subsequent ones (>= 0.13.0) being incompatible with angel_framework (due to an incompatibility between http_parser versions), thus being outdated by four versions.

The logging package provides the programmer with a Logger that asynchronously handles logged information in any way implemented in its handler. Prior to migrating to Angel3, version 0.11.3 had to be used due to subsequent ones (>= 0.11.4) being incompatible with angel_framework, thus being outdated by three versions.

Prior to migrating from using a bridge package to communicate with the InfluxDB instance, the influxdb_client package [81] was used. Upon migrating to Angel3, it became unusable due to package incompatibility issues.

5.2 Data collection using a time series database (TSDB)

In the field of mathematics, a time series is a sequence of data points that happen successively in time. This temporal aspect being applied to the data allows for its evolution in time to be analysed and trends/patterns to be noticed through the construction of graphs with time as the independent variable [130].

When delving into programming, time series retain the previously described properties but also move into another field: data storage, through the usage of TSDBs. A TSDB [131] is a database optimized for handling timestamped data, such as metrics and events. This data can then be aggregated and manipulated in multiple ways to better gauge the situation at hand. TSDBs differ from conventional databases due to all stored data having an associated timestamp, following a specific lifecycle (data can be set to be deleted after a certain amount of time has passed since its insertion) and time series aware queries being able to be performed (such as querying data for specific time intervals).

Given InfluxDB's rising popularity, it was used to store metric data relative to back-end API and database operation timings, and then analysed through the usage of the Flux language [132]. Flux Open-Source Query Language, or simply Flux, is a data scripting language that allows developers to both code and query time series data simultaneously. Not only is it simple to learn due to its easy readability and syntax, but also highly modular, allowing for queries to be divided into sub-components and tested/used individually.

```
from(bucket:"example-bucket")
  |> range(start: -15m)
  |> filter(fn: (r) =>
    r._measurement == "cpu" and
    r._field == "usage_system" and
    r.cpu == "cpu-total"
  )
  |> yield()
```

Figure 39 – Code: querying InfluxDB using the Flux language [133].

In Figure 39, a Flux language query example is shown. In this example, a data source is initially set through the usage of the from() function which allows for InfluxDB instance specifications to be passed as arguments, such as the "host" or "org" arguments. In the given example, a "bucket" was

specified and passed the value of "example-bucket", meaning the bucket being used as a data source is the one with said name. After specifying the data source, the time range is set through the use of the range() function. Due to InfluxDB being a TSDB, data points have an associated timestamp and that can be used when creating queries. In the given example, the "start" argument was set to have the "-15m" value which means that only data points from the last 15 minutes (relative to when the query is run, are taken into account. The usage of the range() function, as well as the remaining ones, is prefaced with "|>" in the example. This is the pipe-forward operator and it allows for the output of a function to be used as input for another.

```
{foo: "bar", baz: 123.4, quz: -2}

{"Company Name": "ACME", "Street Address": "123 Main St.", id: 1123445}
```

Figure 40 – Flux record examples [134].

After specifying the time range, the example query goes on to filter data through the usage of the filter() function. This function iterates over each data point and creates Flux records, examples of which are shown in Figure 40. These records are enclosed by curly brackets, their properties are comma-delimited and each property is set as a key-value pair where keys are of the String type and values are of any. Key values may or may not be enclosed by double quotes, only requiring their usage when whitespace is present. Records are then passed into a predicate function as "r" and rows are evaluated against filters passed into the filter() function, dropping ones marked as "false" by the evaluation. In the case of the given example, three filters are present and it results in evaluating the value of three different columns per row of data.

Finally, data is returned through the usage of the yield() function. Flux automatically assumes the existence of a yield() function at the end of each script, only being required to use it when returning multiple query results.

5.2.1 InfluxDB communication

InfluxDB is a time series platform built with the intent of handling large volumes of timestamped information, provided by sources such as applications or sensors, to assist with monitoring and analysis. TSDBs have been on the rise and given InfluxDB's properties it was chosen as the way to store data relative to event occurrence time during the conducted study.

Rank			DBMS	Database Model	Score		
Sep 2021	Aug 2021	Sep 2020			Sep 2021	Aug 2021	Sep 2020
1.	1.	1.	InfluxDB	Time Series, Multi-model	29.50	-0.06	+6.17
2.	2.	2.	Kdb+	Time Series, Multi-model	8.13	+0.14	+0.70
3.	3.	3.	Prometheus	Time Series	6.43	+0.24	+0.74

Figure 41 – DB Engines time series DBMS (Database Management System) ranking [135].

The main reason TSDBs have been growing in popularity is the rise of InfluxDB. In September 2021, InfluxDB held spot #1 for TSDB management systems, or time series DBMS, with a score of 29.50 which is 362.85% of the score value held by spot #2, Kdb+, as shown in Figure 41. Additionally, InfluxDB held spot #28 in the overall DBMS category, led by Oracle and MySQL with scores of 1271.55 and 1212.52, respectively, as shown in Figure 42.

Rank			DBMS	Database Model	Score		
Sep 2021	Aug 2021	Sep 2020			Sep 2021	Aug 2021	Sep 2020
1.	1.	1.	Oracle	Relational, Multi-model	1271.55	+2.29	-97.82
2.	2.	2.	MySQL	Relational, Multi-model	1212.52	-25.69	-51.72
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	970.85	-2.50	-91.91
4.	4.	4.	PostgreSQL	Relational, Multi-model	577.50	+0.45	+35.22
5.	5.	5.	MongoDB	Document, Multi-model	496.50	-0.04	+50.02
6.	6.	↑ 7.	Redis	Key-value, Multi-model	171.94	+2.05	+20.08
7.	7.	↓ 6.	IBM Db2	Relational, Multi-model	166.56	+1.09	+5.32
8.	8.	8.	Elasticsearch	Search engine, Multi-model	160.24	+3.16	+9.74
9.	9.	9.	SQLite	Relational	128.65	-1.16	+1.98
10.	↑ 11.	10.	Cassandra	Wide column	118.99	+5.33	-0.18
28.	28.	↑ 29.	InfluxDB	Time Series, Multi-model	29.50	-0.06	+6.17

Figure 42 – DB Engines overall DBMS ranking [136].

Dart is lacking when it comes to logging solutions, which is why InfluxDB was chosen for continuous metric collection and analysis in conjunction with the standard application output to mimic actual state-of-the-art logging systems in terms of function.

As shown in Figure 26 (in the "CCM (Crypto Currency Manager) implementations)" section) the InfluxDB instance is located within the conceptual logging system, only being directly communicated with by the logging system's Dart Angel subsystem. This subsystem sends the information via an HTTP request after receiving it from either one of the CCM back-ends or a front-end application.

There are a variety of ways to write data into an InfluxDB instance:

- Telegraf [137], InfluxData's open-source server agent for metric collection, bolsters a wide array of available plugins to use for collecting data from a variety of different sources.
- InfluxDB scrapers [138] periodically collect data from HTTP/HTTPS-accessible endpoints with data returned in the Prometheus data format.
- Influx CLI [139], or Influx command-line interface, is an interactive shell that allows to write (manual/file input), view and query data in an InfluxDB instance through its HTTP API.
- Influx API [140], which allows users to send HTTP requests to a variety of endpoints to conduct the needed operations directly.

During the conducted study, third-party bridge libraries/packages were initially used. These would work as wrappers for diverse actions, reducing bloat code when sending requests to the Influx API. Eventually, not only due to incompatibility issues but also to decrease the number of possible variables in the study, the bridge libraries/packages were replaced by direct API communication.

Element	Optional/Required	Description	Type (See data types for more information.)
Measurement	Required	The measurement name. InfluxDB accepts one measurement per point.	String
Tag set	Optional	All tag key-value pairs for the point.	Tag keys and tag values are both strings.
Field set	Required. Points must have at least one field.	All field key-value pairs for the point.	Field keys are strings. Field values can be floats, integers, strings, or Booleans.
Timestamp	Optional. InfluxDB uses the server's local nanosecond timestamp in UTC if the timestamp is not included with the point.	The timestamp for the data point. InfluxDB accepts one timestamp per point.	Unix nanosecond timestamp. Specify alternative precisions with the InfluxDB API.

Figure 43 – InfluxDB data syntax description [141].

Despite Telegraf and InfluxDB scrapers being a viable, and even convenient, choice for the study from a development perspective, they would negatively impact the data being stored in the InfluxDB instance as the requests sent to the logging system's Dart Angel subsystem also contain data relevant to the standard output portion of the system, rather than the InfluxDB instance. Due to this, only approaches that would allow for the received data to be manipulated prior to being stored were considered. Finally, due to being a command-line interface and not relevant to the conducted study, Influx CLI wasn't used.

To write information to an InfluxDB instance through its HTTP API, the first step is knowing the format the data is required to be sent in: the InfluxDB line protocol [142]. The InfluxDB line protocol is a simple text-based format used to write data points into an InfluxDB instance and an example of its structure is shown in Figure 37. The "data" String object shown begins with the measurement name, a mandatory argument. It is then instantly followed by a comma and a tag set, an optional argument. After the tag set, a single space is used before a field set is specified, this being another mandatory argument. Both the tag sets and field sets used must follow the "key=value" structure, as shown in the given example. Finally, despite not used in the example, a timestamp can be specified after the last field set, leaving a space between them. By not specifying a timestamp for a sent data point, the InfluxDB instance adds one using the current machine's time in UTC, or Universal Time Coordinated. A description of all the line protocol syntax is shown in Figure 43.

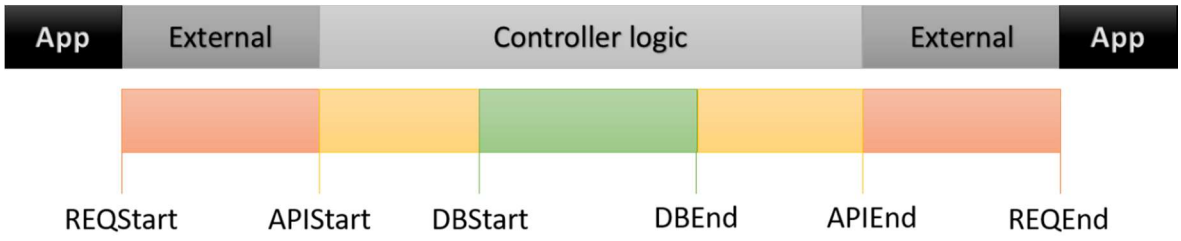

Figure 44 – Full request time scope and event logging instances.

In the conducted study, the measurement argument was used to specify the type of system sending the data, such as "SC4ALLAngel" for data points originating from the Dart Angel CCM, "SC4ALLSpring" for data points originating from the Spring Boot CCM, and "SC4ALLFlutter" for data points originating from the web/mobile Flutter applications; a tag key argument was used to specify the data point source, which was used to aggregate data points depending on their origin/test case; a tag value argument was used to specify the type of event the data point corresponds to, these being "REQStart" for when the mobile/web Flutter application sends a request, "APIStart" for when the CCM request handler's body begins, "DBStart" for when the CCM data-related operations begin, "DBEnd" for when the CCM data-related operations finish, "APIEnd" for when the CCM request

handler's body finishes, and "REQEnd" for when the mobile/web Flutter application receives the request response (all event types are shown in Figure 44); a field key argument was used to specify a request's ID, which is used to aggregate data from the same request; and a field value argument was used to specify the timestamp at which the given event occurred in the system itself. No direct timestamp value was used, making it so InfluxDB handles it.

6 Results and discussion

The study's main goal was to assess Dart's suitability to be used to develop back-end servers. Two evaluations were performed in a production environment:

- Based on CCM (Crypto Currency Manager) system performance using timing logs to assess both system – the Dart CCM and the Spring Boot CCM, the reference.
- Subjective from the developer perspective, namely system development considerations and opinions regarding the development process.

6.1 Performance results

Performance results for CCM systems were obtained using timing data collected related to requests sent from either the developed front-end applications or external tools to the CCM back-end servers and stored in an InfluxDB (Influx Database) instance. This data was then treated using Flux language code to achieve the desired type of resulting information.

6.1.1 Evaluation deployment

To compare and evaluate both the Dart and Java-built back-end CCM servers we avoided the use of the development/testing environment – supported on the same machine without any virtualization nor isolation, where sharing system resources could naturally affect overall performance.

We used Docker [143] containers to run an NginX [119] instance, both CCM implementations and the logging system (Dart Angel logging subsystem and the InfluxDB instance) in five individual containers on the same machine (a machine being used for the sole purpose of hosting the servers), achieving system isolation guaranteeing that each part of the overall running system was only using the resources available to its own container. The front-end applications were run on a separate machine in the same network and all tests were conducted under the same conditions.

NginX was used as reverse proxy mainly to address CORS-related (Cross-Origin Resource Sharing-related) issues described earlier.

Some choices were made while developing the overall system to increase evaluation fairness:

- No security measures being implemented when it comes to storage/communication (no encryption/decryption of data).
- CCM instances shared the same base scenario, providing external users with the same type of endpoints that perform the same operations.
- CCM instances shared the same database type, SQLite [92], with the same table specifications and communicating with it using the same queries.

- All logging was performed through HTTP requests to the same logging system endpoint.
- NginX was used to provide an equal form of routing to both CCM implementations.
- Docker containers were used to minimize explicit dependences and resource sharing.

6.1.2 Data gathering logic

The main objective was to compare two systems using timing metrics related to relevant HTTP requests. The scope of time relative to an HTTP request being handled begins when the request is initially sent and ends when the request's reply reaches the request's sender. Both instances of time occur outside the scope of the developed CCM back-end servers, meaning that the front-end applications developed also had to be capable of sending requests to the developed logging system for metric gathering. Gathering data from both time instances allowed for the total time taken per request to be recorded, yet other timing data had to be considered such as the time taken within the CCM themselves, in specific for data-related operations. As such, both CCM implementations also logged timing data relative to when a request first entered the controller body, when internal data operations, such as database access, started, when they finished, and when the reply to the received request was about to be returned by the controller. These additional metrics allowed for calculations to be performed relative to the time that the CCM implementations took to handle requests when removing the added time from database operations, which were not the focus of this study.

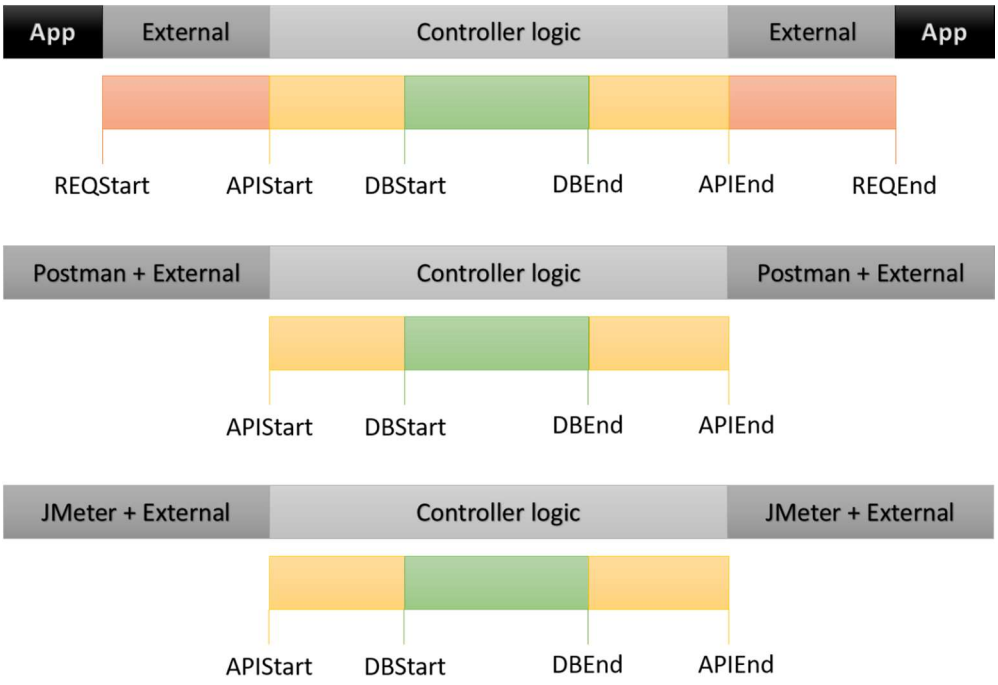

Figure 45 – Full request time scope and event logging instances for each test case: using the App, Postman and JMeter.

The considerations regarding request time scopes explained above allowed for a testing case to be carried out where 1000 copies of the same request, a simple HTTP GET request, which always returned the same response, were sent by the front-end application to both CCM implementations. The goal of these tests was to get timing data relative to the total request time, particularly timings related to when the request is outside of a CCM controller body. Additional tests were thereafter conducted, yet did not rely on the front-end applications developed, instead turning to Postman [144] and JMeter [145], API (Application Programming Interface) load testing tools.

Postman allows for request collections to be created and then run multiple times in succession. As such, two collections, one for each CCM, containing all ten endpoints that both CCM implementations provide users with were created in such a way that replies were always identical in results during tests. These collections were run 100 times (for a total of 1000 requests being sent) and results from running them were compared to those obtained from the first testing case.

JMeter goes a step further and allows for concurrent calls to be performed through the usage of various threads. The final test case involved running the same type of request run in the first case another 1000 times, but this time through JMeter and using five threads at once. This was done to compare how each CCM implementation handled concurrent requests being received and the impact on the average time each request would take to complete.

The metrics collected in each test case are shown in Figure 45.

6.1.3 The usage of Flux Open-Source Query Language

Flux Open-Source Query Language [132], or simply Flux, is a data scripting language that allows developers to both code and query time series data simultaneously. It is the main way to treat data stored in an InfluxDB database through the InfluxDB web user interface (UI) and, due to its convenience, was opted as the way to do so during this study.

Flux allows for scripts to be created to efficiently treat data present not only in typical programming variable types such as integers and floats but also tables. Due to data stored into InfluxDB being sent as data points following a specific structure, a group of data points can be considered a group of rows for the same data table. As such, all operations performed during this study using the Flux language revolved around using tables, mainly manipulating column data.

```
api_start = from(bucket: "SC4ALL")
  |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
  |> filter(fn: (r) => r["_measurement"] == "SC4ALLAngel")
  |> filter(fn: (r) => r["Postman"] == "APIStart")
  |> drop(columns: ["_time", "_start", "_stop", "_measurement", "Postman"])
```

Figure 46 – Code: obtaining data relative to when requests start being handled by an Angel CCM API controller in the Postman testing case using Flux.

A total of six scripts were written to treat the collected data, two for each of the three testing cases (one for the Angel CCM and another for the Spring Boot CCM). Every script initially selected the timing data relative to when requests both started and ended being handled by an API controller ("APIStart" and "APIEnd"), relative to when data related operations started and ended ("DBStart" and "DBEnd"), and, in the case of the Flutter application testing case, relative to when a request was initially sent and when the response was received ("REQStart" and "REQEnd"). The Flux code used to obtain data relative to when requests started being handled by an Angel CCM API controller in the Postman testing case is shown in Figure 46 as an example. To obtain data from the Spring Boot CCM the reference to "SC4ALLAngel" was changed to "SC4ALLSpring" and to obtain from the Flutter application, for the "REQStart" and "REQEnd" metrics rather than the shown "APIStart", it was changed to "SC4ALLFlutter". The reference to "Postman" was changed to "JMeter" for the JMeter testing case and to "FlutterApp" for the Flutter application one. Finally, every column that would no longer be used was dropped, as shown in the last line of code in the figure, resulting in a table that simply contained the request id and relevant timestamp columns.

```
api_data = join(tables: {d1: api_start, d2: api_end}, on: ["_field"])
  |> map(fn: (r) => ({
      id: r._field,
      start: r._value_d1,
      end: r._value_d2,
      duration: r._value_d2 - r._value_d1
    })
  )
```

Figure 47 – Code: obtaining a table with all data relative to API controller timings using Flux.

Once all individual metrics were obtained, they could start being grouped up for treatment. This was done by joining tables, as shown in Figure 47. The join() function is used to join two tables and one of its uses in the conducted study was to group up tables relative to the start and end of each time scope (the API time scope, DB time scope and, in the case of the application testing case, the REQ

time scope). The tables were joined on the "_field" column, which is where the request id value was stored, meaning that data joined together was relative to the same handled request. Then, having the data relative to the start and end of a time scope present in the same table, its duration could be calculated via a simple subtraction between the "_value_d2" and "_value_d1" columns which pertained to the "_value" columns present in the join() function's d1 and d2 tables, the start and end tables which contained the request start and end timestamps, respectively.

```
full_data = join(tables: {d1: api_data, d2: db_data}, on: ["id"])
  |> map(fn: (r) => ({
      id: r.id,
      api_duration: r.duration_d1,
      db_duration: r.duration_d2,
      api_dur_no_db: r.duration_d1 - r.duration_d2
    })
  )
  |> map(fn:(r) => ({ r with api_duration: float(v: r.api_duration) }))
  |> map(fn:(r) => ({ r with db_duration: float(v: r.db_duration) }))
  |> map(fn:(r) => ({ r with api_dur_no_db: float(v: r.api_dur_no_db) }))
  |> yield(name: "full_data")
```

Figure 48 – Code: obtaining the result table using Flux.

Having calculated the durations needed, the resulting tables were joined once more to obtain the results table. Figure 48 shows how this results table was obtained in both the Postman and JMeter testing cases, with the application case requiring an additional table joining. At this point, additional relevant results were calculated, such as the duration relative to requests being handled by a controller without the impact of data related operations and, in the case of the application testing case, the duration relative to the entire request's time scope without the impact of API and data related operations (time taken while outside an API controller's scope). The results were then cast to float values using the float() function so that they could be used in the InfluxDB UI histogram tool [146], and the table was set to be returned at the end of the script via the use of the yield() function.

```
mean_api = full_data
  |> mean(column: "api_duration")
  |> yield(name: "api_duration")
```

Figure 49 – Code: obtaining the API-related duration mean using Flux.

Finally, the mean value for each of the final columns, excluding the id column, was calculated using the mean() function, as exemplified in Figure 49 for the API duration mean calculation. All mean values were also set to be returned at the end of the script via the yield() function.

6.1.4 Results

The results obtained for each testing case will be presented in this section using images pertaining to histograms obtained from using the InfluxDB UI's histogram tool. Some images were cropped to assist visualization and information relevant to the shown histograms, such as meaningful histogram bucket intervals and their respective request values, will also be presented. The X axis of each histogram denotes duration values in milliseconds (some interval values mentioned are approximations due to the tool's limitations) while the Y axis denotes the number of requests, from the 1000 sent in each case, that fell within a given duration interval.

6.1.4.1 General result overview

Given the abundance of results, mean values will be shown throughout this section in an organized manner to allow for a better comparative overview.

6.1.4.1.1 Flutter application case

	Dart Angel CCM (ms)	Spring Boot CCM (ms)
Data-related operation duration	0.757	23.298
Full API duration	1.71	33.332
API duration without the impact from data-related operations	0.953	10.034
Full request duration	273.941	142.925
Full request duration without the impact from API or data-related operations	272.231	109.593

Table 7 – App: Mean duration values for CCM back-end servers.

For data-related operation durations, the Angel Dart CCM took approximately 3.25% of the Spring Boot CCM's time to complete; for full API durations, it took 5.13% of the Spring Boot CCM's time to complete; and for the API duration without the impact from data-related operations, it took approximately 9.50% of the Spring Boot CCM's time to complete.

For full request durations, the Spring Boot CCM took approximately 52.17% of the Dart Angel CCM's time to complete, and for full request durations without the impact from API or data-related operations it took approximately 40.25% of the Dart Angel CCM's time to complete.

Refer to Table 7 for all mean values pertaining to the Flutter application case.

6.1.4.1.2 Postman case

	Dart Angel CCM (ms)	Spring Boot CCM (ms)
Data-related operation duration	6.627	31.921
Full API duration	7.413	42.752
API duration without the impact from data-related operations	0.786	10.831

Table 8 – Postman: mean duration values for CCM back-end servers.

For data-related operation durations, the Angel Dart CCM took approximately 20.76% of the Spring Boot CCM's time to complete; for full API durations, it took 17.34% of the Spring Boot CCM's time to complete; and for the API duration without the impact from data-related operations, it took approximately 7.26% of the Spring Boot CCM's time to complete.

Refer to Table 8 for mean values pertaining to the Postman case.

6.1.4.1.3 JMeter case

	Dart Angel CCM (ms)	Spring Boot CCM (ms)
Data-related operation duration	0.49	38.855
Full API duration	1.095	60.623
API duration without the impact from data-related operations	0.605	21.768
JMeter completion time	~ 9000	~ 20000

Table 9 – JMeter: mean duration values for CCM back-end servers.

For data-related operation durations, the Angel Dart CCM took approximately 1.26% of the Spring Boot CCM's time to complete; for full API durations, it took 1.81% of the Spring Boot CCM's time to complete; and for the API duration without the impact from data-related operations, it took approximately 2.78% of the Spring Boot CCM's time to complete.

Despite JMeter's completion time in the result tree only displaying the elapsed seconds, the difference in times between CCM implementations was still apparent with the Dart Angel CCM taking approximately 45% of the time the Spring Boot CCM took to complete.

Refer to Table 9 for mean values pertaining to the JMeter case.

6.1.4.2 Flutter Application case results

In this testing case, requests originated from a Flutter application and were sent to the NginX reverse proxy. They were then forwarded to the at the time active CCM implementation which served the request and sent a reply through the inverse path.

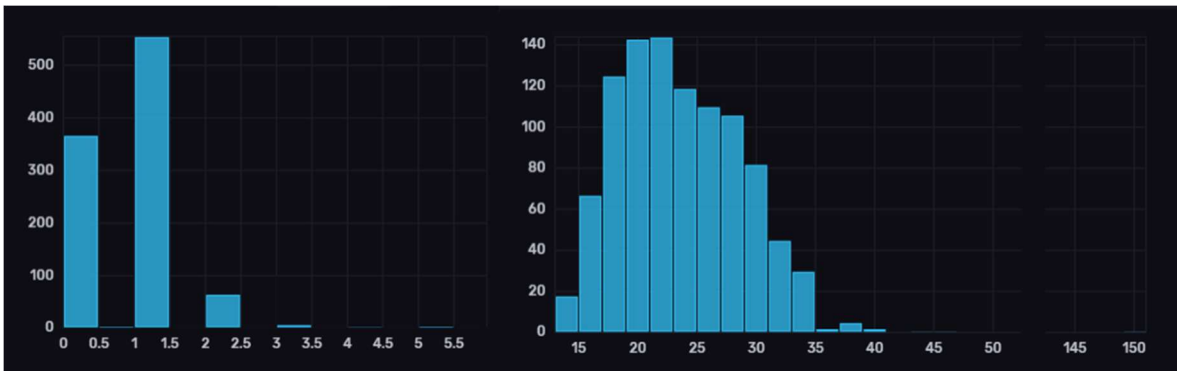

Figure 50 – App: Data-related operation durations (milliseconds) - Dart (left) and Spring Boot (right).

Dart Angel CCM data-related operation durations (Figure 50 - left) were very concentrated near the start of the spectrum. The interval with the most requests was [1, 1.5[with 55.5% of the totality, followed by [0, 0.5[with 36.7% and [2, 2.5[with 6.5%. The remaining intervals having an extremely low number of requests. The mean duration was 0.757 milliseconds.

Spring Boot CCM data-related operation durations (Figure 50 - right) resulted in a bell-shaped histogram peaking on the [19, 21[and [21, 23[intervals with 14.3% and 14.4% of the request totality, respectively. Other intervals of note were [17, 19[with 12.5% of the totality, [23, 25[with 11.9%, [25, 27[with 11.0% and [27, 29[with 10.6%. The mean duration was 23.298 milliseconds.

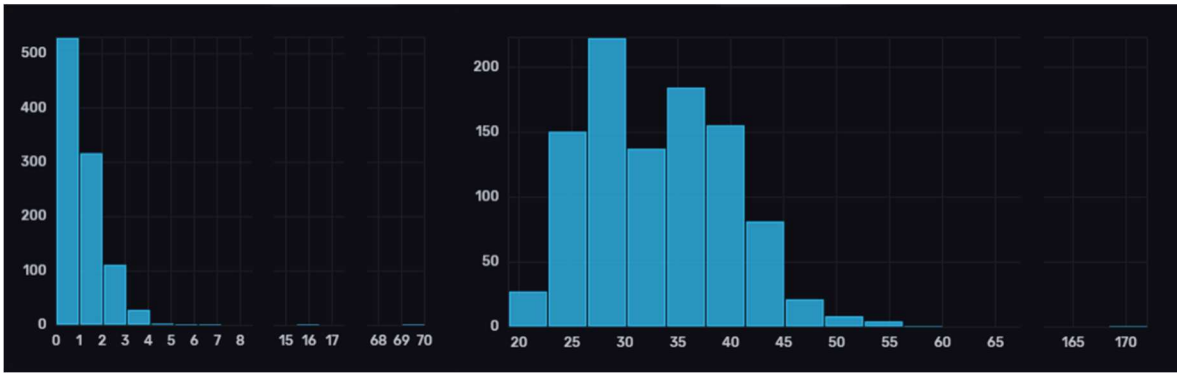

Figure 51 – App: Full API durations (milliseconds) - Dart (left) and Spring Boot (right).

Full Dart Angel CCM API durations (Figure 51 - left) were very concentrated at the start of the spectrum, decreasing as intervals progressed and resulting in a positively skewed histogram. Most requests fell into the [0, 1[interval leading into it stacking up to 53.0% of the request totality, followed by [1, 2[interval's 31.8%, [2, 3[interval's 11.3% and [3, 4[interval's 3.0%. The mean duration was 1.71 milliseconds.

Full Spring Boot CCM API durations (Figure 51 - right) resulted in a bimodal shaped histogram with both peaks achieved in the [26, 30[interval, the highest, with 22.3% of the request totality and in the [33, 37[interval with 18.5%. Other intervals of note were [22, 26[with 15.1% of the totality, [30, 33[with 13.8%, [37, 41[with 15.6% and [41, 45[with 8.2%. The mean duration was 33.332 milliseconds.

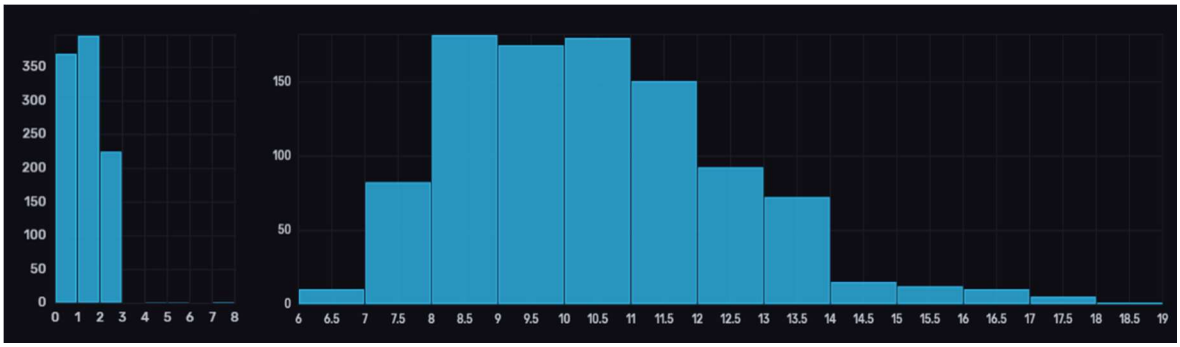

Figure 52 – App: API durations without the impact from data-related operations (milliseconds) - Dart (left) and Spring Boot (right).

Dart Angel CCM API duration without the impact from data-related operations (Figure 52 - left) were very concentrated at the start of the spectrum. The interval with the most requests was [1, 2[with 39.8% of the totality, followed by [0, 1[with 37.1% and [2, 3[with 22.6%. The mean duration was 0.953 milliseconds.

Spring Boot CCM API durations without the impact from data-related operations (Figure 52 - right) resulted in a bell-shaped histogram, behaving like the data-related operation duration results, peaking on the [8, 9[, [9, 10[and [10, 11[intervals with 18.1%, 17.5% and 18.0% of the request totality, respectively. Other intervals of note were [7, 8[with 8.3% of the totality, [11, 12[with 15.1%, [12, 13[with 9.3% and [13, 14[with 7.3%. The mean duration was 10.034 milliseconds.

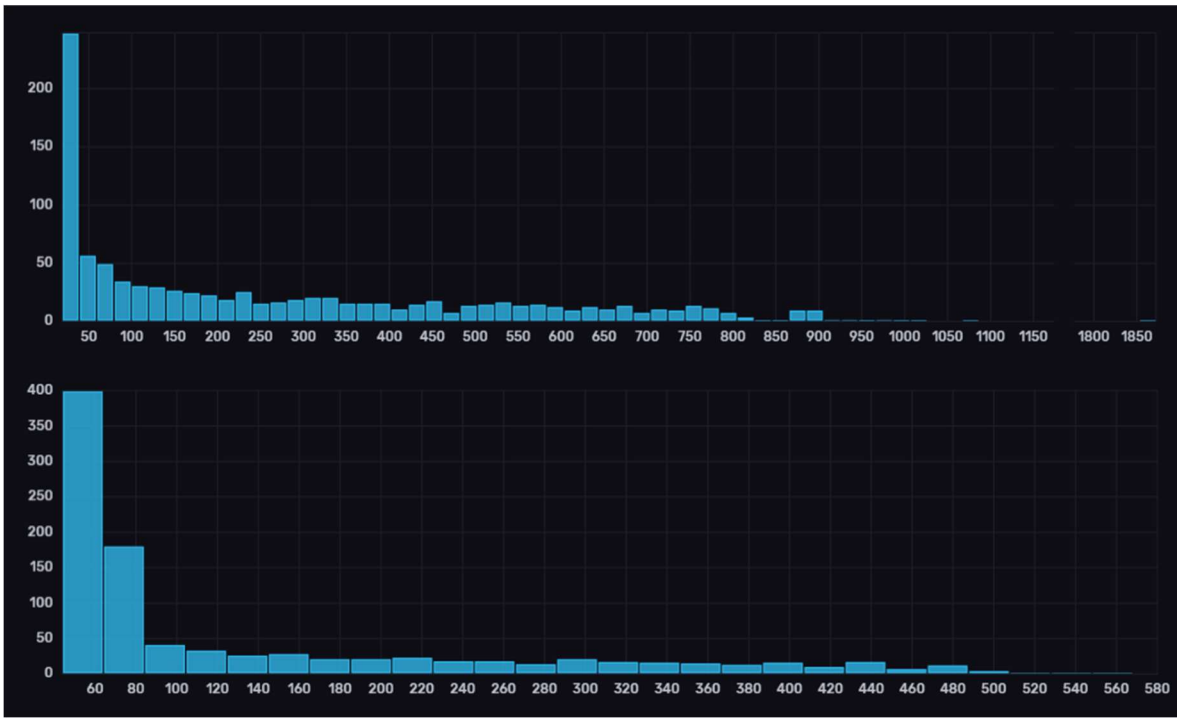

Figure 53 – App: Full request durations (milliseconds) - Dart (top) and Spring Boot (bottom).

Full Dart Angel CCM API request durations (Figure 53 - top) were moderately concentrated in the [19, 39[interval having 24.8% of the request totality with the remaining requests trailing this peak, resulting in a positively skewed histogram. The remaining 75.2% were distributed in a similar way through the remaining spectrum, all the way to around the 900-millisecond mark, with the most concentrated intervals being the ones after the first mentioned in this paragraph: [39, 59[with 5.7%, [59, 79[with 5.0% and [79, 99[with 3.5%. The mean duration was 273.941 milliseconds.

Full Spring Boot CCM request durations (Figure 53 - bottom) were very concentrated at the start of the spectrum with the [44, 64[interval, the peak, having 40.0% of the request totality and [64, 84[having 18.1% with the remaining requests trailing the peak, resulting in a positively skewed histogram. The remaining 41.9% of requests were distributed in a similar way through the remaining spectrum, all the way to around the 500-millisecond mark. The mean duration was 142.925 milliseconds.

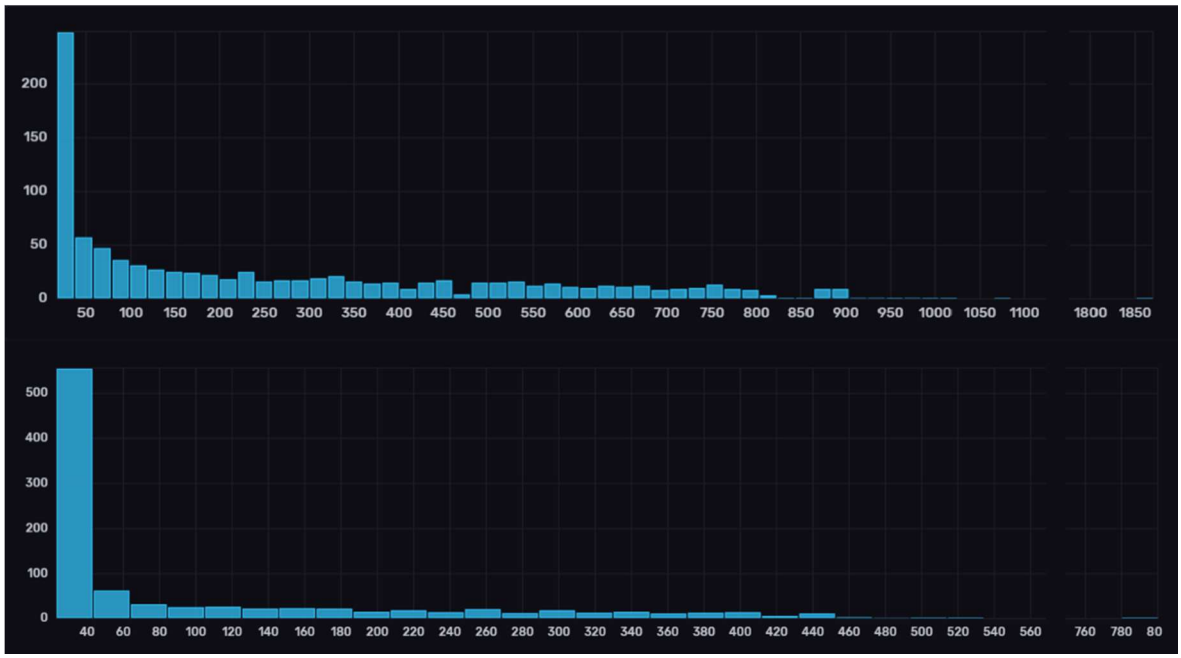

Figure 54 – App: Request durations without the impact from API and data-related operations (milliseconds) - Dart (top) and Spring Boot (bottom).

Dart Angel CCM request durations without the impact from API and data-related operations (Figure 54 - top) were moderately concentrated in the [18, 38[interval having 24.9% of the request totality with the remaining requests trailing this peak, resulting in a positively skewed histogram. The remaining 75.1% were distributed in a similar way through the remaining spectrum, all the way to around the 900-millisecond mark, behaving like the full request duration results. Following the previously mentioned interval, the most concentrated were [38, 58[with 5.8% of the totality, [58, 78[with 4.8% and [78, 98[with 3.7%. The mean duration was 272.231 milliseconds.

Spring Boot CCM request durations without the impact from API and data-related operations (Figure 54 - bottom) were very concentrated in the [23, 43[interval having 55.6% of the request totality with the remaining requests trailing the peak, resulting in a positively skewed histogram. The remaining 44.4% of requests were distributed in a similar way through the remaining spectrum, with only the [43, 63[interval standing out with 6.4% of the request totality. The duration mean was 109.593 milliseconds.

6.1.4.3 Postman case results

In this testing case, requests originated from Postman and were sent to the NginX reverse proxy. They were then forwarded to the at the time active CCM implementation which served the request and sent a reply through the inverse path.

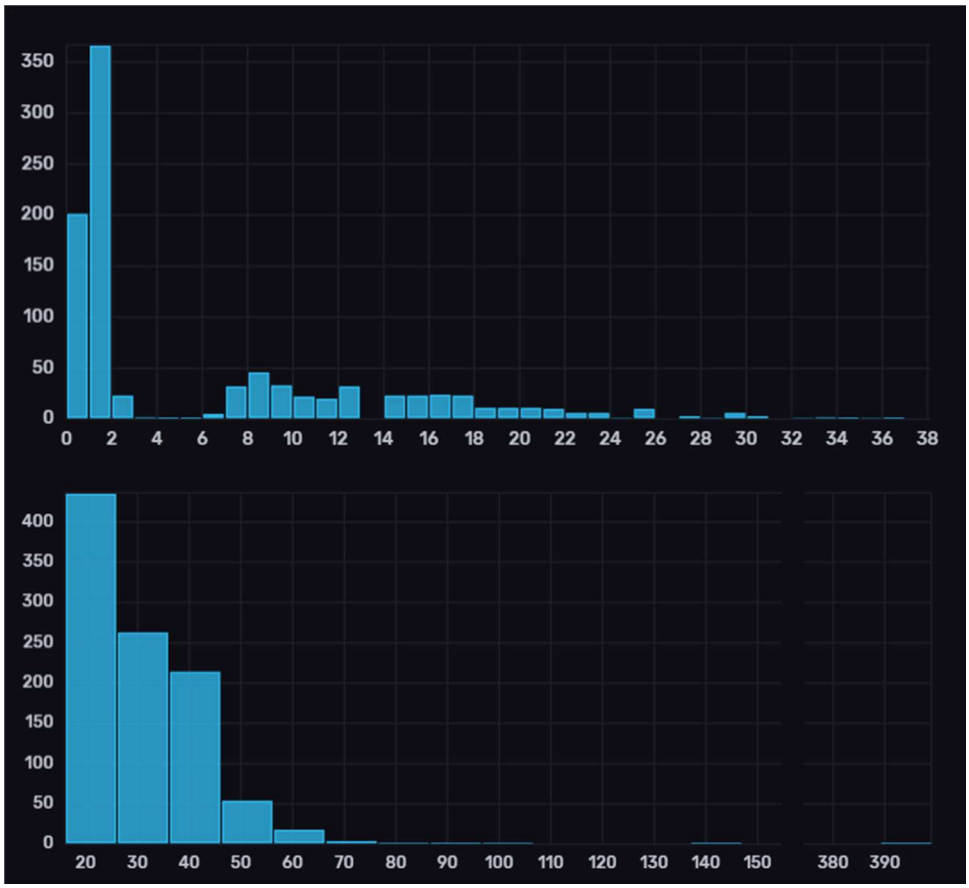

Figure 55 – Postman: Data-related operation durations (milliseconds) - Dart (top) and Spring Boot (bottom).

Dart Angel CCM data-related operation durations (Figure 55 - top) were very concentrated at the start of the spectrum with the [1, 2[interval, the peak, having 36.7% of the request totality and [0, 1[having 20.2%. The remaining 43.1% were distributed somewhat randomly throughout the rest of the spectrum. The mean duration was 6.627 milliseconds.

Spring Boot CCM data-related operation durations (Figure 55 - bottom) were very concentrated near the start of the spectrum, decreasing as intervals progressed and resulting in a positively skewed histogram. Most requests fell into the [16, 26[interval leading into it stacking up to 43.6% of the request totality, followed by [26, 36[interval's 26.4%, [36, 46[interval's 21.5% and [46, 56[interval's 5.5%. The mean duration was 31.921 milliseconds.

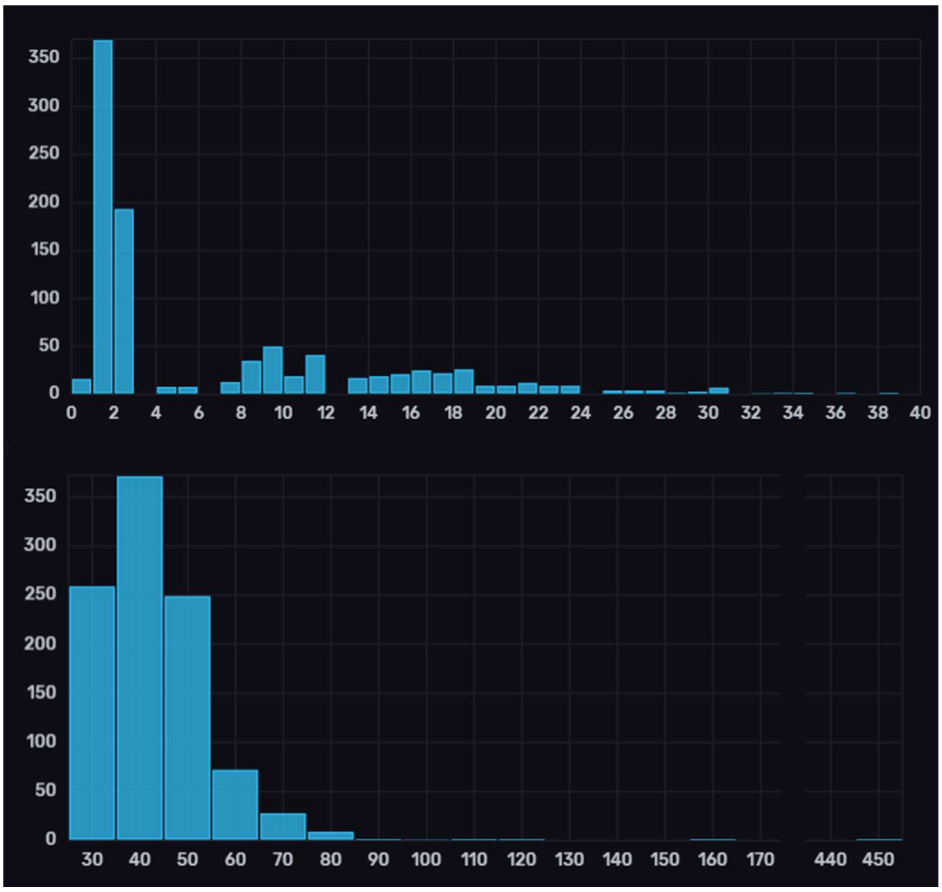

Figure 56 – Postman: Full API durations (milliseconds) - Dart (top) and Spring Boot (bottom).

Full Dart Angel CCM API durations (Figure 56 - top) were very concentrated near the start of the spectrum. The interval with the most requests was [1, 2[with 37.0% of the totality, followed by [2, 3[with 19.4%. The remaining 43.6% were distributed somewhat randomly throughout the rest of the spectrum. The mean duration was 7.413 milliseconds.

Full Spring Boot CCM API durations (Figure 56 - bottom) were very concentrated near the start of the spectrum. The interval with the most requests was [35, 45[with 37.2% of the totality, followed by [25, 35[with 26.0%, [45, 55[with 25.0%, [55, 65[with 7.3% and [65, 75[with 2.9%. The mean duration was 42.752 milliseconds.

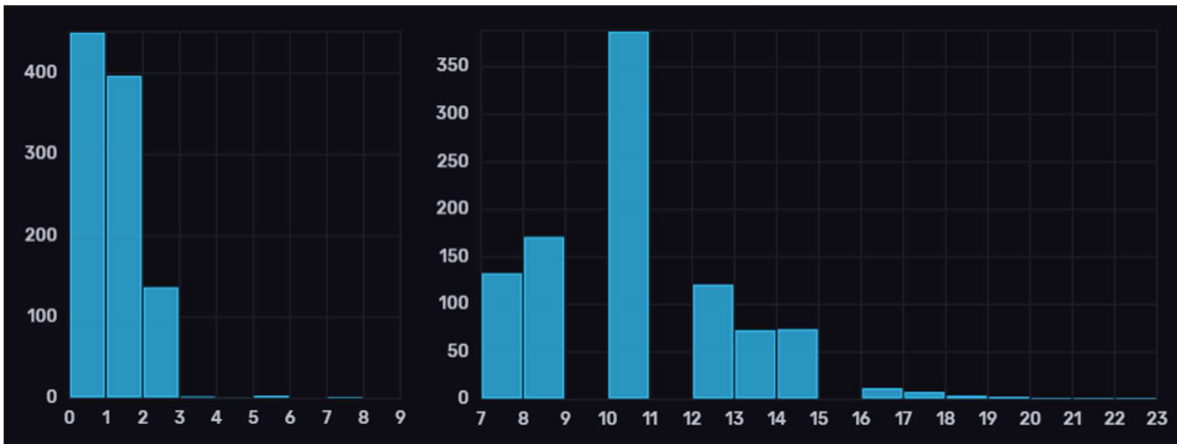

Figure 57 – Postman: API durations without the impact from data-related operations (milliseconds) - Dart (left) and Spring Boot (right).

Dart Angel CCM API durations without the impact from data-related operations (Figure 57 - left) were very concentrated at the start of the spectrum, decreasing as intervals progressed and resulting in a positively skewed histogram. Most requests fell into the [0, 1[interval leading into it stacking up to 45.1% of the request totality, followed by [1, 2[interval's 39.8% and [2, 3[interval's 13.8%. The mean duration was 0.786 milliseconds.

Spring Boot CCM API durations without the impact from data-related operations (Figure 57 - right) were moderately concentrated in the [10, 11[interval with 38.8% of the request totality. The remaining 61.2% were distributed somewhat randomly throughout the rest of the spectrum with intervals of note being [7, 8[with 13.4% of totality, [8, 9[with 17.2%, [12, 13[with 12.2%, [13, 14[with 7.4% and [14, 15[with 7.5%. The mean duration was 10.831 milliseconds.

6.1.4.4 JMeter case results

In this testing case, requests originated from JMeter and were sent to the NginX reverse proxy. They were then forwarded to the at the time active CCM implementation which served the request and sent a reply through the inverse path.

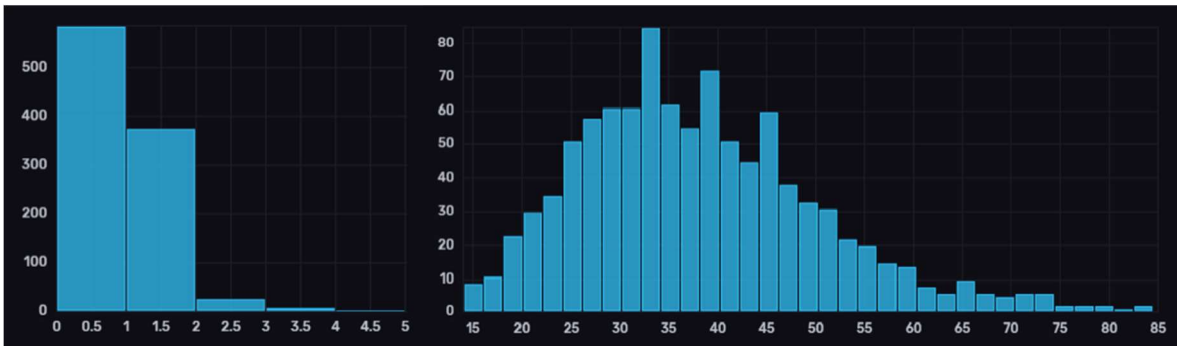

Figure 58 – JMeter: Data-related operation durations (milliseconds) - Dart (left) and Spring Boot (right).

Dart Angel CCM data-related operation durations (Figure 58 - left) were very concentrated at the start of the spectrum, decreasing as intervals progressed and resulting in a positively skewed histogram. Most requests fell into the [0, 1[interval leading into it stacking up to 58.6% of the request totality, followed by [1, 2[interval's 37.6%, [2, 3[interval's 2.7% and [3, 4[interval's 0.9%. The mean duration was 0.49 milliseconds.

Spring Boot CCM data-related operation durations (Figure 58 - right) resulted in a bell-shaped histogram peaking on the [32, 34[interval with 8.5% of the request totality. The remaining intervals around the top of the bell shape ranged from 4.5% to 7.2% of the totality. The mean duration was 38.855 milliseconds.

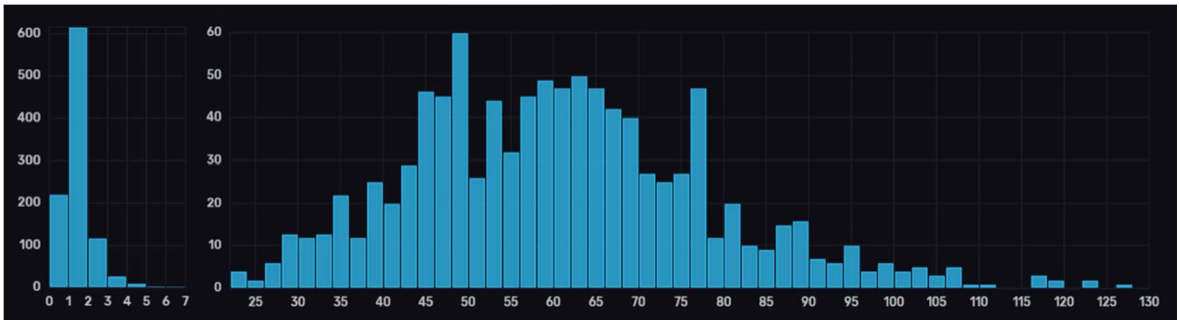

Figure 59 – JMeter: Full API durations (milliseconds) - Dart (left) and Spring Boot (right).

Full Dart Angel CCM API durations (Figure 59 - left) were very concentrated at the start of the spectrum. The interval with the most requests was [1, 2[with 61.6% of the totality, followed by [0, 1[with 22.1%, [2, 3[with 11.8% and [3, 4[with 2.8%. The mean duration was 1.095 milliseconds.

Full Spring Boot CCM API durations (Figure 59 - right) resulted in a randomly shaped histogram which seemed to be approaching a bell shape. The histogram peaked at the [48, 50[interval having 6.0% of the request totality with the remaining high-end intervals ranging from 4.4% to 5.0% in value. The mean duration was 60.623 milliseconds.

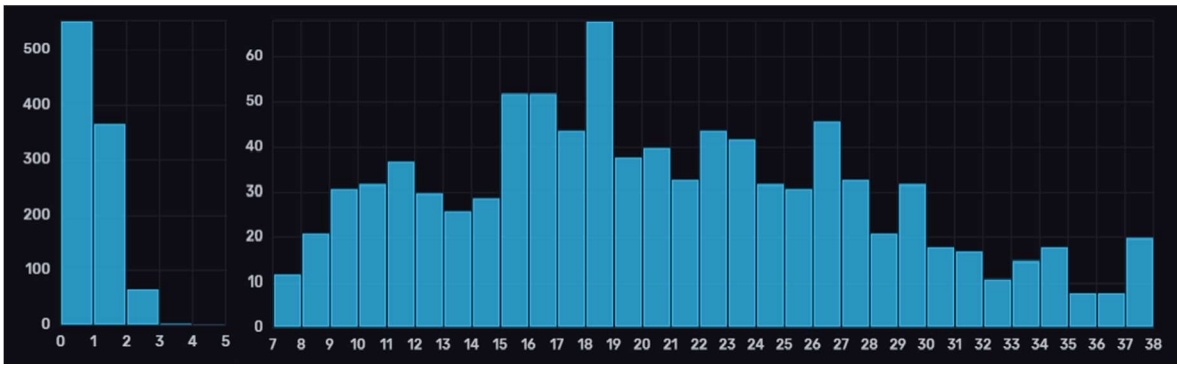

Figure 60 – JMeter: API durations without the impact from data-related operations (milliseconds): Dart (left) and Spring Boot (right).

Dart Angel CCM API durations without the impact from data-related operations (Figure 60 - left) were very concentrated at the start of the spectrum, decreasing as intervals progressed and resulting in a positively skewed histogram. Most requests fell into the [0, 1[interval leading into it stacking up to 55.4% of the request totality, followed by [1, 2[interval's 36.8% and [2, 3[interval's 6.7%. The mean duration was 0.605 milliseconds. It is also worth noting that the five threads used in JMeter to execute the 1000 requests took roughly nine seconds to complete the assigned task.

Spring Boot CCM API durations without the impact from data-related operations (Figure 60 - right) resulted in a randomly shaped histogram peaking at [18, 19[with 6.8% of the request totality. Other intervals of note were [15, 16[and [16, 17[both with 5.2% of the totality. The mean duration was 21.768 milliseconds. It is also worth noting that the five threads used in JMeter to execute the 1000 requests took roughly 20 seconds to complete the assigned task.

6.1.5 Result Discussion

In this section, the obtained results will be discussed, such as the meaning behind the obtained histogram shapes, what they imply about the obtained data and what could have been the influence, as well as the obtained means and the possible causes for the disparity between CCM implementations.

6.1.5.1 Dart Angel CCM – Flutter application case

The three histograms pertaining to when a request was within a controller's scope saw most of its concentration at the start of the spectrum. In terms of data-related operation durations they were expected to be low due to, in the Flutter application case, the requests performed only resulting in a map access to return a list of data. In terms of the full API durations, they seem to have been highly impacted by data-related operations due to them comprising approximately 44.27% of the duration.

Despite this, values were still much lower than the expected, mainly due to the API duration without the impact from data-related operation being so low, at a mere 0.953 milliseconds.

```
int endTimeDB = DateTime.now().millisecondsSinceEpoch;

LogHandler.log('E#$id POST: /crypto/exchange/purchase', 'SC4ALLAngel',
    req.headers['source'].first, 'DBEnd', id.toString(), endTimeDB);

int endTimeAPI = DateTime.now().millisecondsSinceEpoch;
```

Figure 61 – Code: logging between two timing event instances in the Dart Angel CCM.

The reason behind the Dart Angel CCM's API timings being so low is most likely due to how efficiently Dart handles asynchronous tasks. Over a request's handling process, four timing events were logged (an example of one is shown in Figure 61), being sent to the logging system. This was done via calling an asynchronous method to handle said logging request while continuing to handle the initial request. The API results when removing the impact from data-related operations show how low of an overhead Dart has for initiating asynchronous tasks.

The two histograms pertaining to when the request was outside the scope of a CCM controller were identical in shape, mainly due to full API durations only comprising approximately 0.62% of the total duration. Despite the peak in number of requests being at a considerably low duration interval, the mean duration value was higher than what would have been expected, being superior to the Spring Boot CCM's mean despite it peaking at a superior interval. This indicates a higher degree of inconsistency in the duration between the application and Dart Angel CCM's controller.

6.1.5.2 Spring Boot CCM – Flutter application case

Both the data-related operation duration and the API duration when excluding the data-related operation's impact histograms resulted in a bell shape which is indicative of a single source of data existing (timings from data-related operations and from the remaining controller actions, respectively) and, due to not being as concentrated as the Dart Angel CCM intervals, means that API operations were highly inconsistent in duration. The full API duration histogram, on the other hand, resulted in a bimodal shape which is more indicative of having two sources of data. This was the case due to these timing metrics including the timing data from both previously mentioned ones with similar impact.

Data-related operation metrics were mainly collected to better assess the duration of the remaining API controller operations by subtracting their values. Because of their stronger relation to language

and specific library performance when handling an external tool, they fell out of the scope of this study. Additionally, the fact that the package/library used in the developed CCMs to handle the SQLite database differed in implementation philosophy, sophistication, and features, such as the existence of transaction logic, made it harder to judge either language/framework in terms of competence. The extreme difference in timing values, with the Angel Dart CCM taking approximately 3.25% of the Spring Boot CCM's time, could also be related to the fact that the sending of information to the database is done asynchronously and Dart was built around the idea of being an asynchronous language, unlike Java.

```
long endTimeDB = System.currentTimeMillis();

LogHandler.log("E#" + callID + " POST: /crypto/exchange/purchase", "SC4ALLSpring",
        source, "DBEnd", String.valueOf(callID), (int) endTimeDB);

long endTimeAPI = System.currentTimeMillis();
```

Figure 62 – Code: logging between two timing event instances in the Spring Boot CCM.

The mean duration for the API timings when excluding the impact from data-related operations was considerably higher in the Spring Boot CCM (with the Dart Angel CCM's duration being approximately 9.50% of its duration). This is the case despite the logic employed being the same in both CCM implementations, as is shown in Figure 61 and Figure 62. Dart was built around the principle of being an asynchronous language, yet Java wasn't. In Spring Boot's case, a way to handle asynchronous actions (using state-of-the-art solutions) is through the @Async annotation which allows for a method to be executed in a separate thread. The obtained results show that, in the eventuality that requests are received in a successive way, Spring Boot's overhead to begin asynchronous tasks is roughly ten times Dart's native overhead.

The two histograms pertaining to when the request was outside the scope of a CCM controller were very similar in shape, mainly due to full API durations only comprising approximately 23.32% of the total duration and most requests being situated at intervals which would be in the start of the request duration spectrum, causing a slight translation of the histogram data towards the origin. As stated in the previous section, request durations without the impact from API operations peaked at a later interval for the Spring Boot CCM yet the mean duration was lower, meaning a lot more consistency in request handling duration externally to controller logic.

6.1.5.3 Dart Angel CCM – Postman case

Both the data-related operation duration and the full API duration histograms resulted in a similarly random shape, which would be hard to have happen by coincidence. That was not the case here as the histogram for API durations when excluding the impact from data-related operations was very well defined with its values concentrated at the start of the spectrum, meaning data-related operation durations were the main time-consuming actions within the controller's scope. This is supported by the fact that the mean duration for API timings when excluding these operations consists of a mere 10.60% of the full API duration. Additionally, the randomness obtained in the data-related operations most likely stemmed from the fact that the Postman case involved running collections of ten different complexity requests 100 times. With different requests taking a different amount of time to complete, a histogram with the results was bound to end up illustrating it.

API duration when excluding data-related operations saw a somewhat considerable relative difference when compared to the Dart Angel CCM's mean in the Flutter application case. In this case, it was 82.47% of its duration which is significant, yet the values themselves are so low, at 0.786 milliseconds for this case and 0.953 for the previous one, that this divergence could simply be related to not running the tests in a perfect environment where every request would be handled in a deterministic way (same request under the same circumstances always taking the same amount of time to complete) due to using real hardware.

6.1.5.4 Spring Boot CCM – Postman case

The histogram for data-related operation durations was very concentrated near the start of the spectrum, meaning operations were overall handled in a very consistent way despite collections of ten requests differing in complexity being sent, which would be expected to generate more randomness, akin to what was experienced in the Dart Angel CCM results. The histogram shape for full API durations was very random, something that was not expected, especially considering the bell shape obtained during the Flutter application case. It could be the case that a lot of requests ended up falling into the peak's interval rather than the adjacent ones which would contribute towards the bell shape, another case of non-deterministic environments possibly affecting results.

Despite the inconsistency associated with the API durations when excluding data-related operations, the histogram for the full API durations was as consistent as the data-related operation duration histogram due to these data-related operations accounting for 74.67% of the full API duration, with the peak interval slightly adjusting its position in the resulting shape (possibly due to the interval the peak was reached at in the histogram relative to API durations when excluding data-related operations).

6.1.5.5 Dart Angel CCM – JMeter case

Both the data-related operation duration and the API duration when excluding the data-related operation's impact histograms resulted in the same shape with a positively skewed histogram, peaking in the [0, 1[interval and concentrating the requests at the start of the spectrum. The full API duration histogram was also very concentrated at the start, as expected given the shape of the other previously mentioned ones, with the peak shifting to the [1, 2[interval, due to the histogram's data resulting from the values obtained in the other two histograms. Those histograms peaking at [0, 1[meant that a good amount of full API durations would end up moving up an interval to [1, 2[from both absolute durations adding up to values within that range.

It's worth noting that the shape of histograms obtained in the JMeter case were very similar to ones obtained in the Flutter application case which is likely related to the fact that all 1000 sent requests were of the same type, just as in that case.

In the JMeter case, multiple threads were used to send concurrent requests to the CCM to assess how that would affect the system's performance and the results ended up very positive with the CCM demonstrating Dart's asynchronous capabilities. The mean API duration when excluding data-related operations was 63.48% of the mean obtained in the Flutter application case, which is quite a considerable relative difference. Again though, as was the case with the Postman case, the absolute difference in values was very low and could be related to a non-deterministic environment being used for testing. Be it the case or not, the fact that the mean duration did not increase attests to the value Dart can bring to back-end server development.

6.1.5.6 Spring Boot CCM – JMeter case

As was the case in the Flutter application case, the data-related operation histogram's shape was a bell-shape, mainly related to the fact that a single source of data existed. The same cannot be said for the histogram related to API duration when excluding data-related operations as the obtained shape was not that of a bell, but a random histogram that had some visible similarities to a bell shape. This could be because of having to handle multiple requests being sent concurrently by JMeter, causing some strain on the embedded TomCat server, which handles starting new threads when requests are received. Another possible case could be the efficiency at which Java handles synchronized access to critical sections.

```
static int getCallID()
{
  int id;
  Lock lock = Lock();

  lock.synchronized(()
  {
    id = callID++;
  });

  return id;
}
```

```
public static int getCallID()
{
    int id = -1;
    reentrantlock.lock();
    try
    {
        id = callID++;
    }
    catch (Exception e)
    {
        logger.error(e.getMessage());
    }
    finally
    {
        reentrantlock.unlock();
    }
    return id;
}
```

Figure 63 – Code: request ID synchronization - Dart Angel CCM (left) and Spring Boot CCM (right).

Code pertaining to how request ID synchronization was handled in both CCM systems is shown in Figure 63. The logic employed was the same, yet the libraries and underlying language logic and capabilities differ. Due to this, either of the two languages used could have suffered in terms of API handling duration.

While the mean value for API durations when excluding data-related operations was lower in the Dart Angel CCM during this case when compared to the Flutter application case, the Spring Boot CCM saw an approximate increase of 116.94% to the mean duration, attesting to the fact that Java was not initially envisioned as an asynchronous language, simply adding support for it later.

Finally, the Dart Angel CCM's test completion time was roughly 9 seconds while the Spring Boot CCM's was roughly 20 seconds, meaning the Dart Angel CCM's duration was considerably smaller, which was not the case in the Flutter application case when looking at the full request time mean durations. As stated previously, this could have been the case due to Spring Boot's embedded server, TomCat, being strained when receiving multiple concurrent requests, slightly lowering its efficiency.

6.2 The developer perspective

As a developer, during this dissertation, some observations and opinions were gathered that, although not quantifiable, are valid contributions towards the main objective: assert if Dart is worthy as a full-stack development solution.

6.2.1 Dart VS Java frameworks

Both Java's Spring Boot and Dart frameworks were clearly designed to appeal to the programmer's ability to recognise all the similarities to smoothen out the learning curve for these newer frameworks. From Dart's side, both the Aqueduct and Angel frameworks have a lot of similar aspects to Spring Boot in terms of HTTP request handling and ORM (Object-Relational Mapping) support, which attests to the previous statement.

Dart provides native options for handling both server creation and external communication, yet it still has limited usage/presence in real systems and there are still many custom made frameworks to support back-end development that do not use such resources explicitly. This often translates to better HTTP request handlers, such as controllers, but also into excessive bloat code.

Spring Boot is an application framework built using Java that has as its objectives the increase in speed, simplicity and productivity when creating back-end servers for services. Due to this, it has been a staple in full-stack development and one of the main factors in slowing down Dart's growth in this field.

Dart frameworks are still in a very early stage, requiring more time to mature and add support for other commonly used tools, as well as better documentation. The only redeeming aspect right now is the fact that Dart's asynchronous traits are positively impacting timings related to API operations.

ORM solutions were not used in both CCM mainly due to DART related problems. Although Spring boot has state-of-the-art ORM options such as JPA [47]/Hibernate [48] available, one was not use in the Spring Boot CCM implementation due to an ORM not being used in the Dart Angel CCM and to reduce disparity between both implementations. Even if the cause behind the ORM failing to work in the Dart CCM were to be inexperience with the packages, the fact that the documentation is both lacking and confusing, which is supported by their 10/20 documentation score at pub.dev [147], added to the fact that a lot of work is required prior to having a functional ORM, simply makes it not worth using in its current state as ORMs are employed during development to ease the process, not increase complexity and workload. In the end, the type of persistence used in this study was merely secondary and a statement to how much polish a language has in terms of developer quality of life, with the actual timings related to request handling being of principal importance.

6.2.2 Dart VS Java logging availability

Full-stack systems should provide or, at least, be associated to a monitoring system, namely to provide (ideally, real-time) a solution to follow the functioning system and detect either errors or decrease in performance indicators. In general, logging solutions tend to be both used to provide feedback on system operations and to communicate with runtime monitoring systems who externally handle data analysis. In Dart's case, aside from a few options that require the use of multiple technologies in an unorthodox and overcomplicated manner, there does not seem to be any solution akin to something like log4j [123] in Java. In this study, alternative custom implementations were experimented on with InfluxDB being the destination runtime system. The result of this was better than expected, having been able to create a logging solution that completed the required tasks. Despite this though, its lack of integrations would hinder it from being used in higher-end projects and InfluxDB's histogram tool's overly simplistic configuration options leave it wanting for more.

Despite log4j being a much more robust solution than Dart's logging package [100], Dart's performance within a full-stack environment could continue to be evaluated by using it to develop the logging system's request handling subsystem while not negatively impacting the CCM comparison.

6.2.3 Dart VS Java community support/activity

During the dissertation, it became clear that one of the main issues behind Dart's popularity in the back-end server area was due to low community engagement. The small number of packages and the amount of them that were either outdated or straight-up abandoned increased development time and made it so the number of options available was lower than in Java, for example. This contributed towards the Dart Angel CCM's development seeing a great number of dependency changes throughout.

Bridge libraries for tools like InfluxDB, which tend to operate individually and externally to back-end servers, increase the need for there to be continued community support in keeping the packages up to date, which seems like the bane of the current Dart community.

Unlike what happened while developing the Dart-based CCM, there were no meaningful dependency changes in the Spring Boot counterpart, only ever having to update the SQLite dependency. On one end, this could indicate that the community is stagnate and libraries are getting abandoned, as with Dart. On the other hand, however, is the case that most of the libraries available, especially regarding Spring Boot, are stable and more than ready to be used due to continued and active community support for years, something that Dart cannot hold true to.

6.2.4 Dart and CORS

The existence of Cross-Origin Resource Sharing (CORS) [118] is something that developers must often address when developing their systems in order to make their resources available externally.

In Angel's case, packages to handle CORS exist with them being angel_cors [148] for Angel2 and angel3_cors [149] for Angel3, both analogous in functionality. This functionality, however, is not ideal since it only seems to work when using Angel's basic routing and not controllers (through the usage of something like an annotation, as is the case in Spring Boot). This can be considered counter intuitive because CORS is often handled when the system is already mostly complete, or at least one path is functional. Someone implementing HTTP request handling using controllers would then not be able to handle CORS-related issues when keeping the already developed codebase as is.

The fact that CORS is such a hassle to deal with in Dart usually leads to a solution based on common reverse proxy servers that mediate the actual services. This is a common solution which has made most developers simply resort to NginX [119], acting as a reverse proxy. Having used NginX in the conducted study and obtained positive results in this area, it seems like a competent solution especially considering it decouples CORS handling from request handling logic.

6.2.5 Code evaluation

Programming language performance and frameworks/tool availability are very important aspects when it comes to choosing the technologies used to develop a project yet there are two other key aspects that must be considered and those are code quality and code security. As such, these aspects were evaluated in the context of the developed CCM back ends using SonarQube [150], a codebase analyser that uses existing (most of which are freely available) code analysis tools and integrates the results into a unified report with most popular code-based metrics.

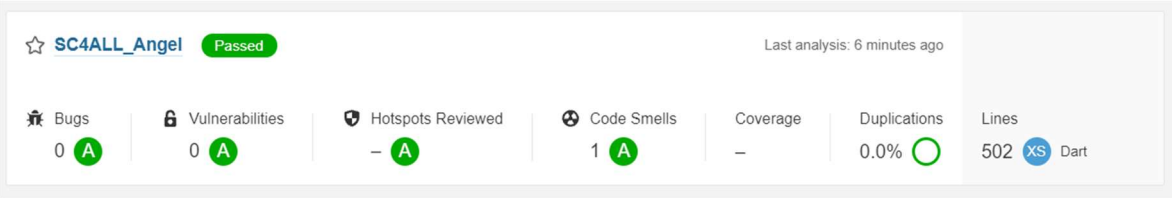

Figure 64 – SonarQube summary for the Dart Angel CCM's statistics.

The first analysis run was on the Dart Angel CCM's code, and a summary of the results is shown in Figure 64. No bugs or vulnerabilities were picked up, no security hotspots required reviewing, no duplicated code blocks were found, and a single code smell was brought up, resulting in a clean "A" rating being attributed in every category. The one code smell that was picked up had to do with the

timer used to pseudo-randomize the currency value fluctuation. This timer's variable is "never accessed" and this is due to, for the purposes of this study, it never being closed in the code, stopping once the server does.

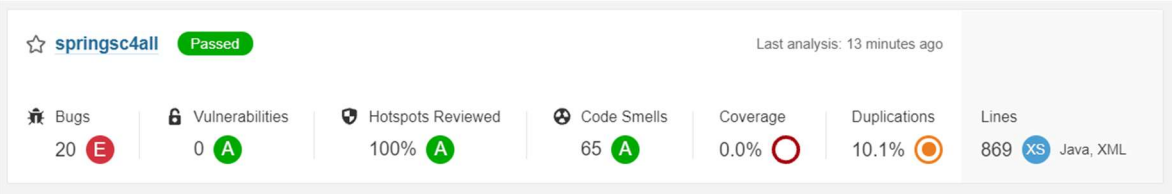

Figure 65 – SonarQube summary for the Spring Boot CCM's statistics.

Contrasting the Dart Angel CCM's successful analysis results we have the Spring Boot CCM's which are shown in Figure 65. Just as with Dart's implementation, no vulnerabilities were picked up yet 20 bugs were. All these bugs stemmed from the same root issue with it being a closing instruction for statement objects in try blocks, which can lead to a memory leak. Two security hotspots required reviewing, yet they both presented themselves as irrelevant considering the CCM's context: the first was related to low cryptography related to the usage of pseudo-randomization, which was irrelevant due to real-life systems not fluctuating currency values in this way, hence the fact that this security hotspot would not exist; and the second was related to the usage of a static IP when sending a logging request, which is irrelevant due to its unchanging nature (due to using a static configuration in the deployed NginX reverse proxy). As for code smells, a total of 65 were found and an "A" rating was still achieved. These were all related to three main aspects: the usage of String literals being passed directly into the LogHandler.log() methods rather than being defined in constants for future use; the usage of string concatenation rather than format specifiers when logging information, which is safe; and the usage of String literals in SQL queries rather than declaring constants, which was only picked up due to the used queries being split into multiple lines for maintainability purposes. Finally, 10.1% of the code was marked for duplications, yet all these duplications happened within the SQLiteConnection file and stemmed final aspect mentioned regarding the code smells.

Both CCM servers were developed by the same person, one with far more experience in Java than in Dart. Despite this, the developed Dart code resulted in better analysis results than the Java code. Even with the issues found in the Java analysis being minor and easily/quickly fixable, the fact that this happened is a big statement towards how easy it is to develop using Dart when compared to Java. Java is often criticized for both its boilerplate-heavy code and its syntax, both of which aren't as prominent in Dart. This could have easily been the cause behind these overlooked aspects during development which, over the course of larger projects, could result in increased debugging time. For some context, both CCM's code were tagged by SonarQube as "XS" with the Dart-based one having

a total of 502 lines and the Java-based one having 869, an approximate increase of 73.11%. Considering the analysis results, the Dart-based CCM's technical debt, or approximate time estimated to rework/fix projects, was five minutes while the Java-based CCM's was one day and four hours. This implies that the time to fix the Dart-based CCM would be approximately 0.30% of the time required in the Java-based one, which is huge. Then, increase the scale of the project to one where debugging a Dart project would take a week over its entire development process and you have the equivalent to a production disaster on the Java side of things.

Java has native support for analysis in SonarQube, yet Dart does not. Results obtained during this study were via the usage of a third-party plugin, sonar-flutter [151], which allows for SonarQube to be generate analysis reports on both Flutter and native Dart projects.

7 Conclusions

The main objective of this dissertation was to explore if Dart is already a viable option to develop a full-stack solution comprising the presentation/front-end layer and back-end layer. With that in mind, a simple scenario and a system specification, the Crypto Currency Manager (CCM), were considered and used as proof of concept to benchmark a Dart Angel implementation versus a Spring Boot's – the reference state-of-the-art system.

It was hypothesized that Dart is powerful enough of a language to be able to create competent back-end servers, especially given the number of available frameworks, even if performance-wise it ended up being worse than current state-of-the-art solutions. Given Dart's focus on asynchrony, operations revolving around this concept were hypothesized to be handled efficiently.

The CCM scenario, which depicted a realistic situation for a full-stack system, was developed and deployed. Although custom made, it allowed to explore most typical constructs for full-stack solutions, hence providing a sound solution to perform Dart's viability analysis in back-end server development.

The main topic covered in the performed study was the comparison between developing back-end servers using Java's Spring Boot and Dart's Angel frameworks. This comparison was imperative to be performed, seeing as it not only allowed for there to be a way to compare Dart's performance but also to know what aspects there even were to compare to begin with. The way both languages/frameworks approach controller logic, logging availability, communication, data persistence and some security aspects, namely CORS (Cross-Origin Resource Sharing) related, were the primary focus.

It was decided to not delve into other comparisons, be it using other Dart frameworks, such as Aqueduct [9] or Jaguar [31], or frameworks built atop other languages, such as Kotlin [152] Spring Boot [8] or JavaEE/JakartaEE [153].

CI/CD (Continuous Integration/Continuous Deployment) are important aspects to consider in this area as they assist in automating and distributing Dart-built full-stack systems. Due to not yet knowing if Dart is suitable as a language to develop complex systems, this was not covered during this study.

To simplify metric obtaining and overall communication, no security measures were implemented when it comes to data storage/communication. By removing the additional time-altering variable that would be encryption and decryption algorithms, a better comparison was achieved, only having to evaluate timings related to database and API (Application Programming Interface) operations.

Despite the issues regarding Dart frameworks, namely the community and framework volatility, the CCM was able to be implemented using the Angel framework which was then successfully

compared to a Java-based CCM using the Spring Boot framework. From purely a performance perspective, it was found that under most conditions the Dart Angel implementation outperformed the reference system. The only outlier was in the Flutter application case where timings external to CCM controllers saw Dart performing worse than Java, which could have been due to the usage of a non-deterministic production environment. Either way, for this type of metrics, Dart still outperformed Java in the remaining two cases proving to be a superior choice or, at least, as viable of a choice timings-wise. As for the remaining timing metrics, Dart always far outperformed, clearly proving its ability to compete with and, possibly, outclass current state-of-the-art solutions.

While it was hypothesized that Dart would be powerful enough of a language to be able to create competent back-end servers it was not expected to see such a great difference in performance between both CCM implementations. Dart is an extremely new and powerful language which is being heavily invested into by Google. As time goes on, it is bound to keep improving and, hopefully, catching the attention of other developers in a way that leads to an ever-growing community to be established.

Not only did the conducted study prove that Dart can be used to develop back-end servers, proving the initial hypothesis correct, but it also demonstrated that Dart's asynchronous nature could highly benefit it in the area, possibly being able to overtake Java's Spring Boot in popularity if more time and resources are invested into it.

From a developer's perspective, it already provides its community with a robust base to build upon. Not only is the language itself powerful and versatile, but the documentation and supporting websites, such as pub.dev [154], increase its appeal and learnability/maintenance capabilities. Developers have already delved into framework development providing the community with a starting point to build off of, with frameworks like Angel allowing for increased modularity. Despite there being issues, such as third-party documentation quality, Dart's upsides vastly outweigh its current, and fixable, downsides.

In spite of Dart's myriad advantages, one of its apparent disadvantages is the lack of community support when compared to other languages which results in less options when it comes to choosing packages to use during development, possibly making it so the usage of a specific technology is not viable due to the sheer amount of time and resources it would require to manually develop a bridge package to support it. As such, Dart is currently not ready to be considered state of the art, despite being able to achieve better performance results than current state of the art. For simpler projects though, it is still an option that should be considered, filling a role akin to what Python [155] does as a language in terms of providing users with an easy and efficient way to create scripts.

As for how viable Dart is to be used for back-end development, it highly depends, as of the time of writing, on the type of project being developed, namely the tools it would require using.

7.1 Logging

Logging is an extremely important part of any system and, as such, this study could not consider itself credible if the topic had not approached. When researching Dart's options in this area it became apparent that they were lacking when compared to Java's own options, such as log4j [124]. Dart's main logging option was the "logging" package [100] which provides very basic logging functionality, not presenting developers with any support for external systems (through appenders). As such, to collect system metrics for better analysis, a new logging solution was developed. This solution revolved around the existence of an addition system dedicated to logging which was accessible through a REST (Representational State Transfer) API and would treat data before storing it into an InfluxDB (Influx Database) [79] instance.

Despite being a simple solution, the developed logging system fulfilled its purposes and allowed for a comparison to be done between both CCM system's collected timing metrics. The main lacking feature in Dart when exploring logging is remote logging capability, meaning third-party developers would need to invest time into creating packages to support the required logging tools.

7.2 Future work

In the conducted study, a comparison between a Dart-built and a Java-built back-end server was performed to be able to assess Dart's capabilities. Due to its popularity, Angel was chosen as the framework used in both the Dart-based CCM and the logging system, yet this brought forth a new question: Was Angel the best option?

Given that it was already asserted that Dart is competent enough of a language to develop back-end servers with possibly better performance than current state of the art, the next logical step is to further explore Dart itself and the available options. Initially, the plan was to use Aqueduct, yet those plans were changed when development on it ceased. Aqueduct's successor, Liquidart [30], is still currently being developed so it would be advantageous to study it and verify what advantages and disadvantages it presents when compared to Angel. Another robust-looking option for this study would be the Jaguar framework.

Flutter has already settled in as state of the art for front-end development. Due to its popularity, a lot of community focus has been towards developing Flutter packages rather than Dart native ones. As such, the possibility of developing a back-end server using Flutter arises, allowing for an entire full-stack system to be developed solely using Flutter. This would allow for multiple packages that require the Flutter SDK to function, such as Drift [95], to be usable, increasing developer agency over the project via increasing choice diversity.

A subject that is very important when it comes to development yet was not touched upon during the conducted study is DevOps. DevOps aims to optimize some parts of the development process by automating work and allowing for value to be delivered more frequently. While some initial research prior to the study beginning was conducted, it was decided to not be included into its scope since it was not yet known if it was possible to develop competent back-end servers using Dart, reducing the importance of DevOps if Dart had been found to be inept.

Aside from simple technical evaluations; maintainability, usability, and solution appeal must also be studied. Not only does this impact framework choice but also general project design choices, such as the possibility of developing while following the BLoC pattern [156], which already has package support in Dart [157], or the usage of the OpenAPI [36] specification to generate communication endpoints.

Finally, whether further research into Dart is performed or not, the community must increase its support to be able to achieve a healthy amount of tool support via third-party packages to further increase Dart's performance and usage viability.

References

[1] "Dart programming language | Dart," Google, [Online]. Available: https://dart.dev/. [Accessed 19 October 2021].

[2] "Celebrating Dart's birthday with the first release of the Dart SDK," Google, [Online]. Available: https://news.dartlang.org/2012/10/dart-m1-release.html. [Accessed 19 October 2021].

[3] "Beautiful native apps in record time | Flutter," Google, [Online]. Available: https://flutter.dev/. [Accessed 19 October 2021].

[4] "FAQ | Flutter," Google, [Online]. Available: https://flutter.dev/docs/resources/faq. [Accessed 19 October 2021].

[5] "Showcase | Flutter," Google, [Online]. Available: https://flutter.dev/showcase. [Accessed 19 October 2021].

[6] "Cross-platform mobile frameworks used by global developers 2021 | Statista," Statista, [Online]. Available: https://www.statista.com/statistics/869224/worldwide-software-developer-working-hours/. [Accessed 19 October 2021].

[7] "Java | Oracle," Oracle, [Online]. Available: https://www.java.com/en/. [Accessed 19 October 2021].

[8] "Spring Boot," Pivotal Software, [Online]. Available: https://spring.io/projects/spring-boot. [Accessed 19 October 2021].

[9] "Aqueduct | Multi-threaded Dart Server-Side Framework," Stable Kernel, [Online]. Available: https://aqueduct.io/. [Accessed 19 October 2021].

[10] "Angel Framework," Dukefirehawk Consulting, [Online]. Available: https://angel3-framework.web.app/#/about. [Accessed 19 October 2021].

[11] K. Walrath, "Dart asynchronous programming: Isolates and event loops | by Kathy Walrath | Dart | Medium," 25 July 2019. [Online]. Available: https://medium.com/dartlang/dart-asynchronous-programming-isolates-and-event-loops-bffc3e296a6a. [Accessed 19 October 2021].

[12] "Asynchronous programming: Streams | Dart," Google, [Online]. Available: https://dart.dev/tutorials/language/streams. [Accessed 19 October 2021].

[13] "Asynchronous programming: futures, async, await | Dart," Google, [Online]. Available: https://dart.dev/codelabs/async-await. [Accessed 19 October 2021].

[14] “Sound null safety | Dart,” Google, [Online]. Available: https://dart.dev/null-safety. [Accessed 19 October 2021].

[15] “Understanding null safety | Dart,” Google, [Online]. Available: https://dart.dev/null-safety/understanding-null-safety. [Accessed 19 October 2021].

[16] “Introduction to widgets | Flutter,” Google, [Online]. Available: https://flutter.dev/docs/development/ui/widgets-intro. [Accessed 19 October 2021].

[17] “React – A JavaScript library for building user interfaces,” Facebook, [Online]. Available: https://reactjs.org/. [Accessed 19 October 2021].

[18] “JavaScript.com,” JavaScript, [Online]. Available: https://www.javascript.com/. [Accessed 19 October 2021].

[19] “FutureBuilder (Flutter Widget of the Week) - YouTube,” Google, [Online]. Available: https://www.youtube.com/watch?v=ek8ZPdWj4Qo. [Accessed 19 October 2021].

[20] “FutureBuilder class - widgets library - Dart API,” Google, [Online]. Available: https://api.flutter.dev/flutter/widgets/FutureBuilder-class.html. [Accessed 19 October 2021].

[21] “StreamBuilder class - widgets library - Dart API,” Google, [Online]. Available: https://api.flutter.dev/flutter/widgets/StreamBuilder-class.html. [Accessed 19 October 2021].

[22] “AsyncWidgetBuilder typedef - widgets library - Dart API,” Google, [Online]. Available: https://api.flutter.dev/flutter/widgets/AsyncWidgetBuilder.html. [Accessed 19 October 2021].

[23] “StreamBuilder (Flutter Widget of the Week) - YouTube,” Google, [Online]. Available: https://www.youtube.com/watch?v=MkKEWHfy99Y&t=1s. [Accessed 19 October 2021].

[24] M. Kazlauskas, “Flutter Design Patterns: 18 — Builder | by Mangirdas Kazlauskas | Flutter Community | Medium,” 15 April 2020. [Online]. Available: https://medium.com/flutter-community/flutter-design-patterns-18-builder-cdc90b222724. [Accessed 19 October 2021].

[25] “InheritedWidget (Flutter Widget of the Week) - YouTube,” Google, [Online]. Available: https://www.youtube.com/watch?v=1t-8rBCGBYw. [Accessed 19 October 2021].

[26] “InheritedWidget class - widgets library - Dart API,” Google, [Online]. Available: https://api.flutter.dev/flutter/widgets/InheritedWidget-class.html. [Accessed 19 October 2021].

[27] “provider | Flutter Package,” dashoverflow, 24 September 2021. [Online]. Available: https://pub.dev/packages/provider. [Accessed 19 October 2021].

[28] B. Egan, “scoped_model | Flutter Package,” 18 October 2021. [Online]. Available: https://pub.dev/packages/scoped_model. [Accessed 19 October 2021].

[29] L. Gupta, “What is REST,” Lokesh Gupta, 19 October 2021. [Online]. Available: https://restfulapi.net/. [Accessed 19 October 2021].

[30] “Liquidart Documentation,” Aldrin's Art Factory, [Online]. Available: https://aldrinsartfactory.github.io/liquidart/. [Accessed 19 October 2021].

[31] “Jaguar,” Jaguar Dart, [Online]. Available: https://jaguar-dart.com/. [Accessed 19 October 2021].

[32] Y. Lvivski, “start | Dart Package,” 11 May 2021. [Online]. Available: https://pub.dev/packages/start. [Accessed 19 October 2021].

[33] “shelf | Dart Package,” Google, 8 July 2021. [Online]. Available: https://pub.dev/packages/shelf. [Accessed 19 October 2021].

[34] “vane | Dart Package,” Sourcevoid, 11 February 2021. [Online]. Available: https://pub.dev/packages/vane. [Accessed 19 October 2021].

[35] “alfred | Dart Package,” 18 October 2021. [Online]. Available: https://pub.dev/packages/alfred. [Accessed 19 October 2021].

[36] “Home - OpenAPI Initiative,” The Linux Foundation, [Online]. Available: https://www.openapis.org/. [Accessed 19 October 2021].

[37] “OAuth 2.0 — OAuth,” [Online]. Available: https://oauth.net/2/. [Accessed 19 October 2021].

[38] “ASP.NET documentation | Microsoft Docs,” Microsoft, [Online]. Available: https://docs.microsoft.com/en-us/aspnet/core/?view=aspnetcore-5.0. [Accessed 19 October 2021].

[39] “Express - Node.js web application framework,” OpenJS Foundation, [Online]. Available: https://expressjs.com/. [Accessed 19 October 2021].

[40] “Stack Overflow - Where Developers Learn, Share, & Build Careers,” Stack Exchange, [Online]. Available: https://stackoverflow.com/. [Accessed 19 October 2021].

[41] “Stack Overflow Developer Survey 2018,” Stack Exchange, [Online]. Available: https://insights.stackoverflow.com/survey/2018. [Accessed 19 October 2021].

[42] “Stack Overflow Developer Survey 2019,” Stack Exchange, [Online]. Available: https://insights.stackoverflow.com/survey/2019. [Accessed 19 October 2021].

[43] “Stack Overflow Developer Survey 2020,” Stack Exchange, [Online]. Available: https://insights.stackoverflow.com/survey/2020. [Accessed 19 October 2021].

[44] “Spring | Home,” Pivotal Software, [Online]. Available: https://spring.io/. [Accessed 19 October 2021].

[45] “2021 Java Developer Productivity Report | Rebel,” Perforce Software, [Online]. Available: https://www.jrebel.com/resources/java-developer-productivity-report-2021. [Accessed 19 October 2021].

[46] “2020 Java Developer Productivity Report | JRebel & XRebel by Perforce,” Perforce Software, [Online]. Available: https://www.jrebel.com/resources/java-developer-productivity-report-2020. [Accessed 19 October 2021].

[47] “Spring Data JPA,” Pivotal Software, [Online]. Available: https://spring.io/projects/spring-data-jpa. [Accessed 19 October 2021].

[48] “Hibernate. Everything data. - Hibernate,” Hibernate, [Online]. Available: https://hibernate.org/. [Accessed 19 October 2021].

[49] “17. Web MVC framework,” Pivotal Software, [Online]. Available: https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/mvc.html. [Accessed 19 October 2021].

[50] A. vyas, “MVC Pattern. The Model-View-Controller (MVC) is an… | by Anshul vyas | Medium,” 3 October 2018. [Online]. Available: https://anshul-vyas380.medium.com/mvc-pattern-3b5366e60ce4. [Accessed 19 October 2021].

[51] “Fuchsia,” Google, [Online]. Available: https://fuchsia.dev/. [Accessed 19 October 2021].

[52] “1. Getting Started - Aqueduct,” Stable Kernel, [Online]. Available: https://aqueduct.io/docs/tut/getting-started/. [Accessed 19 October 2021].

[53] “Controllers - Angel3 Developer Guide,” Dukefirehawk Consulting, [Online]. Available: https://angel3-docs.dukefirehawk.com/guides/controllers. [Accessed 19 October 2021].

[54] “Basic Routing - Angel3 Developer Guide,” Dukefirehawk Consulting, [Online]. Available: https://angel3-docs.dukefirehawk.com/guides/basic-routing. [Accessed 19 October 2021].

[55] “Getting Started | Building an Application with Spring Boot,” Pivotal Software, [Online]. Available: https://spring.io/guides/gs/spring-boot/. [Accessed 19 October 2021].

[56] “About,” SmartBear Software, [Online]. Available: https://swagger.io/about/. [Accessed 19 October 2021].

[57] “OpenAPI Design & Documentation Tools | Swagger,” SmartBear Software, [Online]. Available: https://swagger.io/tools/. [Accessed 19 October 2021].

[58] M. Hoyos, “What is an ORM and Why You Should Use it | by Mario Hoyos | Bits and Pieces,” 24 December 2018. [Online]. Available: https://blog.bitsrc.io/what-is-an-orm-and-why-you-should-use-it-b2b6f75f5e2a. [Accessed 19 October 2021].

[59] “MySQL,” Oracle, [Online]. Available: https://www.mysql.com/. [Accessed 19 October 2021].

[60] “Modeling Data and Relationships - Aqueduct,” Stable Kernel, [Online]. Available: https://aqueduct.io/docs/db/modeling_data/. [Accessed 19 October 2021].

[61] “PostgreSQL: The world's most advanced open source database,” The PostgreSQL Global Development Group, [Online]. Available: https://www.postgresql.org/. [Accessed 19 October 2021].

[62] “Basic Queries - Aqueduct,” Stable Kernel, [Online]. Available: https://aqueduct.io/docs/db/executing_queries/. [Accessed 19 October 2021].

[63] “Migrations - Angel3 Developer Guide,” Dukefirehawk Consulting, [Online]. Available: https://angel3-docs.dukefirehawk.com/guides/orm/migrations. [Accessed 19 October 2021].

[64] “Basic Functionality - Angel3 Developer Guide,” Dukefirehawk Consulting, [Online]. Available: https://angel3-docs.dukefirehawk.com/guides/orm/basic-functionality. [Accessed 19 October 2021].

[65] “Spring Boot Reference Documentation,” Pivotal Software, [Online]. Available: https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto.data-access.separate-entity-definitions-from-spring-configuration. [Accessed 19 October 2021].

[66] “H2 Database Engine,” [Online]. Available: https://www.h2database.com/html/main.html. [Accessed 19 October 2021].

[67] “HSQLDB,” The HSQL Development Group, [Online]. Available: http://hsqldb.org/. [Accessed 19 October 2021].

[68] “Repository (Spring Data Core 2.5.5 API),” Pivotal Software, [Online]. Available: https://docs.spring.io/spring-data/commons/docs/2.5.5/api/org/springframework/data/repository/Repository.html. [Accessed 19 October 2021].

[69] “CrudRepository (Spring Data Core 2.5.5 API),” Pivotal Software, [Online]. Available: https://docs.spring.io/spring-data/commons/docs/2.5.5/api/org/springframework/data/repository/CrudRepository.html. [Accessed 19 October 2021].

[70] “NiceHash - API Docs,” Nicehash, [Online]. Available: https://www.nicehash.com/docs/rest. [Accessed 19 October 2021].

[71] “Leading Cryptocurrency Platform for Mining and Trading | NiceHash,” NiceHash, [Online]. Available: https://www.nicehash.com/. [Accessed 19 October 2021].

[72] “Commits · stablekernel/aqueduct · GitHub,” Stable Kernel, [Online]. Available: https://github.com/stablekernel/aqueduct/commits/master. [Accessed 19 October 2021].

[73] T. Osakwe, “Angel - Dart on the Server,” [Online]. Available: https://angel-dart.dev/. [Accessed 19 October 2021].

[74] T. Osakwe, “angel_framework | Dart Package,” 5 February 2020. [Online]. Available: https://pub.dev/packages/angel_framework/versions/2.1.1. [Accessed 19 October 2021].

[75] M. Thomsen, “Announcing Dart 2.12. Sound null safety and Dart FFI ship to… | by Michael Thomsen | Dart | Medium,” 3 March 2021. [Online]. Available: https://medium.com/dartlang/announcing-dart-2-12-499a6e689c87. [Accessed 19 October 2021].

[76] “angel3_framework 4.0.0 | Dart Package,” Dukefirehawk Consulting, 14 May 2021. [Online]. Available: https://pub.dev/packages/angel3_framework/versions/4.0.0. [Accessed 19 October 2021].

[77] “History for packages/framework - dukefirehawk/angel · GitHub,” Dukefirehawk Consulting, [Online]. Available: https://github.com/dukefirehawk/angel/commits/master?after=d0fc25c4e191b1379b98c

33075a389c90cf08cb3+34&branch=master&path%5B%5D=packages&path%5B%5D=framework. [Accessed 19 October 2021].

[78] "angel3_framework 4.1.0 | Dart Package," Dukefirehawk Consulting, 22 June 2021. [Online]. Available: https://pub.dev/packages/angel3_framework/versions/4.1.0. [Accessed 19 October 2021].

[79] "InfluxDB: Purpose-Built Open Source Time Series Database | InfluxData," InfluxData, [Online]. Available: https://www.influxdata.com/. [Accessed 19 October 2021].

[80] T. Osakwe, "angel_orm | Dart Package," 19 August 2019. [Online]. Available: https://pub.dev/packages/angel_orm. [Accessed 19 October 2021].

[81] "influxdb_client | Dart Package," 22 October 2021. [Online]. Available: https://pub.dev/packages/influxdb_client. [Accessed 22 October 2021].

[82] "Downsampling and data retention | InfluxDB OSS 1.7 Documentation," InfluxData, [Online]. Available: https://docs.influxdata.com/influxdb/v1.7/guides/downsampling_and_retention/. [Accessed 19 October 2021].

[83] "How to Use Swagger Inspector," SmartBear Software, [Online]. Available: https://swagger.io/docs/swagger-inspector/how-to-use-swagger-inspector/. [Accessed 19 October 2021].

[84] T. Osakwe, "angel/packages/orm at master · angel-dart/angel · GitHub," [Online]. Available: https://github.com/angel-dart/angel/tree/master/packages/orm. [Accessed 19 October 2021].

[85] T. Osakwe, "angel_orm_generator | Dart Package," 4 July 2019. [Online]. Available: https://pub.dev/packages/angel_orm_generator. [Accessed 19 October 2021].

[86] "build_runner | Dart Package," Google, 1 October 2021. [Online]. Available: https://pub.dev/packages/build_runner. [Accessed 19 October 2021].

[87] T. Osakwe, "angel_serialize | Dart Package," 5 October 2019. [Online]. Available: https://pub.dev/packages/angel_serialize. [Accessed 19 October 2021].

[88] T. Osakwe, "https://pub.dev/packages/angel_serialize," [Online]. Available: https://github.com/angel-dart-archive/serialize. [Accessed 19 October 2021].

[89] T. Osakwe, "angel_model | Dart Package," 26 April 2019. [Online]. Available: https://pub.dev/packages/angel_model. [Accessed 19 October 2021].

[90] T. Osakwe, “angel_serialize_generator | Dart Package,” 4 July 2019. [Online]. Available: https://pub.dev/packages/angel_serialize_generator. [Accessed 19 October 2021].

[91] “angel3_serialize - Dart API docs,” Tobe Osakwe, [Online]. Available: https://pub.dev/documentation/angel3_serialize/latest/. [Accessed 19 October 2021].

[92] “SQLite Home Page,” SQLite Consortium, [Online]. Available: https://www.sqlite.org/index.html. [Accessed 19 October 2021].

[93] S. Binder, “sqlite3 | Dart Package,” 19 October 2021. [Online]. Available: https://pub.dev/packages/sqlite3. [Accessed 19 October 2021].

[94] S. Binder, “moor | Dart Package,” 13 October 2021. [Online]. Available: https://pub.dev/packages/moor. [Accessed 19 October 2021].

[95] S. Binder, “Drift,” [Online]. Available: https://drift.simonbinder.eu/. [Accessed 19 October 2021].

[96] “sqflite | Flutter Package,” tekartik, 24 August 2021. [Online]. Available: https://pub.dev/packages/sqflite. [Accessed 19 October 2021].

[97] “Timer.periodic constructor - Timer class - dart:async library - Dart API,” Google, [Online]. Available: https://api.dart.dev/stable/2.14.4/dart-async/Timer/Timer.periodic.html. [Accessed 19 October 2021].

[98] “angel3_framework | Dart Package,” Dukefirehawk Consulting, 4 october 2021. [Online]. Available: https://pub.dev/packages/angel3_framework. [Accessed 19 October 2021].

[99] “http | Dart Package,” Google, 4 October 2021. [Online]. Available: https://pub.dev/packages/http. [Accessed 19 October 2021].

[100] “logging | Dart Package,” Google, 14 September 2021. [Online]. Available: https://pub.dev/packages/logging. [Accessed 19 October 2021].

[101] “synchronized | Dart Package,” tekartik, 17 February 2021. [Online]. Available: https://pub.dev/packages/synchronized. [Accessed 19 October 2021].

[102] “Maven Repository: com.influxdb » influxdb-client-java,” InfluxData, 17 September 2021. [Online]. Available: https://mvnrepository.com/artifact/com.influxdb/influxdb-client-java. [Accessed 19 October 2021].

[103] “HttpURLConnection (Java SE 14 & JDK 14),” Oracle, [Online]. Available: https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/net/HttpURLConnection.html. [Accessed 19 October 2021].

[104] “Wireshark · Go Deep.,” Wireshark Foundation, [Online]. Available: https://www.wireshark.org/. [Accessed 19 October 2021].

[105] “HttpClient (Java SE 14 & JDK 14),” Oracle, [Online]. Available: https://docs.oracle.com/en/java/javase/14/docs/api/java.net.http/java/net/http/HttpClient.html. [Accessed 19 October 2021].

[106] “RestController (Spring Framework 5.3.11 API),” Pivotal Software, [Online]. Available: https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/RestController.html. [Accessed 19 October 2021].

[107] “SpringFox by springfox,” springfox, [Online]. Available: https://springfox.github.io/springfox/. [Accessed 19 October 2021].

[108] “Scheduled (Spring Framework 5.3.11 API),” Pivotal Software, [Online]. Available: https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/scheduling/annotation/Scheduled.html. [Accessed 19 October 2021].

[109] “Timer (Java SE 10 & JDK 10),” Oracle, [Online]. Available: https://docs.oracle.com/javase/10/docs/api/java/util/Timer.html. [Accessed 19 October 2021].

[110] “Maven – Welcome to Apache Maven,” The Apache Software Foundation, [Online]. Available: https://maven.apache.org/. [Accessed 19 October 2021].

[111] “Maven – POM Reference,” The Apache Software Foundation, [Online]. Available: https://maven.apache.org/pom.html. [Accessed 19 October 2021].

[112] “Maven Repository: org.springframework.boot » spring-boot-starter-data-rest,” Pivotal Software, 21 October 2021. [Online]. Available: https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-data-rest. [Accessed 22 October 2021].

[113] “Maven Repository: org.springframework.boot » spring-boot-starter-web,” Pivotal Software, 21 October 2021. [Online]. Available: https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-web. [Accessed 22 October 2021].

[114] “Maven Repository: org.springframework.boot » spring-boot-starter-log4j2,” Pivotal Software, 21 October 2021. [Online]. Available:

https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-log4j2. [Accessed 22 October 2021].

[115] "Maven Repository: org.xerial » sqlite-jdbc," xerial, 30 August 2021. [Online]. Available: https://mvnrepository.com/artifact/org.xerial/sqlite-jdbc. [Accessed 19 October 2021].

[116] "Maven Repository: org.springframework.boot » spring-boot-maven-plugin," Pivotal Software, 21 October 2021. [Online]. Available: https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-maven-plugin. [Accessed 22 October 2021].

[117] "sleek_circular_slider | Flutter Package," 27 June 2021. [Online]. Available: https://pub.dev/packages/sleek_circular_slider. [Accessed 19 October 2021].

[118] "Cross-Origin Resource Sharing (CORS) - HTTP | MDN," Mozilla, [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS. [Accessed 19 October 2021].

[119] "NGINX | High Performance Load Balancer, Web Server, & Reverse Proxy," F5 Networks, [Online]. Available: https://www.nginx.com/. [Accessed 19 October 2021].

[120] "CORS support in Spring Framework," Pivotal Software, [Online]. Available: https://spring.io/blog/2015/06/08/cors-support-in-spring-framework. [Accessed 19 October 2021].

[121] "Beginner's Guide," F5 Networks, [Online]. Available: http://nginx.org/en/docs/beginners_guide.html. [Accessed 19 October 2021].

[122] "Logback Home," QOS.ch, [Online]. Available: http://logback.qos.ch/. [Accessed 19 October 2021].

[123] "Log4j – Apache Log4j 2," The Apache Software Foundation, [Online]. Available: https://logging.apache.org/log4j/2.x/. [Accessed 19 October 2021].

[124] "Apache log4j 1.2 - Short introduction to log4j," The Apache Software Foundation, [Online]. Available: https://logging.apache.org/log4j/1.2/manual.html. [Accessed 19 October 2021].

[125] E. Dietrich and K. Estreich, "Log Appender: What Is It and Why Would You Use It? - DZone DevOps," 7 January 2018. [Online]. Available: https://dzone.com/articles/log-appender-what-is-it-and-why-would-you-use-it. [Accessed 19 October 2021].

[126] "Logz.io: Cloud Observability for Engineers," Logz.io SLA, [Online]. Available: https://logz.io/. [Accessed 19 October 2021].

[127] “logging_appenders | Dart Package,” codeux.design, 28 February 2021. [Online]. Available: https://pub.dev/packages/logging_appenders. [Accessed 19 October 2021].

[128] “GitHub - grafana/loki: Like Prometheus, but for logs.,” Grafana Labs, [Online]. Available: https://github.com/grafana/loki. [Accessed 19 October 2021].

[129] “logging/logger.dart at master · dart-lang/logging · GitHub,” Google, [Online]. Available: https://github.com/dart-lang/logging/blob/master/lib/src/logger.dart. [Accessed 19 October 2021].

[130] “What is Time Series Data? | Definition, Examples, Types & Discussion,” InfluxData, [Online]. Available: https://www.influxdata.com/what-is-time-series-data/. [Accessed 19 October 2021].

[131] “Time Series Database (TSDB) Explained | InfluxDB | InfluxData,” InfluxData, [Online]. Available: https://www.influxdata.com/time-series-database/. [Accessed 19 October 2021].

[132] “Flux Open Source Query Language | InfluxData,” InfluxData, [Online]. Available: https://www.influxdata.com/products/flux/. [Accessed 19 October 2021].

[133] “Query InfluxDB with Flux | InfluxDB OSS 2.0 Documentation,” InfluxData, [Online]. Available: https://docs.influxdata.com/influxdb/v2.0/query-data/get-started/query-influxdb/. [Accessed 19 October 2021].

[134] “Work with records in Flux | Flux 0.x Documentation,” InfluxData, [Online]. Available: https://docs.influxdata.com/flux/v0.x/data-types/composite/record/. [Accessed 19 October 2021].

[135] “DB-Engines Ranking - popularity ranking of time Series DBMS,” solid IT, [Online]. Available: https://db-engines.com/en/ranking/time+series+dbms. [Accessed 10 September 2021].

[136] “DB-Engines Ranking - popularity ranking of database management systems,” solid IT, [Online]. Available: https://db-engines.com/en/ranking. [Accessed 10 September 2021].

[137] “Telegraf Open Source Server Agent | InfluxData,” InfluxData, [Online]. Available: https://www.influxdata.com/time-series-platform/telegraf/. [Accessed 19 October 2021].

[138] “Scrape data using InfluxDB scrapers | InfluxDB OSS 2.0 Documentation,” InfluxData, [Online]. Available: https://docs.influxdata.com/influxdb/v2.0/write-data/no-code/scrape-data/. [Accessed 19 October 2021].

[139] “influx - InfluxDB command line interface | InfluxDB OSS 2.0 Documentation,” InfluxData, [Online]. Available: https://docs.influxdata.com/influxdb/v2.0/reference/cli/influx/. [Accessed 19 October 2021].

[140] “InfluxDB v2 API | InfluxDB OSS 2.0 Documentation,” InfluxData, [Online]. Available: https://docs.influxdata.com/influxdb/v2.0/reference/api/. [Accessed 19 October 2021].

[141] “InfluxDB line protocol reference | InfluxDB OSS 1.8 Documentation,” InfluxData, [Online]. Available: https://docs.influxdata.com/influxdb/v1.8/write_protocols/line_protocol_reference/. [Accessed 19 October 2021].

[142] “Line protocol | InfluxDB OSS 2.0 Documentation,” InfluxData, [Online]. Available: https://docs.influxdata.com/influxdb/v2.0/reference/syntax/line-protocol/.

[143] “Empowering App Development for Developers | Docker,” Docker, [Online]. Available: https://www.docker.com/. [Accessed 19 October 2021].

[144] “Postman API Platform | Sign Up for Free,” Postman, [Online]. Available: https://www.postman.com/. [Accessed 19 October 2021].

[145] “Apache JMeter - Apache JMeter™,” The Apache Software Foundation, [Online]. Available: https://jmeter.apache.org/. [Accessed 19 October 2021].

[146] “Histogram visualization | InfluxDB Cloud Documentation,” InfluxData, [Online]. Available: https://docs.influxdata.com/influxdb/cloud/visualize-data/visualization-types/histogram/. [Accessed 19 October 2021].

[147] “angel3_orm | Dart Package,” Dukefirehawk Consulting, 29 July 2021. [Online]. Available: https://pub.dev/packages/angel3_orm/score. [Accessed 19 October 2021].

[148] T. Osakwe, “angel_cors | Dart Package,” 7 February 2019. [Online]. Available: https://pub.dev/packages/angel_cors. [Accessed 19 October 2021].

[149] “angel3_cors | Dart Package,” Dukefirehawk Consulting, 10 July 2021. [Online]. Available: https://pub.dev/packages/angel3_cors. [Accessed 19 October 2021].

[150] “Code Quality and Code Security | SonarQube,” SonarSource, [Online]. Available: https://www.sonarqube.org/. [Accessed 19 October 2021].

[151] “GitHub - insideapp-oss/sonar-flutter: SonarQube plugin for Flutter / Dart,” [Online]. Available: https://github.com/insideapp-oss/sonar-flutter. [Accessed 19 October 2021].

[152] “Kotlin Programming Language,” Kotlin Foundation, [Online]. Available: https://kotlinlang.org/. [Accessed 19 October 2021].

[153] “Jakarta® EE | Cloud Native Enterprise Java | Java EE | the Eclipse Foundation | The Eclipse Foundation,” Eclipse Foundation, [Online]. Available: https://jakarta.ee/. [Accessed 19 October 2021].

[154] “Dart packages,” Google, [Online]. Available: https://pub.dev/. [Accessed 19 October 2021].

[155] “Welcome to Python.org,” Python Software Foundation, [Online]. Available: https://www.python.org/. [Accessed 19 October 2021].

[156] M. Gerken, “What is the BLoC pattern in Flutter?,” [Online]. Available: https://www.flutterclutter.dev/flutter/basics/what-is-the-bloc-pattern/2021/2084/. [Accessed 19 October 2021].

[157] “bloc | Dart Package,” The Bloc Community, 26 September 2021. [Online]. Available: https://pub.dev/packages/bloc. [Accessed 19 October 2021].