FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Exploring the Scala Macro System for Compile Time Model-Based Generation of Statically Type-Safe REST Services

Filipe R. R. Oliveira

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Hugo Sereno Ferreira

Co-supervisor: Tiago Boldt Sousa

July 2015

Exploring the Scala Macro System for Compile Time Model-Based Generation of Statically Type-Safe REST Services

Filipe R. R. Oliveira

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Doctor Jorge Manuel Gomes Barbosa

External Examiner: Doctor Ricardo Jorge Silvério Magalhães Machado

Supervisor: Doctor Hugo José Sereno Lopes Ferreira

July 2015

Abstract

The amount of web services has increased dramatically over the past years due to the growth of mobile applications that use them, and to the business potential that they offer (e.g. Google Maps API, Facebook API). REpresentational State Transfer (REST) is a prolific architectural style used as an interface to these web services, mainly due to its better performance (when compared to other techniques like SOAP), scalability and simplicity. Some common usages of this style include the simple manipulation of structured data, by using default behaviors that are based on user defined models and which provide default CRUD operations. Known implementations are Django REST framework, Eve, Sails, and the LoopBack framework.

These frameworks apply the models' logic in run-time usually using reflection or in-memory data structures, and are commonly written in programming languages that are based on (or at least optionally support) dynamic type systems. Although easier to write and more flexible, such technical choices usually hinder two software quality attributes: performance (due to run-time adaptation) and maintainability (due to absence of compile-time guarantees).

Assuming such observations and potential shortcomings, the obvious follow-up would be to explore more efficient (and correct) solutions, by interpreting the models at compile-time using a programming language with a sufficiently powerful static and extendable type system. Scala meets these requirements, while still providing enough flexibility to developers looking for it in this kind of framework. With that in mind this work aimed to explore the design and implementation of such frameworks using the Scala Macro System.

The result of this research was materialized in the Metamorphic framework, which by using a semantically rich DSL is capable of generating an entire application from data storage to services logic. Such generated applications mostly contain components of well known and tested libraries, following either an interface or a model.

Evaluation was executed by performing both quantitative benchmarks and qualitative analysis of Metamorphic comparing with other frameworks.

Resumo

A quantidade de serviços web tem crescido dramaticamente nos últimos anos devido ao crescimento de aplicações móveis que os usam, e ao potencial de negócio que representam (por exemplo Google Maps API, Facebook API). REpresentational State Transfer (REST) é um promissor estilo arquitetural usado como interface para estes serviços web, principalmente devido à sua melhor performance(quando comparado com outras técnicas como SOAP), escalabilidade e simplicidade. Alguns usos comuns deste estilo são a simples manipulação de estruturas de dados, usando comportamentos por omissão baseados em modelos e que implementam operações CRUD. As *frameworks* Django REST, Eve, Sails and LoopBack são exemplos de tal.

Estas *frameworks* aplicam a lógica associada aos modelos em tempo de execução usando reflеção ou estruturas de dados em memória, e são normalmente implementadas em linguagens de programação que são baseadas (ou pelo menos suportam) sistemas de tipos dinâmicos. Apesar de permitirem uma escrita mais rápida e flexível, tais questões técnicas levam à degradação de dois aspetos da qualidade do *software*: performance (devido à adaptação em tempo de execução) e manutenção (devido à falta de garantias em tempo de compilação).

Assumindo tais observações e potenciais desvantagens, o caminho óbvio seria explorar soluções mais eficientes (e corretas), interpretando os modelos em tempo de compilação e usando uma linguagem de programação com sistema de tipos estático, poderoso e extensível. Scala vai de encontro com estes requisitos, para além de que providencia uma flexibilidade que programadores normalmente procuram neste tipo de *frameworks*. Sendo assim, este trabalho procurou explorar o desenho e implementação *frameworks* do género usando o sistema de macros de Scala.

O resultado desta investigação foi materializado na *framework* Metamorphic, que através de uma DSL semanticamente rica é capaz de gerar uma aplicação inteira desde o armazenamento de dados até à lógica dos serviços. As aplicações geradas são compostas maioritariamente por componentes de bibliotecas, bem conhecidas e testadas, que implementam alguma interface ou estam de acordo com algum modelo.

A avaliação foi efetuada através de testes quantitativos (performance) e testes qualitativos que colocam frente a frente a *framework* Metamorphic e todas as outras do género.

Acknowledgements

As you may know, I'm not a man of many words, so...

I would like to thank my supervisor Hugo Sereno Ferreira and co-supervisor Tiago Boldt Sousa by their always useful input. To Eugene Burmako, the face of Scala macros, from École Polytechnique Fédérale de Lausanne that kindly reviewed most of this document.

Special thanks to my "laboratory" friends: Alexey Seliverstov, Joaquim Barros, Tiago Azevedo, and Tiago Costa. To Rui Gonçalves, Luís Fonseca, and Bruno Maia at ShiftForward that took some of their time to help me. And at last but not the least, to all the people that made chapter 6 of this document possible: Alexey Seliverstov, Diogo Pinela, Joaquim Barros, Jorge Costa, Manuel Pereira, Tiago Azevedo, Tiago Costa, and Tiago Vieira.

Filipe R. R. Oliveira

"If we wait until we're ready,
we'll be waiting for the REST of our lives."

Lemony Snicket

Contents

List of Figures

LIST OF FIGURES

List of Tables

List of Sources

LIST OF SOURCES

Abbreviations

API	Application Program Interface
AST	Abstract Syntax Tree
CORS	Cross-Origin Resource Sharing
CRUD	Create, Read, Update, and Delete
DAO	Data Access Object
DSL	Domain Specific Language
DSML	Domain Specific Modeling Language
GPL	General Purpose Language
HATEOAS	Hypermedia as the Engine of Application State
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
MDE	Model-Driven Engineering
MOF	Meta-Object Facility
MVC	Model-View-Controller
MVP	Minimum Viable Product
ORM	Object-Relational Mapping
OSI	Open Systems Interconnection
REST	Representational State Transfer
RPC	Remote Procedure Call
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
UML	Unified Modeling Language
URI	Uniform Resource Identifier
WSDL	Web Service Definition Language

Chapter 1

Introduction

1.1 Context

Internet users have grown since its most widely adoption around the year of 1990, having reached the total number of 1 billion users in 2005 and estimates indicate that by 2015 this number will triple [Soc]. This growth happened mostly due to its dissemination in developing countries, as they represent more than 50% of users, and due to the appearance of smartphones around 2007, as they represent more than 50% of mobile phones. Meanwhile, other new devices that connect to the internet have emerged such as tablets, smart TVs, and smart watches. Also the internet of things concept [Ash09] starts to gain some shape with estimates indicating that 30 billion devices will be wirelessly connected by 2020 [Res13].

These users and devices stay connected and explore their potentialities through the consumption of web services, such as static or dynamic web pages, mobile applications content, and real-time services like chats and notifications. In fact, web services became so important that companies started to open their services to third-parties, such as the Facebook Graph API [Fac] and the Google Calendar API [Goo]. Others even created businesses only based in services, such as Parse [Par] and Mailgun [Rac].

Two common architectures for implementing these services were the remote procedure call (RPC) and the service-oriented architecture (SOA) [KFB14]. The RPC architecture hides network communication making services' calls look like as if they were in the same program. However, it has issues that result from its own definition such as tight coupling, inconsistent states due to network problems, and being hard to scale. The SOA approach has been explored through the web services description language (WSDL) and the simple object access protocol (SOAP), being the second most accepted as it solves many of RPC's issues.

In the same time-frame of SOAP's specification, as explained in section 2.5, Roy Fielding defined the REST architectural style [Fie00] to be applied in distributed hypermedia systems. The style defines a set of six constraints: client-server, stateless, cache, uniform interface, layered system, and code-on-demand. The uniform constraint is based on other four constraints that define

the concept of resource, and that it must be uniquely identifiable and manipulated using self-descriptive messages. All these constraints enable scalability, portability, visibility, and simplicity in exchange for some degraded efficiency and reliability. It is normally preferred to the SOAP approach [KFB14] as it achieves better performance most of the times [PAUP12].

In practice REST is usually implemented recurring to URI for resource identification, and to HTTP for stateless client-server communication, using messages with semantics that also allows caching. This sometimes leads to wrong opinions about the style, such as thinking that any HTTP-based web API is RESTful. With that in mind, as explained in section 2.5, the Richardson maturity model defines four levels to achieve RESTful web services.

Despite the architecture of web services there has been a need to implement more complex and robust services, which lead to the development of several web frameworks. These aim to better structure implementations and to provide solutions for recurrent problems - most using a model-view-controller (MVC) approach [Jaz07] - by raising sometimes the level of abstraction. Consequently, the set of possible implementation errors is reduced, that together with the conceptual simplicity enables faster system's development.

A current common practice in web based development is the separation of concerns between the back-end and client applications which also enables the existence of more specialized teams. This has also motivated the implementation of web services that deliver CRUD operations for a given set of model entities. The nature of these services obliges the existence of repeated code, even when using most traditional frameworks. Framework developers saw in this problem an opportunity to deliver more value to their users, by supporting the implementation of model-based services such as the popular Django REST framework [Chr]. As explained in section 2.1, this approach has the following advantages: short-time-to-market, fewer bugs, increased reuse, and easier-to-understand up-to-dated documentation.

1.2 Motivation

In general, current model-driven REST frameworks do in fact solve the boilerplate code problem[1] but due to implementation decisions there are two main problems.

Firstly, they are mostly implemented in dynamically typed languages such as Python and JavaScript. As these kind of languages don't require the use of explicit types, type-related errors happen more often. This fact combined with the forced dynamic typechecking, leads to more and longer debugging sessions. Strongly and statically typed languages reduce substantially this problem and consequently enable better performed services, as most compilers are capable of implementing optimizations based on types.

Secondly, these frameworks implement model-based generation through the inspection of variables or introspection (section 2.3.1) for collecting the schema, and through parameterized functions or intercession (section 2.3.1) for responding to requests. All this work is done at run-time, increasing the program's setup time or even the response time to requests.

[1] Repetition of code with minimal changes is hard to maintain

1.3 Research Problem

Having identified two main problems (section 1.2) with current model-driven REST frameworks, a question can be raised:

> *Consider a statically type-safe programming language that enables generation of REST services in compile time. Can a model-driven REST framework written in this language improve the development process and execution performance when compared to current ones?*

Scala [EPF] can be used as proof of concept to answer this question. This language, that was built with scalability in mind, "is a Java-like programming language which unifies object-oriented and functional programming" [OAC+04]. It offers a strong static type system, and an easy capacity for compile-time generation, through macros. These characteristics promise that developers may be able to implement their model-driven REST services even faster and with more robustness, as type errors may be identified sooner.

1.3.1 Main Goals

The result of this dissertation is expected to accomplish the following topics and answer the identified questions:

1. **Research on REST frameworks.** Which frameworks' features enable creation of REST architectures? Which other features do frameworks provide? What is the importance of these features? How are they incorporated in frameworks' architectures? It is important to understand what has been done, how has been done, and whether has been well done in order to achieve a complete and correct solution. The result of this research has been resumed in Chapter 3.

2. **Research on model-driven development.** How to capture domains based on models? Which web frameworks enable model-driven REST development? How is the model-driven part implemented? In which programming languages are they implemented on and are they dynamically typed? This confirms the premises of this work and may reveal good practices for model-driven approaches. The result of this research can be found in sections 2.1, 2.2, and 3.3.

3. **Development of a model-driven REST framework using Scala macros.** Which Scala frameworks already exist? What are their advantages and disadvantages? Do they have all identified features? Which extensions do they need? Does any library already implement these extensions? Are these libraries good solutions? How can the Scala macro system be explored to implement model-driven development? Is modeling required to be done via a DSL? The use of proven solutions for general framework features concentrates the focus in the model-driven problem, for statically typed languages. The use of the Scala macro

system is merely the means to a proof of concept. Specification of such framework can be found in Chapter 4.

4. **Evidence of the framework benefits.** For the same domain problem. Does the framework require less coding than existing ones? Does it induces developers to do less type errors? Is the development time reduced? Is the response time of services decreased? Validation of the framework is accomplished through benchmarks against all the identified model-driven frameworks, and through a controlled academic quasi-experiment against only one of these, by human and time resources limitations. The results of these experiments can be found in Chapters 5 and 6.

5. **Contribute to the community.** The developed framework will be distributed as open-source with the means to receive feedback from non-academic experts. This may also benefit developers in their work or be the source of motivation to explore Scala macros beyond current usage.

1.3.2 Framework Scope

For this dissertation, the scope of this research aimed to implement a framework with model-based generation with some degree of customization. The framework must support:

- **Simple field types.** Strings, integers, floating-point numbers, booleans, dates, and date-times. These may be defined as optional.

- **Relations between entities.** One to one, many to one, and many to many relations. These should not require indication of foreign keys or new tables.

- **Adequate routing.** The base URLs for operations on collections and instances of entities must be automatically computed. GET may be used with collections and instances of entities. POST only with collections. PUT and DELETE only with instances. By default all operations for an entity should be enabled, but should exist the possibility to individually disable them.

- **Customization.** Default implementation of operations may be overridden, in which case a data storage implementation should be available to use.

- **Server configurations.** Host, port, and database configurations.

1.3.3 Research Methods

Zelkowitz and Wallace [ZW98], based on a categorization of research methods for science in general, defined a taxonomy that identifies twelve different experimental approaches to be applied for software engineering. These approaches can be categorized in three broader groups as quoted:

- **Observational methods.** "An observational method will collect relevant data as a project develops. There is relatively little control over the development process other than using the new technology that is being studied." The approaches are: project monitoring, case study, assertion, and field study.
- **Historical methods.** "An historical method collects data from projects that have already been completed. The data already exists; it is only necessary to analyze what has already been collected." The approaches are: literature search, study of legacy data, study of lessons-learned, and static analysis.
- **Controlled methods.** "A controlled method provides for multiple instances of an observation in order to provide for statistical validity of the results. This is the more classical method of experimental design in other scientific disciplines." The approaches are: replicated experiments, synthetic environment experiments, dynamic analysis, and simulation.

1.3.4 Validation Methodology

Besides all the logic argumentation that leads to the development of the envisioned framework there has to be a way to empirically validate the concepts. This shows practical viability, enforces acceptance of new knowledge, and may encourage the exploration of other related works. For this dissertation in specific, validation intents to verify some characteristics in the developed framework:

- **Quick and easy to use.** Developers that look for these kind of frameworks want to have a minimum viable product (MVP) as soon as possible without having to explore all the framework's documentation.
- **Error preventive.** Statically typechecked programs reassure developers about their code quality and reduce frustration when debugging.
- **Better response times.** Due to its programming language origin, improvements in performance should be noted when compared with existing model-driven frameworks.

These characteristics present two distinct natures: the first and second are related with the developer's interaction and perception concerning the framework, while the last is related with the performance impact of the framework when the services are used by end-users. The developer related validation was pursued using *synthetic environment experiments* (section 1.3.3), executed in the form of an academic quasi-experiment, within some time and human resources restrictions. End-users' validation was conducted through *dynamic analysis* (section 1.3.3).

1.4 Outline

This document is structured in seven chapters, starting by this one that exposes the foundations of the proposed work, what was expected to achieve, and how it was validated.

The chapter *Background* (2) is an overview on a set of concepts that are the basis to the identified research problem such as model-driven engineering, metaprogramming especially in Scala, type systems, and REST. In chapter *REST Frameworks* (3) are described common problems when building frameworks, the features that can be implemented, and the state of the art in REST frameworks (model-driven and in Scala).

Next, *A Model-Driven REST Framework in Scala* (4) describes the process followed to implement the proposed framework, the architectures behind it, including a DSL, and how it was verified. In chapter *Benchmarks* (5) it is described one of the validation strategies that aimed to compare performances with current model-driven frameworks. Another validation strategy consisted in creating an *Academic Quasi-Experiment* (6). At last, *Conclusions* (7) summarizes the document, exposes contributions and proposes some possible future work.

Chapter 2

Background

This chapter explores some concepts underlying the identified problem, giving an overview to first-learners instead of exhaustively exploring them which can be achieved by following the given references.

As basis model-driven development, its advantages and disadvantages are explained in section 2.1, which uses domain specific modeling languages defined in section 2.2, where is also justified the wide adoption of internal languages. Model-driven development is often done through the use of metaprogramming techniques such as reflection and macros that are characterized in general and in particular for Scala in section 2.3.

In section 2.4 differences between static and dynamic typing and typechecking are referred by which it's possible to identify the benefits of having statically type-safe services, besides the model-driven approach. In section 2.5, REST and the possible degrees of RESTfulness of web services are defined that is normally implemented using HTTP. The most relevant features of this protocol when applied to the REST specification are identified in section 2.6.

2.1 Model-Driven Engineering

The model-driven engineering (MDE) concept appears as a solution to the growth of complexity in system architectures and the nonexistence of programming languages capable of reducing this complexity, using an effective model approach. So the purpose of MDE is to reduce the gap between problem definition that can be defined with high-level models and software implementation.

In order to do this, and as illustrated in Figure 2.1, MDE first uses domain-specific modeling languages (DSML) to obtain "the application structure, behavior, and requirements within particular domains" [Sch06]. The declarative nature of the specification gives emphasis to domain semantics and constraints. After this by using the domain information, transformation engines and generators several types of artifacts can be produced that implement fully or partially the target system, such as source code or executable files. This second phase is composed by two main

activities [LFA+05]:

(i) *domain engineering* produces artifacts that implement generic functionality that is independent from the domain instance; (ii) *application engineering* reuses the result from the first activity to build final artifacts that depend on the domain instance.

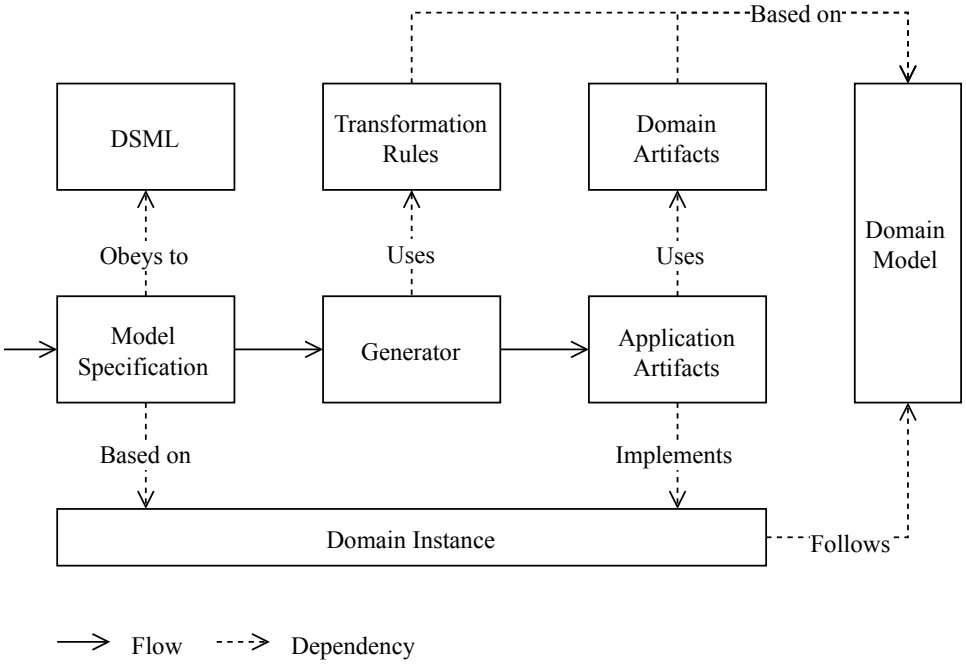


Figure 2.1: Model-driven engineering approach

The MDE approach has several advantages over traditional ones [RFBLO01]:

1. **Shorter time-to-market.** As the main focus is domain definition that is a closer concept to end-user needs than implementation details;

2. **Increased reuse.** Not only code reuse is enforced in the application engineering activity but also different domain models can be generated using the same base code;

3. **Fewer bugs.** Most bugs occur at a higher level of abstraction than traditionally, meaning less chances of bugs, that when detected and corrected have an impact on all lower levels;

4. **Easier-to-understand systems and up-to-date documentation.** As the system design has a direct mapping to the implementation, using its semantics documentation can be easily produced.

On the other hand it may be hard to explore and simulate different models due to code generation from models, compilation, system installation, configuration and restart.

2.2 Domain Specific Languages

Domain specific languages (DSLs) allow programmers to express easily their solutions in a format that is near to the problem's domain than a general purpose language (GPL). Quoting [VDKV00]:

> "A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain."

Normally DSLs are small as they attempt to include rich semantics in syntax which also leads to restricted expressiveness. The advantages of these languages when compared to GPLs are highly correlated with the advantages of using MDE (section 2.1). They mostly comprise the consequences of raising the level of abstraction, such as, programs easier to understand, test, and modify that result in more productivity. On the other hand: (i) solutions will have additional costs for the design, implementation and specially maintenance of these languages; (ii) and, may not be easy to define the scope of the language when the domain is not well defined or may evolve in time.

A DSL is considered a domain specific modeling language (DSML) when the used abstractions represent solely models [SRGT14]. The specification of this models follow a model that is denominated meta-model (e.g. UML specification), which dictates what can be expressed in models. Normally another level of abstraction is also applied to meta-models, meaning that these also should follow some structure denominated metameta-model (e.g. MOF[1]) [AZW06].

Considering only programming languages, DSLs were built just like GPLs, beginning with the definition of a grammar, then the implementation of a parser and sometimes integration with IDE's. So the development of a DSL was hard which lead to the appearance of internal languages that reuse a host language syntax for expressing the targeted semantics.

2.3 Metaprogramming

Metaprogramming can be described as the process used for building (meta-)programs, that are aware of the structure and behavior of other programs either by producing or by manipulating them [CI84]. In spite of being common practice to write meta-programs that are also the target program nothing forces them to be written in the same language. The approach can be used to generate programs that consist of well-known repetitive patterns, to enable frameworks with easy to use interfaces, or in a more general sense, to empower program's capabilities.

2.3.1 Reflection

Reflection is the ability of a program to manage structures that keep track of its state and behavior during execution. The ability to acknowledge and reason about the program's state is called *introspection* while the ability to change the state or its interpretation is called *intercession* [BGW93].

[1] Meta-Object Facility [Obj] is the meta-model used by UML

This property enables the development of programs with higher complexity, such as, the generation of programs by using reflection in a internal DSL (sections 2.2 and 2.3).

2.3.2 Macros

A macro is a feature related to programming languages that produces code based on a high-level pattern language [CR91]. It is used to extend languages' capabilities in general, being often associated with the reduction of boilerplate code as it started to be used in low-level languages, such as Assembly, C, and Common Lisp. Macros can be expanded at compile-time or before compilation, normally using preprocessors as in the C language (Source 2.1).

```
1 #define PI 3.14159
2 float circle_perimeter(float radius)
3 {
4   return 2 * PI * radius; // transformed to 2 * 3.14159 * radius
5 }
```

Source 2.1: Example of a simple macro in C

2.3.3 Metaprogramming with Scala

Metaprogramming for Scala is still experimental and is currently provided by the scala.reflect API and the macro paradise compiler plugin [Bur14]. The scala.reflect API is available since version 2.10 of Scala and was built to support full run-time reflection, as the the Java reflection API only exposed Java elements, and to support compile-time reflection in the form of macros [MBH]. The macro paradise plugin adds functionalities to the scala.reflect API and is distributed separately from the official Scala distribution, as it is used for experimentation of new features [Bur14]. The following sections 2.3.3.1 and 2.3.3.2 present how reflection and macros can be explored in Scala.

2.3.3.1 Reflection

The scala.reflect API can be used for run-time or compile-time reflection. Run-time reflection allows inspection of a object's type, including generic types, instantiation of new objects, and access or invocation of that object's members. On the other hand, compile-time reflection is used for manipulation of a program's abstract syntax tree (AST), types, and symbols.

In other to achieve run-time reflection of a given compile-time type T the compiler encapsulates all of its information in a TypeTag[T] type. This happens whenever is used an implicit parameter or a context bound that will translate in a implicit TypeTag[T] like in Source 2.2. The API defines the concept of symbol as the binding between names and entities, such as classes or methods.

```
1  import scala.reflect.runtime.{universe => ru}
2  def getType[T: ru.TypeTag](obj: T) = ru.typeOf[T]
3  // getType: [T](obj: T)(implicit evidence$1: ru.TypeTag[T]) ru.Type
4
5  val list = List(1,2,3)
6  val listType = getType(list)
7  // ru.Type = List[Int]
8
9  val decls = listType.declarations.take(3)
10 // Iterable[ru.Symbol] = List (constructor List, method companion, method isEmpty)
```

Source 2.2: Access to a object's type and its declarations

Access to packages, objects, classes, fields, methods, variables, and instances are achieved using mirrors. For example, the continuation of Source 2.2 in Source 2.3 describes the steps to call the method isEmpty on the original list using a classloader mirror, Mirror, and two invoker mirrors, InstanceMirror and MethodMirror.

```
1  val rMirror = ru.runtimeMirror(getClass.getClassLoader)
2  // ru.Mirror = JavaMirror ...
3
4  val iMirror = rMirror.reflect(list)
5  // ru.InstanceMirror = instance mirror for List(1, 2, 3)
6
7  val mMirror = iMirror.reflectMethod(decls.last.asMethod)
8  // ru.MethodMirror = method mirror for def isEmpty: Boolean (bound to List(1, ...
9
10 mMirror()
11 // Any = false
```

Source 2.3: Method invocation through reflection

2.3.3.2 Macros

"Macros are functions that are called by the compiler during compilation" [Bur]. These functions can generate, analyze and typecheck code. They also have the capability to abort compilation or report warnings.

The most basic flavor of Scala macros are the *def macros* that are methods expanded at compile time, transforming their declaration into code with similar interface. These macros can be defined "either inside or outside of class, can be monomorphic or polymorphic, and can participate in type inference and implicit search" [Bur13]. As these macro calls have nothing different from normal method calls many features can be empowered without users even noticing of their use.

```
1  def printf(format: String, params: Any*): Unit = macro impl
2  def impl(c: Context)(format: c.Expr[String], params: c.Expr[Any]*):
       c.Expr[Unit] = {
3    ...
4    c.Expr[Unit](q"print($result)")
5  }
6
7  printf("Hello %s!", "John") // Hello John!
```

Source 2.4: Example of a def macro

The implementation parameters and result are of type c.Expr[T] instead of any original type T, as in Source 2.4. It is possible to access to a context in which the macro is being used, enabling more flexibility. The use of quasiquotes[2] increase readability, just like in the line four of the example. Other types besides Expr may be used for building an AST such as ClassDef, ModuleDef, ValDef, DefDef, TypeDef, Literal, Constant, and more. This allows parameterized implementation of type providers as shown in Source 2.5.

```
1  def h2db(connString: String): Any = macro ...
2  val db = h2db("jdbc:h2:coffees.h2.db")
3
4  // expands into
5  // val db = {
6  //   trait Db {
7  //     case class Coffee(name: String, price: Decimal)
8  //     val Coffees: Table[Coffee] = ...
9  //   }
10 //   new Db {}
11 // }
12
13 db.Coffees.all
14 // List[Db$1.this.Coffee] = List(Coffee(...))
```

Source 2.5: Example of a type provider def macro [BOV+13]

Another macro flavor is *implicit macros* that allows methods to be called without having the users write the calls explicitly, through, for example, materialization of type class instances and implicit conversions. These parameters are inferred from the current scope based on the type of the target, that must be declared with the keyword *implicit* [Bur13]. When the use of this kind of macros leads to boilerplate code, the combination with def macros can solve the problem as shown in Source 2.6.

[2] String interpolators that build code

```
trait Showable[T] { def show(x: T): String }
def show[T](x: T)(implicit s: Showable[T]) = s.show(x)

implicit def materialize[T]: Showable[T] = macro ...

show(person)
// show(person)(materialize[Person]) --- implicit macro expansion
// show(person)(new Showable[Person] { ... }) --- def macro expansion
```

Source 2.6: Materialization with implicit macros [BOV+13]

The last major macro flavor is *macro annotations* that can be used to transform definitions such as classes, objects and fields. They allow not only the expansion of classes but also to create or modify companion objects as applied in Source 2.7. However expansions are constrained as expanded top-level classes and objects must have the same name as the annottee. Annotated expressions are only typechecked after expansion, enabling greater flexibility.

```
@case class C(x: Int)
// expands into
// class C(x: Int) {
//   /* standard case class methods like toString */
// }
// object C {
//   /* standard case companion methods like unapply */
// }
```

Source 2.7: Example of a annotation macro [Bur13]

2.4 Type Systems

A type system is a component of a programming language that manages the types of all expressions in a program, such as variables, and is used to avoid the occurrence of execution errors at run-time. These execution errors can be one of two kinds [Car96]:

- **Trapped errors** that cause the program to stop immediately such as a division by zero or dereferencing a null pointer;
- **Untrapped errors** are less severe by going unnoticed, such as accessing incorrectly a legal address, but cause random behavior later.

Languages that avoid all untrapped errors and a wide set of trapped errors are called *strongly typed* or *type-safe*, while languages that don't avoid all untrapped errors are called *weakly typed*. The definitions of a type system strength are many, but in resume all are based on the degree of avoided errors.

Another important aspect about type systems is time:

1. **When do errors are avoided?** Either at compile time (also know as *static typechecking*), run-time (also known as *dynamic typechecking*) or both as there are errors that can only be checked at run-time.

2. **When do variable types are known?** Types in languages can either be known at compile time or at run-time which are called *static typing* and *dynamic typing*, respectively.

This means that languages with static typing can use dynamic and static checking while dynamic typing only allows dynamic checking. The use of static checking, in particular, leads to:

- **Performance improvements.** Beside the execution of compile time checks that otherwise would occur at run-time, compilers can use information on types to do optimizations.

- **Less debugging.** A developer doesn't have to deploy the system in order to find type errors. This class of errors can be discard as source of any run-time errors.

These consequences are also related with static typing to which are added:

- **Faster compilation.** Programs can be structured into interfaces that increase independence between modules. This means that changes in one module doesn't lead to recompilation of the all system, that is specially important in large systems.

- **More and verbose code.** Types information is mainly described in source code increasing verbosity, which has been a cause for creating dynamically typed languages.

2.5 Representational State Transfer

Representational state transfer (REST) is an architectural style for distributed hypermedia systems defined by Roy T. Fielding [Fie00] in 2000. Fielding was involved in the development of HTTP/1.0, and while working in HTTP/1.1 and in the URI design, he noticed that web applications' architectures could be better guided. So the style is defined as a set of constraints:

- **Client-server.** Applies the separation of concerns principle between data and graphical interfaces. This allows portability of graphical interfaces across platforms, server simplicity that increases scalability, and independent evolution of components.

- **Stateless.** It means that the server doesn't keep any client context information that, if needed, should be provided by the request. This constraint improves visibility as information is restricted to a single request. It improves reliability as recovering from partial failures is less dependent. As by the client-server constraint, not having to manage states, improves simplicity that increases scalability. It has a small trade-off because requests will contain repetitive data reducing network efficiency.

- **Cache.** When responses to requests are labeled as cacheable, clients can reuse these responses later for similar requests. This constraint may help remove some client-server interactions which reduces average latency, and therefore improves efficiency, scalability and user-perceived performance. On the other hand, reliability is decreased as cached data may differ significantly from up-to-date data.

- **Uniform interface.** Applies the generality software engineering principle, meaning that interfaces should be generally defined so that services are independent from implementation. This allows evolvability and increases visibility but may decrease efficiency as communication is done in a standardize form that may not be optimal.

- **Layered system.** Each layer only has knowledge of layers that uses and no knowledge of layers that use it. This constraint reduces system complexity by stratifying in independent layers that may support, among others, legacy services, shared cache, and load balancing. This results in more scalability but due to layers efficiency of each request is reduced, effect that can be decreased by shared cache that reduces average latency.

- **Code-On-Demand.** It is an optional constraint that allows client functionality to be available in servers, which allows code reuse across clients platforms or even extensibility of features after client's deployment.

Besides these general architecture constraints, Fielding also defines a uniform interface as a set of four constraints: "identification of resources; manipulation of resources through representations; self-descriptive messages; and, hypermedia as the engine of application state" (HATEOAS) [Fie00]. A *resource* is an abstract instance of any concept that can be uniquely identified. A *representation* is in fact data (sequence of bytes) that mirrors a resource or it used to perform actions on resources. Lastly *hypermedia* allows data-guided navigability in client applications by the existence of links in representations.

Despite of no mention to a specific protocol in the definition, HTTP (specified in section 2.6) is highly associated with the style as it is the one that more strictly enables the constraints. This and the high-level of abstraction in the definition sometimes leads to misinterpretations of the concept [Fie14, Fie08b, Fie08a] that it is often wrongly used when speaking of web APIs in general. Such errors motivated the definition of the Richardson maturity model that allows the classification web services towards RESTful state. As the Figure 2.2 illustrates, this model divides web services in four levels [WPR10]:

- **Level Zero.** Services are provided using a single HTTP method on a single URI. These usually use the POST method as is example the XML-RPC protocol, or some systems that use SOAP-based communication.

- **Level One.** In this level several URIs are used but each with a single HTTP method. This means URIs expose several logical resources, in contrast with level zero, but also tend to identify operations.

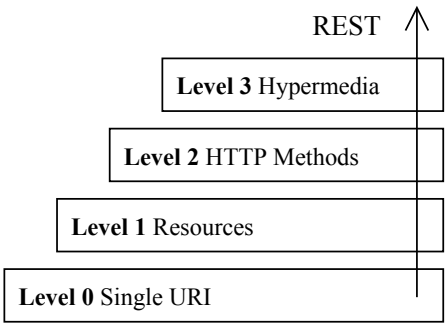


Figure 2.2: Richardson maturity model

- **Level Two.** In this level URIs are built based on resources that can be operated using different HTTP methods, each according with its semantic as specified in section 2.6.1.
- **Level Three.** The last level guarantees that web services are compliant with the REST specification, by simply adding the HATEOAS constraint to level two.

2.6 Hypertext Transfer Protocol

Hypertext transfer protocol (HTTP) is an application layer protocol in the OSI model and is the basis of data transfer in the World Wide Web. Its first version was documented in 1991 and has two major versions: 1.0 released in 1996; and 1.1 released in 1997, improved in 1999 and 2014. All references to the protocol in this document are related with the 2014 update.

Hypermedia resources are identified through URIs that may follow a *http* or *https* scheme, being the last one for secure connections. Messages are composed by a start line, a set of headers, and a message body [FR14c]. The start line of a request contains a method section 2.6.1 and a resource URI while responses contain a status code that describes a server's attempt to answer requests. The headers contain metadata about the request and the message body contains resource representations.

The following sections describe some of the most relevant aspects of the protocol but for full documentation refer to [FR14c, FR14d, FR14b, FLR14, FNR14, FR14a, DS10].

2.6.1 Request Methods

Request methods contain the majority of semantics (Table 2.1) in a request by generally suggesting the request purpose and its expected response [FR14d, DS10]. Responses status codes are categorized as follows: informational (1xx); successful (2xx) such as 200 (OK) and 201 (Created); redirection (3xx); client error (4xx) such as 400 (Bad Request), 403 (Forbidden), and 404 (Not Found); or server error (5xx) such as 500 (Internal Server Error).

The methods GET, HEAD, and OPTIONS are designated safe because they all are read-only, so the client is expecting that no change is going to be made in the server due to requests with

these methods. Safe methods, PUT, and DELETE are designated idempotent because the effect on the server of multiple identical requests with these methods is the same as the effect of a single also identical request.

METHOD	SEMANTIC
GET	Retrieve a current representation of the target resource
HEAD	Same behavior as GET but without returning a payload
POST	Operate on the target resource by processing the request payload
PUT	Replace the complete state of the target resource with the request payload
PATCH	Same as PUT but without the complete constraint
DELETE	Remove any representation of the target resource
OPTIONS	Retrieves information about available communication options

Table 2.1: HTTP request methods' semantics

2.6.2 Media Types

Internet media types [FB96] are composed by: a type that specifies a general type of data, such as *image* and *text*; a subtype that indicates a specific data format, such as *png* and *html*; and a set of parameters that are modifiers of the subtype, such as *charset=utf-8*.

Media types are used in the Content-Type and Accept headers. The Content-Type header indicates the payload format to message readers while the Accept header is specified by clients as the acceptable format to be returned by the server.

2.6.3 Conditional Requests

Requests are said conditional when include one or more headers that indicate a precondition to be validated before initiating any method logic [FR14b]. Two type of preconditions are defined mainly to solve concurrency, and act on resources' modification dates or on the current entity-tag.

In order for clients to know the exact date and time at which their current resource representation has taken effect, servers should respond to modifiable requests with a Last-Modified header (Source 2.8). With that knowledge clients can make requests with the If-Modified-Since (Source 2.9) or the If-Unmodified-Since headers to apply the logic that these headers' names implicitly express.

```
1 HTTP/1.1 200 OK
2 ETag: "some-entity-tag"
3 Last-Modified: Wed, 23 Jan 2015 10:14:23 GMT
4 ...
```

Source 2.8: Example of a HTTP response

A response can contain the ETag header (Source 2.8) which is the most recent entity-tag for the returned representation. An entity-tag is a quoted string defined by the server that allows differentiation between different representations of a resource. Requests that send an entity-tag in the If-Match header are expected to be answered successfully only if the current entity-tag in the server matches, making it useful, for example, to disallow client updates of resources that are based on older representations. Requests can also use the If-None-Match header (Source 2.9), for example, to update client's current representation of resources.

```
1 GET /index HTTP/1.1
2 Host: www.example.com
3 If-None-Match: "some-entity-tag"
4 If-Modified-Since: Wed, 23 Jan 2015 10:14:23 GMT
5 ...
```

Source 2.9: Example of a conditional request

2.6.4 Caching

As described in section 2.5 caching [FNR14] can be used for improving performance through response reuse, and as so HTTP eases its implementation. One way of specifying caching is by using the Expires header that a server can set with a specific date and time after which the response will be considered invalid.

Other option is through caching directives that can be defined either by a requester (client or proxy) or a server, by using the Cache-Control header. A relevant directive that a server can specify is max-age that indicates the number of seconds the response should be considered valid.

After a cached response is consider invalid, validation can be done through conditional requests (section 2.6.3), allowing no transfer of a representation if the one in cache is still up-to-date.

2.6.5 Authentication

The protocol also has the means to authenticate requests [FR14a] as security of information is fundamental for many of today's web-based applications.

When authentication is required and not done or wrongly done, servers should respond with a 401 (Unauthorized) status code and include in the response the WWW-Authenticate header containing the available authentication schemes. The four most used schemes [For15] are Basic, Digest, OAuth also know as OAuth 1.0 that before its last update gained a non official version (OAuth 1.0a), and Bearer also know as OAuth 2.0. Clients can provide credentials using the Authorization header and may receive a 403 (Forbidden) status code in case of valid authentication but invalid authorization.

Chapter 3

REST Frameworks

As one of the main goals of this dissertation is the implementation of a model-driven REST framework in Scala, this chapter gives some insight on: how build frameworks section 3.1; what are the features that usually are needed and implemented, in section 3.2; which current frameworks have a model-driven approach and what is the architecture associated, in section 3.3; and which Scala frameworks provide REST functionalities, in section 3.4.

3.1 Building Frameworks

A framework is a set of abstract classes and instances of those classes that collaborate in order to solve a set of related problems that increases reuse as it raises the level of abstraction [JF88]. A good framework must be simple to understand but also support enough customizable and quick-to-use features so that it can be reused in as many applications as possible. These characteristics make frameworks hard to develop as most times (i) the domain is somehow fuzzy being difficult to find the correct abstractions, (ii) tuning the design requires evaluation with concrete situations, or (iii) the base examples don't represent the whole domain limiting generality and reusability [FAF09].

Building high-quality frameworks is usually the result of many design iterations [WBJ90] and as so there have been proposed some methods in order to reduce the number of these iterations. In general there are two kinds of approaches: *bottom-up* that starts with concrete applications and iteratively abstracts concepts into the framework; and *top-down* which relies on domain knowledge, that in this case would be what is described in section 3.2.

3.2 Features

Frameworks usually grow from necessity and with maturation. This lead to diversity not only in their design but also in the features they provide to developers. In this section are gathered most of these features:

- **HTTP I/O.** Management of HTTP messages from parsing and request encapsulation to rendering, following the protocol. Normally provided through transparent interfaces that underneath handle concurrency and distribution.

- **Data storage.** Allow easy connections to databases. First abstractions were very simple and required developers to write SQL queries like the PDO class in PHP [Gro]. Some frameworks abstract queries through functions enabling reuse across different databases, like the Schema class in the Laravel framework [Otwb]. More recently object-relational mapping (ORM)[1] has been adopted once enables flexibility to change of models, as in the ASP.NET Entity Framework [Mic].

- **Routing.** Mapping of the requests' URIs and HTTP methods to actionable code through pattern matching. This process may include parameters extraction and validation as shown in Source 3.1.

```
1 Route::get('user/{id}/{name}', function($id, $name)
2 {
3     //
4 })
5 ->where(['id' => '[0-9]+', 'name' => '[a-z]+'])
```

Source 3.1: Example of routing with the Laravel framework [Otwa]

- **Serialization.** Used to encapsulate requests information into objects or extract responses information from objects. The use of objects is usually related with the fact that they have a higher degree of semantics. Implementations allow structuring, exclusion and inclusion of information, or even transformation when dealing with composed data.

- **Parsing & Rendering.** Extraction of structured data from requests bodies (parsing) and creation of response bodies from structured data (rendering). Parsers and renderers are used accordingly with the media-types in the Content-Type or the Accept headers (section 2.6.2). The *application/xml* and the *application/json* media-types are the most commonly implemented. Some implementations also apply serialization for performance reasons.

- **Filtering.** Clients often implement search functionalities over collections of resources. When the search involves resource fields is created one service that accepts query parameters that filter the base result (e.g. `/products?isAvailable=true`). For searches that also involve other resources another service should be created that comprises distinct logic (e.g. `/categories/{categoryId}/products`). Filtering may be provided at the data management level or at the service level.

[1] Maps the state and actions of objects to database columns and queries, respectively.

- **Sorting.** In spite of representing an additional overhead on requests and being supported by data management, sorting may be provided if requested through query parameters. These parameters may be the field to order by or the sorting direction.
- **Pagination.** Some collections of resources tend to grow over time, being hard to represent in client applications. Pagination helps solve this problem as it also reduces latency. This feature has the page size or the page number as parameters that can be omitted, the first when the value is constant and the second to retrieve the first page.
- **Projection.** Different client applications may have different information needs. Instead of implementing different services that return more or less information, requests may indicate what information is needed or not. This feature is not very frequent.
- **Validation.** In order to guarantee consistency of data and prevent malicious usage frameworks allow requests to be validated. This goes from preventing that no two users have the same email, guaranteeing that an email field follows the right format, a number field is inside some range, etc.
- **HATEOAS.** Web services to be compliant with REST should use hypermedia as the engine of application state. This means that returned representations include links to related resources. Frameworks are capable until some extent to incorporate these links in responses, for example when using pagination by including the link to the previous and next page.
- **Authentication.** The authentication schemes supported by HTTP (section 2.6.5) are normally implemented by frameworks, requiring only to be plugged and characterized. When authentication is used, services may refer to an object of type user that represents the requester.
- **Authorization.** Access to services can be restricted to groups of users. By defining which users can access to which services, frameworks are capable of handling non-authorized requests without the service code even being called.
- **Conditional requests.** As specified in section 2.6.3, to implement conditional requests, responses must include the E-Tag or the Last-Modified headers and requests conditions must be verified and answered accordingly.
- **Rate limiting.** For preventing abuses in the use of web services, some restrict the amount of requests a user can do during a period of time.
- **Caching.** Is one of the REST constraints (section 2.5) and is supported by HTTP (section 2.6.4) in a way that can be abstracted by frameworks.
- **Content negotiation.** Web services may return representations using a predefined media type without obeying to the Accept header. The process to choose the used media type is called content negotiation and some frameworks provide a set of possible approaches.

- **HTTP method override.** Allows clients to use a HTTP method to call a service that is provided by another HTTP method, using the X-HTTP-Method-Override header. Normally is used by browsers that don't support the PATCH method.

- **Cross-origin resource sharing.** Most browsers impose the same-origin security policy what obliges web services to implement CORS [W3C14].

- **Testing.** In order to validate development and prepare deployment, frameworks may provide an interface for testing services without the need to have client applications implemented or real users.

- **Logging.** During development or even the production phase it is normal the occurrence of errors in services. To better identify and quickly solve any bugs, information about the environment in which those errors occurred is crucial. Frameworks can keep this information through requests' logging.

- **Documentation.** Web services consumers rely mostly in documentation to easily understand capabilities and how to explore them. This means that is important to have the implementation synchronized with documentation. Most frameworks give the means for this to happen.

- **Versioning.** It is useful for evolving APIs without having to manually change routing for each service.

3.3 State of the Art in Model-Driven REST Frameworks

Model-driven REST frameworks are REST frameworks that use model entities as resource representations (section 2.5). These usually only consider two type of resources: instances of model entities and collections of model entities.

In the following sections (3.3.1, 3.3.2, 3.3.3, 3.3.4) are presented four REST frameworks that support model-driven development. Their basic usage is described, analyzed and concluded in section 3.3.5, where their features are compared.

3.3.1 Django REST Framework

Django REST framework [Chr] was created in 2011 and is an open source framework in Python to build Web APIs, and the current version is 3.1. It is built on top of the Django framework [Foua], a tool that enables fast development of web applications, including model-driven development.

Following the quickstart tutorial [fra] it's possible to easily understand the architecture and quickly have a functional application. The framework is funded in: *views* that given requests perform necessary actions and prepare responses; *serializers* that define the structure of object's data; and *urls* that connect URLs with views.

The support for model-driven development is delivered by subclassing a base model, a base model serializer, and a base generic view as shown in Source 3.2. These subclasses override the interface methods and use reflection in order to implement the intended functionality. This adds an overhead on responses when compared with manual implementations that directly access variables without having to inspect their name in the beginning.

```
# Uses ORM from the Django framework
class Category(models.Model):
    name = models.CharField(max_length = 50)
    description = models.CharField(max_length = 100)

class CategorySerializer(serializers.ModelSerializer):
    class Meta:
        model = Category

class CategoryViewSet(viewsets.ModelViewSet):
    queryset = Category.objects.all()
    serializer_class = CategorySerializer

router = routers.DefaultRouter()
router.register(r'categories', CategoryViewSet)
urlpatterns = router.urls
```

Source 3.2: Example of a simple API with the Django REST framework

3.3.2 Eve

Eve [Iara] was created in 2013 and is also an open source framework in Python, and the current version is 0.6. It's built on top of the Flask microframework [Ron] that supports HTTP I/O and routing. Opposed to Django REST that supports four types of SQL databases, Eve only supports the non relational MongoDB databases. It's a framework more based in specification rather than writing code as the quickstart tutorial evidences [Iarb]. As an example, the same functionality of Source 3.2 can be implemented by specifying only the model into a Python variable as in Source 3.3.

```
DOMAIN = {
  'categories': {
    'schema': {
      'name': { 'type': 'string', 'maxlength': 50, 'required': True },
      'description': { 'type': 'string', 'maxlength': 100, 'required': True }}}}
```

Source 3.3: Example of a simple API with the Eve framework

3.3.3 LoopBack

LoopBack [Strb] was created in 2013 and is an open source Node.js[2] framework, which means that is written in JavaScript, and the current version is 2.18.0. It's built on top of the Express framework [Exp] that provides a thin layer of web application features. It considers relations of entities as REST resources besides the usual identified in section 3.3.

The framework tries to hide its inner workings by providing a command-line tool through which model entities are specified, as used in its quickstart tutorial [Stra]. In fact, this tool generates JSON files with the provided specification which may be edited. When the server application is started the model is interpreted and the necessary dispatch functions are dynamically generated, similar to Eve and contrary to Django REST. The example in Source 3.2 would be translated into the JSON file in Source 3.4.

```
1  {
2    "name": "Category",
3    "plural": "categories",
4    "base": "PersistedModel",
5    "idInjection": true,
6    "properties": {
7      "name": { "type": "string", "required": true },
8      "description": { "type": "string", "required": true }
9    },
10   "validations": [],
11   "relations": {},
12   "acls": [],
13   "methods": []
14 }
```

Source 3.4: Example of a simple API with the LoopBack framework

3.3.4 Sails

Sails [McN] was created in 2012 and is an open source Node.js framework, which means that is written in JavaScript, and the current version is 0.11. It's also built on top of the Express framework [Exp] providing an MVC development architecture.

The framework enables model-driven development first by providing entity scaffolding[3] using `sails generate api <entity_name>`, and secondly by expecting entities specification from the created Node modules. Just like the previous examples the model is only known by the framework in run-time by importing the modules. The same example as before (Source 3.2) would be specified as shown in Source 3.5.

[2] An asynchronous event driven framework designed to build scalable network applications [Foub].
[3] Generation of code templates.

```
module.exports = {
  attributes: {
    name: { type: "string", maxLength: 50, required: true },
    description: { type: "string", maxLength: 100, required: true }}};
```

Source 3.5: Example of a simple API with the Sails framework

3.3.5 Conclusion

As indicated in section 1.2 the identified frameworks are implemented in a dynamically type-checked language, either Python or JavaScript. Each of them implements model-based services with different approaches: (i) class specialization in the case of Django REST, (ii) variable initialization in the case of Eve and Sails, (iii) and command-line interaction in the case of LoopBack.

FEATURE	DJANGO REST	EVE	LOOPBACK	SAILS
HTTP I/O	Yes	Yes	Yes	Yes
Data storage	Relational	MongoDB	Relational & MongoDB	Relational & MongoDB
Routing	Yes	Yes	Yes	Yes
Serialization	Yes	Yes	No	Yes
Parsing	6 parsers	JSON & XML	JSON, XML & urlencoded	JSON & urlencoded
Rendering	10 renderers	JSON & XML	JSON & XML	JSON, XML & HTML
Filtering	Yes	Yes	Yes	No
Sorting	Yes	Yes	Yes	No
Pagination	Yes	Yes	Yes (allows)	No
Projection	No	Yes	Yes	No
Validation	Yes	Yes	Yes	Yes
HATEOAS	Semi-automatic	Yes	No	No
Authentication	Basic, Token, OAuth1.0a & OAuth2.0	Basic & Token	Token & OAuth2.0	Basic & Passport[4]
Conditional requests	No	Yes	No	No
Rate limiting	Yes	Yes	Yes	No
Caching	Yes	Yes	No	Yes
Content negotiation	Yes	No	No	Yes
HTTP method override	Yes	Yes	No	No

FEATURE	DJANGO REST	EVE	LOOPBACK	SAILS
CORS	Yes (third-party)	Yes (disabled)	Yes (enabled)	Yes
Testing	Yes	Yes (Flask)	Yes	Yes
Logging	Automatic	Automatic	Manual	Automatic
Documentation	Yes	No	Yes	No
Versioning	Yes (version 3.1)	Yes	Yes	No

Table 3.1: Comparison of model-driven frameworks by features

Aside from the problems that these present, they implement most of the features a REST framework may have, as specified in Table 3.1. The Django REST framework seams to be the most complete of all four frameworks, by supporting many parsers, renderers, and authentication schemes, and the only supporting more than two renderers.

3.4 STATE OF THE ART IN REST FRAMEWORKS IN SCALA

In the following sections (3.4.1 and 3.4.2) are presented two of the most used REST frameworks in Scala, and their features compared in section 3.4.3. Other frameworks were not included as they: are not suitable for implementing REST, such as the Xitrum [Xit] and the Simply Lift [Dav] web frameworks; or are not mature enough to be reused, such as the BlueEyes web framework [Blu] or the Unfiltered toolkit [Unf].

3.4.1 Spray

Spray [Typb] was created in 2011 and is an open source lightweight toolkit for building REST/ HTTP applications, and the current version is 1.3.2. It's built only with Scala, without any legacy Java libraries, and on top of Akka [Typa], another toolkit that handles concurrency and distribution using the concept of actors. In fact, the development of the module *spray-io* has influenced so much Akka evolution that both projects are now part of the Typesafe Reactive Platform [Typc]. Despite not being a framework in its definition, it is usually an alternative to other frameworks as implements many of their features efficiently. One important feature that doesn't provide is data storage.

3.4.2 Play

Play [Type] was created in 2011 and is an open source framework for building web applications with Java and Scala, and the current version is 2.3.8. It's also built on top of Akka and belongs to the Typesafe Reactive Platform. The framework restricts applications to follow a MVC architecture and is RESTful by default. Contrary to spray, it has data storage support through Anorm, a simple layer over SQL databases that still requires SQL queries to be written.

[4]Passport implements known authentication providers such as Facebook or Google, using OAuth1.0 or OAuth2.0.

3.4.3 Conclusion

As specified in Table 3.2, the frameworks provide almost the same features but diverge in data storage and rendering capabilities. The *play* framework is more developer friendly but *spray* is more suited to built on top of due to its lightweight nature.

FEATURE	SPRAY	PLAY
HTTP I/O	Yes	Yes
Data storage	No	Relational
Routing	Yes	Yes
Serialization	Yes	Yes
Parsing	10 parsers	11 parsers
Rendering	14 renderers	JSON and XML
Filtering	No	No
Sorting	No	No
Pagination	No	No
Projection	No	No
Validation	Yes	Yes
HATEOAS	No	No
Authentication	Basic	OAuth1.0
Conditional requests	Yes	No
Rate limiting	No	No
Caching	Yes	Yes
Content negotiation	Yes	Yes
HTTP method override	No	No
CORS	Yes	No
Testing	Yes	Yes
Logging	Yes	Yes
Documentation	No	No
Versioning	No	No

Table 3.2: Comparison of REST frameworks in Scala by features

Chapter 4

Metamorphic: A Model-Driven REST Framework in Scala

As specified in section 1.2, current model-driven REST frameworks have two major problems: are written with dynamically typed languages and interpret models in run-time. This chapter's framework pretends to be a proof that frameworks written with statically type-safe languages and with compile-time generation have several benefits comparing to existing ones. The Scala language and its macro system are the means to achieve such framework. From this point on this framework may be referred as Metamorphic - as generated applications may have several forms and the word is prefixed by meta that may designate metaprogramming.

This chapter starts by presenting the adopted development process in section 4.1 and an example of model susceptible of being implemented with the framework in section 4.2. Then the architecture of applications generated with Metamorphic is detailed and discussed in section 4.3. An internal DSL required to implement the framework is explained in section 4.4. Framework's implementation and underline architecture are detailed in section 4.5 and its verification presented in section 4.6. Some conclusions about the final result are presented in section 4.7.

4.1 Development Process

To systematize the development of Metamorphic it was used a bottom-up approach (section 3.1). At first, illegible Scala libraries and frameworks were explored through available online examples. Their easiness of use, including installation and implementation, were empirically considered with the means to select the best suited for implementing a proof of concept. Based on the same examples and libraries documentation it was built a base application with the same architecture of the ones the Metamorphic framework should provide.

Looking to the requirements of the framework, needs of the base application, and to the specification of the general purpose metametamodel MOF [Obj], it was designed a metametamodel suited to the requirements. A specific meta-model was also designed in the form of a internal

DSL, with a specification restricted by the macro system and inspired in some of the frameworks identified in section 3.3.

Using the base application as reference and following the *DSL*, the *macros* were then implemented. The development continued by iterating over these components: base application, metametamodel, metamodel, and macros. Framework's debugging and testing was supported through some unit tests, a debug option to output generated code, an example application, and a set of integration tests for the same application.

4.2 Representative Example

In order to exemplify and further illustrate the capabilities of the framework let's consider a simplified example of an online shop, represented as a class diagram in Figure 4.1.

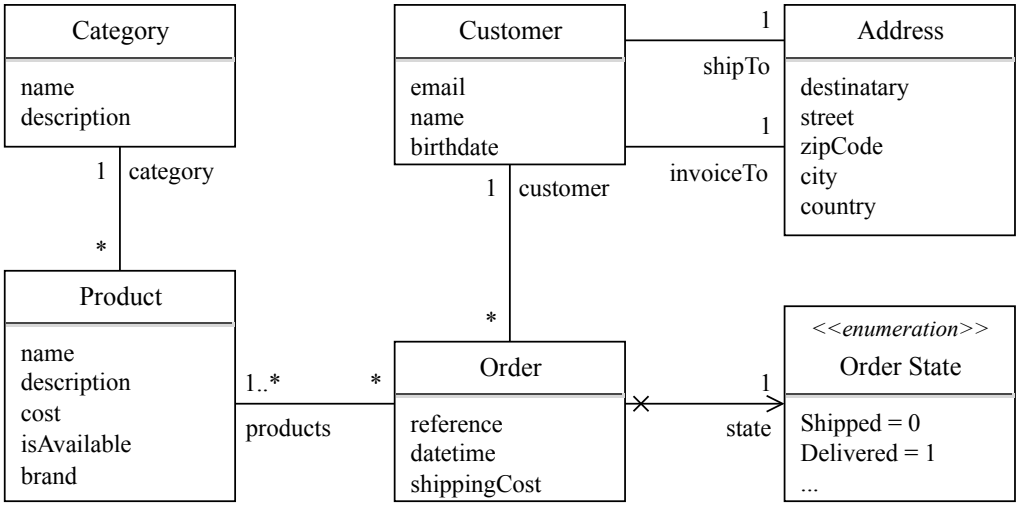


Figure 4.1: Class diagram of a online shop

Shortly, a shop has products divided by categories. About each product it's possible to know its name, description, cost, brand and whether it's available to be sold. These products can be associated with orders that have a reference, the date and time of the order, as well as a current state. Each order has a customer associated to it, which has an email, a name, a birth date, a shipment address, and an invoice address. Addresses have a destinatary, a street, a zip code, a city, and a country.

4.3 Application Architecture

Application of the described development process (section 4.1) culminated in two application programming styles: *synchronous* that is easier to use but may not handle properly large amounts of work; and *asynchronous* that increases development complexity but enables handling of large amounts of work. Both approaches are further explored and explained.

Despite the two dissonant programming styles, the architecture of final applications is only one (Figure 4.2). The architecture envisioned not to be model centered, allowing generation of generic web applications. This fact assures better software quality as components have to be less decoupled from its real use. Influenced by this decision, the architecture is composed by a mandatory layer, the *application logic*, and an optional layer, *data storage*, which may be used by the first layer.

Application logic is implemented by an *App* object that initiates services. These services may require access to data storage through repositories, which must have an entity associated to. The whole application has access to a set of developer settings.

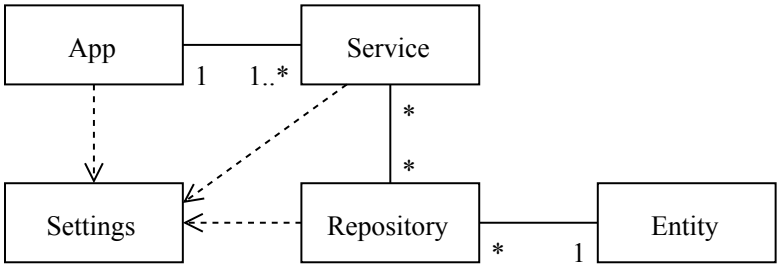


Figure 4.2: Architecture of a generated application

In the following sections, it's presented the interfaces for entities and repositories, a model for the application logic, and the use of developer's settings.

4.3.1 Entities

Entities are part of repositories interface. They are translated into case classes which may have at least a Option[T] *id* field, as illustrated in Source 4.1. Fields may be any of Scala's primitive types or the types LocalDate and DateTime from *org.joda.time*.

```
1 case class Category(
2   id: Option[Int],
3   name: String,
4   description: String)
```

Source 4.1: Example of an application entity

4.3.2 Repositories

Repositories implement the generic trait Repository[T], which defines a set of five possible operations (Source 4.2): getAll, get, create, replace and delete.

```
1 trait Repository[T] {
2   def getAll: List[T]
```

```
3    def get(id: Int): Option[T]
4    def create(instance: T): T
5    def replace(instance: T): Option[T]
6    def delete(id: Int): Boolean }
```

Source 4.2: Synchronous version of trait Repository

When using an asynchronous programming style the return type of operations are of type Future[T] (Source 4.3) where T is the return type of the synchronous version.

```
1  trait Repository[T] {
2    def getAll: Future[List[T]]
3    def get(id: Int): Future[Option[T]]
4    def create(instance: T): Future[T]
5    def replace(instance: T): Future[Option[T]]
6    def delete(id: Int): Future[Boolean] }
```

Source 4.3: Asynchronous version of trait Repository

4.3.3 Application Logic

Being application logic the top layer, it's important not to limit its usage with a fixed interface. An alternative to this solution it's modeling the logic, as shown in Figure 4.3, enabling greater flexibility. The model allows the specification of services, which may have dependencies and a set of operations. Each operation implements an HTTP method for a path, expects the request body to be serializable for a specified class, and contains a body. It's at the operations level that one of the possible programming styles is applied, by using the *isAsync* flag.

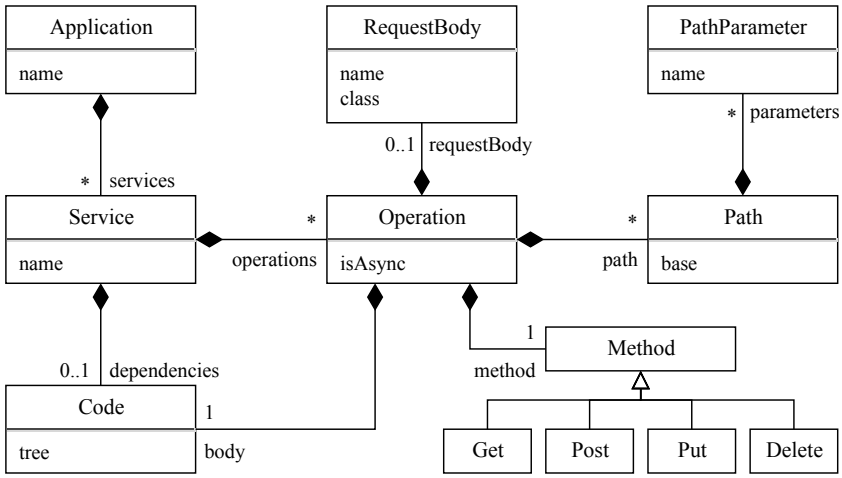


Figure 4.3: Application logic model

4.3.3.1 Entity Operations

The operations identified in section 1.3.2 are implemented using entities plural as the base path, and are identified as follows:

- **Create.** It is mapped to a POST method that after extracting the request body uses a repository to create the entity. The created entity, with the appropriate id, is returned in the response body together with a 201 (Created) status code.
- **GetAll.** It is mapped to a GET method that returns all instances of an entity together with a 200 (Ok) status code.
- **Get.** It is mapped to a GET method that requires a path parameter with the entity id. The entity is returned alongside with a 200 (Ok) status code.
- **Replace.** It is mapped to a PUT method that requires a path parameter with the entity id. The request body is extracted and used together with the id to change the entity in the repository. The new representation is returned with a 200 (Ok) status code.
- **Delete.** It is mapped to a DELETE method that requires a path parameter with the entity id. After deleting the entity from the repository a 204 (No Content) status code is returned.

Create and *Replace* operations return a 400 (Bad Request) status code if the request body isn't properly built. *Get*, *Replace*, and *Delete* operations return a 404 (Not Found) status code if the given entity id doesn't exist. Requests and responses are formatted with JSON.

4.3.4 Settings

The scope of the framework expects the possibility of specifying server configurations (section 1.3.2). The use of the *Config* library [Typd] is common in Scala, and enables the use of one file for setting configurations of all dependencies of a project. This means that besides Metamorphic's configurations, developers can still configure underlying libraries.

As illustrated in Source 4.4, configurations are defined inside the *metamorphic* scope and shall be either host ("localhost" as default), port (8080 as default) or databases. Scopes inside *databases* may specify a name, an user, a password, an host, a port, a number of threads (numThreads) or a maximum queue size (queueSize).

```
1 metamorphic {
2   host = "111.111.111.111"
3   port = 9000
4   databases { default { name = "file.db" } } }
```

Source 4.4: Example of configuration file with a SQLite database

4.4 INTERNAL DSL

Scala macros enables generation of classes, traits and objects either through type providers or macro annotations (section 2.3.3.2). The first discourages reuse of these elements in the scope calling the macros, while the second despite its limitations allows reuse when generated inside an object. This implies that shall exist a language to these annotations that allows a mapping to the final application. In this section a description of such language is made.

So applications shall be specified using the @app annotation over an object, as shown in Source 4.5. This object may have a list of entity definitions, a list of default operations for each entity, and a list of service definitions, according with the specifications given in sections 4.4.1, 4.4.2, and 4.4.3, respectively.

```
1 import metamorphic.dsl._
2 @app object ShopApplication {
3   ... // entities
4   ... // services
5   ... // operations list
6 }
```

Source 4.5: Usage of the @app annotation

4.4.1 Entities

Before presenting how to specify an entity, one shall know which metametamodel is trying to be represented by the DSL. According with the framework's scope (section 1.3.2) the metametamodel in Figure 4.4 was designed. Entities are composed by properties that have a type, either a simple type or a relation type. The relation type contains two ends which may be of type object or list. Each relation end has an entity associated with it, that combined form the entities of a relation.

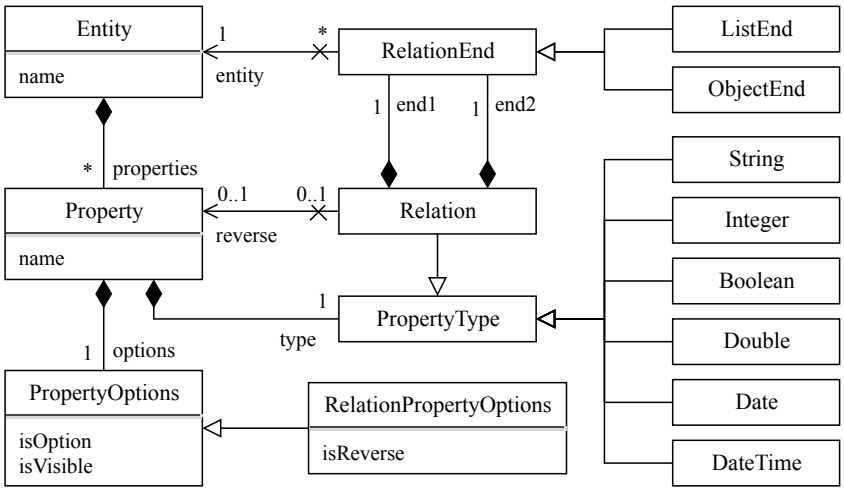


Figure 4.4: Framework's metametamodel

Using the DSL entities can be defined using the @entity annotation over a class, as shown in Source 4.6. The macro annotation expansion adds a companion object with replicated content, which helps to identify some types of errors as the compiler will typecheck the result (section 2.3.3.2). Properties are defined declaring functions that return a Field value. Fields are case classes that accept a variable number of values and may be of type: IntegerField, DoubleField, StringField, BooleanField, DateField, DateTimeField, ObjectField, ListField, ReverseField.

```
1 @entity class Order {
2   def reference = StringField()
3   def datetime = DateTimeField()
4   def shippingCost = DoubleField(Option)
5   def state = IntegerField()
6   def products = ListField(Product)
7   def customer = ObjectField(Customer)
8 }
```

Source 4.6: Example of an entity definition

All fields accept an Option argument, meaning that the property is not required. The first argument of a ListField or an ObjectField is the companion object of an entity definition. These two types of fields are by default mapped to many-to-many and one-to-many relations, respectively. The use of R.Object as argument allows many-to-one and one-to-one relations, respectively.

4.4.2 Operations List

By default, each entity has the operations identified in section 4.3.3.1 automatically implemented. Declaring the *operations* variable, as shown in Source 4.7 in the *@app* scope changes the set of default operations to be implemented. Operations can be identified using the following objects: Create, GetAll, Get, Replace, and Delete.

```
1 val operations = List(Create, Get)
```

Source 4.7: Example of a list of default operations

4.4.3 Entity Services

Customizations to default implemented operations are done by extending the generic trait EntityService[T], as shown in Sources 4.8 and 4.9. Its interface is similar to a repository with the difference that each function has a return type Response. A Response takes the response content as first argument and an HTTP status code as second argument. Besides a service has available a *repository* variable of type Repository[T] for implementing required customizations. An operations list (section 4.4.2) can be declared inside the service to change the set of default operations for a particular entity.

```
class ProductService extends EntityService[Product] {
  def getAll = {
    val result = repository.getAll.filter(product => product.isAvailable)
    Response(result, Ok) } }
```

Source 4.8: Example of a synchronous EntityService implementation

When using an asynchronous programming style the return type of operations are of type Future[Response] (Source 4.9). As mentioned in section 4.3 and proved by these examples, the two styles have different programming complexities. In this case the difference is small as there is only one callback.

```
class ProductService extends EntityService[Product] {
  def getAll = {
    repository.getAll.map(products => {
      val result = products.filter(product => product.isAvailable)
      Response(result, Ok) }) } }
```

Source 4.9: Example of an asynchronous EntityService implementation

4.5 IMPLEMENTATION

Having defined applications' architecture in section 4.3 and a domain specific language capable of supporting such definition in section 4.4, were meet the necessary requirements to implement the framework. The classes and traits required by the DSL were created in package *metamorphic.dsl* (Figure 4.5), including the *@app* annotation. The annotation implementation depends of package *matcher* for translating the code tree into a metametamodel and an application logic model. It also depends of package *generator* for mapping those models into the final application tree.

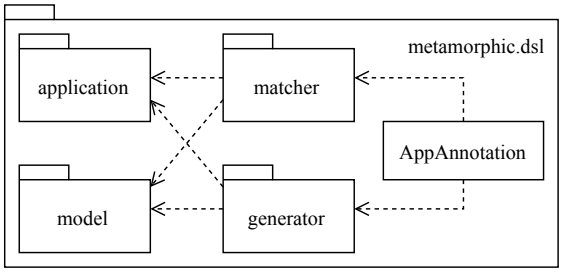


Figure 4.5: Diagram of packages for the Metamorphic framework

When designing a framework, one goal besides having a correct solution is to have a flexible and loosely decoupled architecture that allows long-term maintainability. In this case, in which the framework is generative that goal goes behind the architecture of the application, as one may want to generate applications with different implementations. With that in mind the *generator* package

doesn't provide any concrete generation of applications, building them instead using dependency injection. The framework requires for a project to provide two dependencies/generators: a repository generator and a service generator. These can be specified using a configuration file as already introduced in section 4.3.4, and are instantiated using runtime reflection (section 2.3.1).

For testing this proof of concept were implemented two repository generators based in the Slick library [Inc]. Slick is a functional-relational mapper in Scala that, contrary to an ORM approach, doesn't hide the relational nature enabling faster access to data. One of the generators targets synchronous applications and uses the version 2.1 of the library, while the other targets asynchronous applications and uses the version 3.0. The Figure 4.6 illustrates how dependency injection is performed for a RepositoryGenerator that uses Slick, which also uses dependency injection to implement different database systems. It was also implemented a service generator that uses components from the Spray toolkit.

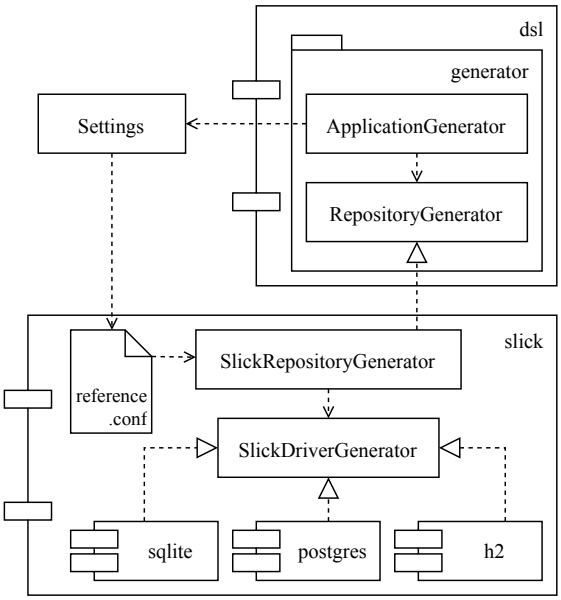


Figure 4.6: Dependency injection of a RepositoryGenerator that uses Slick

A better understanding of the operations executed by the framework are illustrated in Figure 4.7. First the scala compiler expands the inner annotations *@entity* that produces the companions to be used by relation fields, after which they are typechecked. Then on expansion of *@app* the matchers extract information from trees according with the internal DSL to instantiate the two models. Based in these, the application generator creates the case classes for the entities, the repositories and services.

The proof of concept implementation presented some relevant challenges:

- The configuration files weren't available during macro expansion. This happened because the scala compiler only copies project resources to the classpath after compiling the class files. The solution was to change the base project to depend on a mock project that only contains the configuration file.

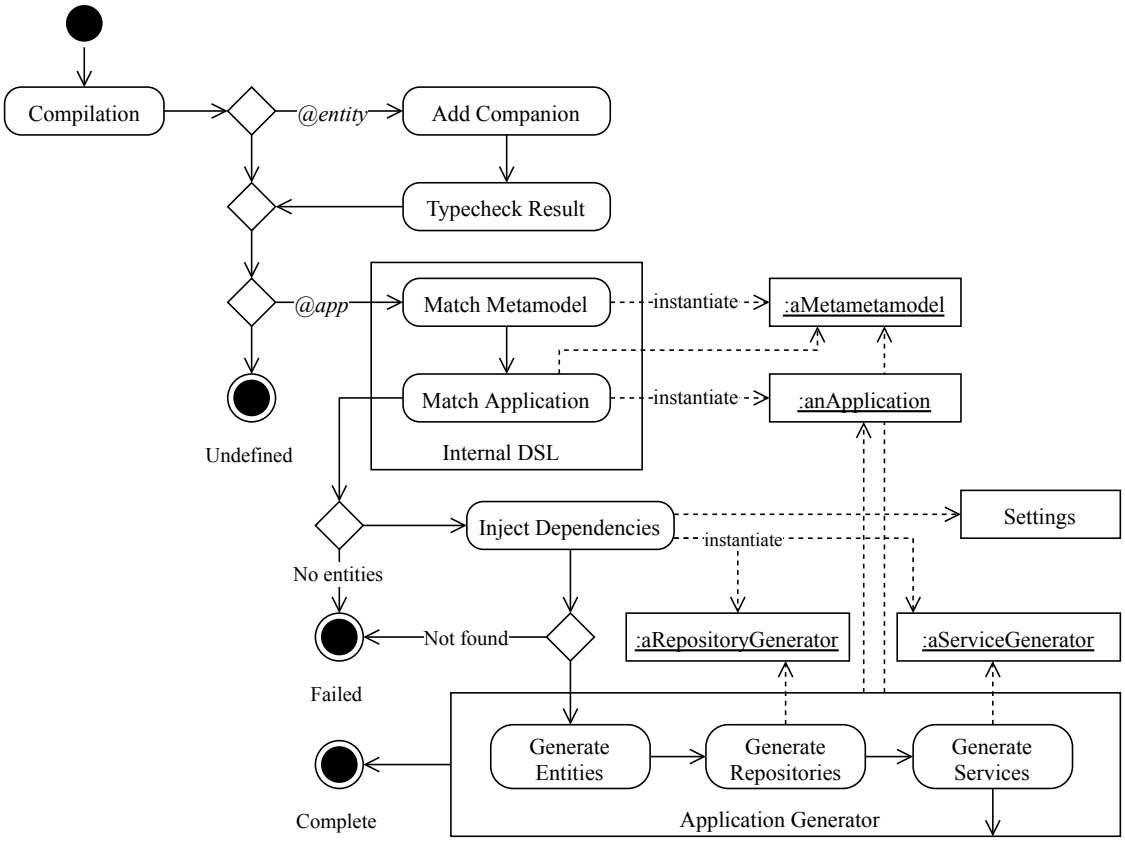


Figure 4.7: Activity diagram of a Metamorphic's application compilation

- Construction of right recursive trees. Despite the great use of quasiquotes, they don't permit building right recursive trees. The solution was to use the conservative class based construction of trees.

- Change to non-blocking database access. After the first benchmarks, further detailed in Chapter 5, the framework didn't perform as expected. After understanding that the database driver used blocking access, the recently launched asynchronous version of Slick was selected to solve the problem.

4.6 VERIFICATION

Verification of the framework was achieved through unit testing and integration testing. Unit testing was done via the ScalaTest library [Art] for verifying the result of the DSL matchers. Doing unit testing to Scala macros helper functions isn't trivial as these depend of a compiler context that cannot be mocked. So the possible solution was to create wrapper classes that define def macros which invoke the classes to be tested (e.g. ModelProvider in Source 4.10). In order for them to work was also necessary to implement lifting, so that return types of the matchers could be translated into expressions that the compiler can understand.

```
test("Simple model") {
  val model = ModelProvider.model {
    @entity class Category {
      def name = StringField()
      def description = StringField()
    }
  }
  assert(model.entities.head.name.equals("Category"))
  assert(model.entities.head.properties.length == 2)
}
```

Source 4.10: Example of a unit test to the model matcher

The generators were tested using integration tests as the complexity of unit testing their output would lead to a greater cost-benefit value. The integration tests were implemented using the Node.js module Frisby [Van] for the application described in Figure 4.1. The tests asserted response bodies and status codes. With these and the use of the boolean setting *macroDebug*, which outputs generated code, correctness of the generators were guaranteed.

4.7 Conclusion

The systematic approach to the development of the framework gives some guarantees not only of its architecture but also of its implementation. The two models behind the generation of an application assures that the framework may have flexibility enough to further evolve. The dependency injection architecture revealed to be a good option as it allowed a quick implementation of a new repository generator. In conclusion, the Metamorphic framework is a viable proof of concept to this dissertation research problem (section 1.3).

Chapter 5

Benchmarks

This chapter aims to assert, through benchmarks, the assumption identified in section 1.3.4, that a framework with the characteristics of Metamorphic's achieve implementations capable of responding to requests faster than the current model-driven frameworks (section 3.3). These benchmarks are further detailed in section 5.1 were the experiment is conceptually defined, including the frameworks to test and the process to follow. Its realization is explained in section 5.2 and the results presented and discussed in section 5.3. A brief summary of the chapter and some conclusions are taken in section 5.4.

5.1 Research Design

Proving a framework to be faster than others is not as trivial as it might seem, once there is several criteria that can be used to compare implementations. Some of these might be: the mean, the standard deviation, the maximum value, the minimum value, the median, the 25^{th} percentile or the 75^{th} percentile.

It should also be taken into account that results shall not be generic, i.e. comparison must be executed between services with the same characteristics. So, the categorization of compared services followed two properties: the type of entities and the type of operations. Entities were: (i) *simple entities* which don't have navigable relations with other entities; (ii) *entities with objects* which have a navigable relation with multiplicity one; (iii) or *entities with lists* which have a navigable relation with an infinite upper bound. Operations were the ones already specified in section 4.3.3.1: Create, GetAll, Get, Replace, and Delete.

5.1.1 Frameworks

In this experiment the same scenario was compared using seven different implementations: (i) one with the synchronous version of Metamorphic, here denominated Metamorphic; (ii) one with the asynchronous version of Metamorphic, here denominated Metamorphic Async; (iii) one with LoopBack [Strb]; (iv) one with Sails [McN]; (v) one with Django REST [Chr] in Python 2.7,

here denominated Django REST; (vi) one with Django REST in Python 3.4, here denominated Django REST 3.4; (vii) and one with Eve [Iara].

All of these used a PostgreSQL 9.3.8 database, except the Eve implementation that doesn't have an easy support to such database system and so used a MongoDB database. To guarantee that the performance of these implementations is maximized they were installed in production environments.

5.1.2 Process

Benchmarks were done through a Node.js application which used the bench-rest library [Jef] for collecting response times. Being N the sample size for each comparison scenario and C the number of entities to have in the database when testing the GetAll operation, the application executes the following set of tasks for each type of entity:

1. **Create** of N entities
2. **Get** of each created entity
3. **Replace** of each created entity
4. **Delete** of each created entity
5. Create of C entities
6. **GetAll** N times
7. Delete of the last created C entities
8. Export of raw data and statistics

This sequence ensures that after terminating the database remains as it started. All batches of operations were performed with a maximum of 10 concurrent requests.

Before executing the tests all implicated databases were verified to be fully empty. Then for each implementation, using $C = 100$, the application was manually executed a first time with $N = 1000$, to discard any possible setup effects, and finally with $N = 5000$ for collecting statistics.

5.2 Experiment Description

An extension to the example in section 4.2 was used, considering the Address entity as a representative of *simple entities*, the Product entity as a representative of *entities with objects*, and the Shop entity as a representative of *entities with lists* as shown in Figure 5.1.

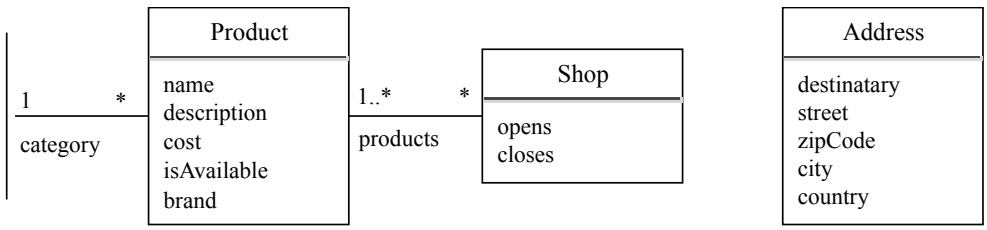


Figure 5.1: Representative entities used in the benchmarks

The experiment was executed in a portable computer Lenovo Thinkpad T430 with the following specifications: Ubuntu 14.04 LTS 32-bit; Intel Core i5-3210M @ 2.5GHzx4; 4GB Sodimm DDR3 Memory (1600 MHz); and 500GB 7200 RPM 32 MB Cache SATA Hard Drive.

In order to eliminate network latency as a validation threat the tests were locally executed. For preventing great impacts of other programs in the results, internet connection was disabled and all non-essential programs were closed having only two terminals in foreground with: the server application and the benchmark application.

5.3 Data Analysis

In the following sections the experiment results by operation are graphically presented and analyzed with help of a rank sum. The charts use box plots [Tuk77] with outliers to present the samples indicating the 25^{th} percentile (Q_1), median, and 75^{th} percentile (Q_3) values. Considering the interquartile range, $IQR = Q_3 - Q_1$, the bottom whisker is equal to $Q_1 - 1.5 \times IQR$ and the top whisker equal to $Q_3 + 1.5 \times IQR$. The outliers are represented with dots.

The results of the Eve framework are presented only for reference and won't be included in comparisons. The full statistics can be found in Appendix A.

5.3.1 Create

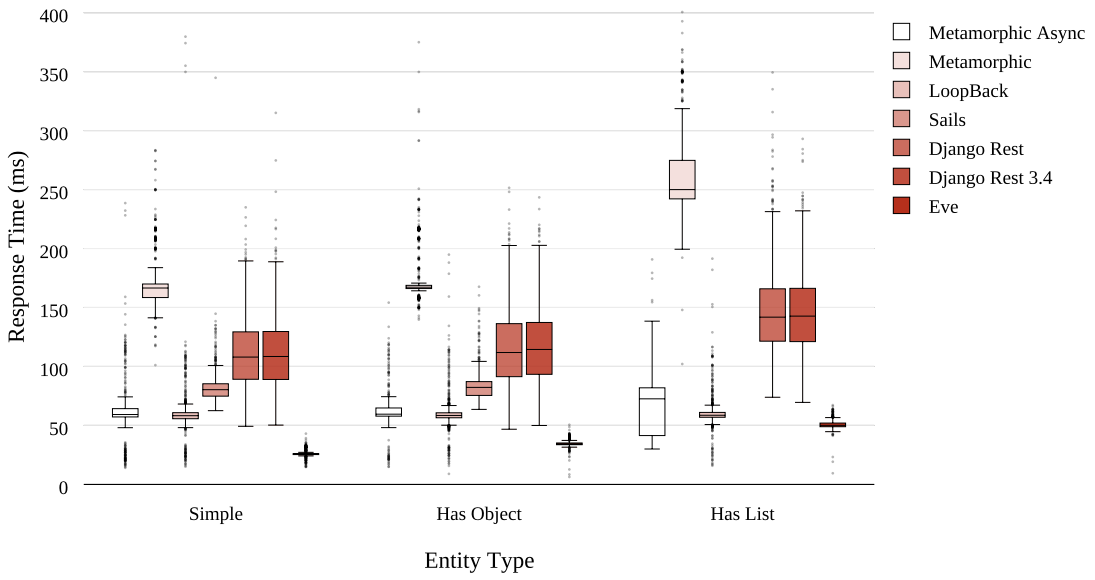


Figure 5.2: Frameworks' response time to Create operation by entity type

This operations on **simple entities** performed better with Metamorphic Async ($\bar{x} = 62.78$, $\sigma = 27.26$) and LoopBack ($\bar{x} = 59.60$, $\sigma = 18.85$) with the second being slightly faster. Metamorphic ($\bar{x} = 173.12$, $\sigma = 28.13$) was the one with worst results.

This operations on **entities with objects** also performed better with Metamorphic Async ($\bar{x} = 61.55$, $\sigma = 13.22$) and LoopBack ($\bar{x} = 59.83$, $\sigma = 14.82$) with the second being slightly faster again. Metamorphic ($\bar{x} = 175.21$, $\sigma = 23.62$) had the worst results again but with a smaller interquartile range.

This operations on **entities with lists** performed better with LoopBack ($\bar{x} = 59.47$, $\sigma = 12.62$). Despite having a similar mean value Metamorphic Async ($\bar{x} = 63.72$, $\sigma = 24.21$) has a wider interquartile range, which may mean that there is room for improvements. Metamorphic ($\bar{x} = 262.90$, $\sigma = 39.06$) had the second worst results as Sails ($\bar{x} = 1487.77$, $\sigma = 830.53$) underperformed a lot.

In conclusion, LoopBack achieves better performances for Create operations followed by Metamorphic Async, as shown by the rank sum in Table 5.1. Also, these operations on entities with lists are not recommended with the Sails framework, which had response times that couldn't even be shown in Figure 5.2.

Framework	Simple	Has Object	Has List	$\sum$
LoopBack	1	1	1	3
Metamorphic Async	2	2	2	6
Sails	3	3	6	12
Django REST	4.5	4	3.5	12
Django REST 3.4	4.5	5	3.5	13
Metamorphic	6	6	5	17
Eve	1*	1*	1*	3*

Table 5.1: Frameworks' rank of Create operation by entity type and their rank sum

5.3.2 GetAll

This operations on **simple entities** performed better with Metamorphic Async ($\bar{x} = 29.23$, $\sigma = 8.59$). Despite being worst than the most, Metamorphic ($\bar{x} = 96.40$, $\sigma = 7.28$) outperformed both Django REST implementations.

This operations on **entities with objects** also performed better with Metamorphic Async ($\bar{x} = 27.04$, $\sigma = 9.11$). Metamorphic ($\bar{x} = 94.73$, $\sigma = 9.31$) outperformed again both Django REST implementations and also Sails.

This operations on **entities with lists** performed better with LoopBack ($\bar{x} = 78.95$, $\sigma = 9.13$) relegating Metamorphic Async ($\bar{x} = 94.68$, $\sigma = 11.61$) to second place. Metamorphic ($\bar{x} = 254.03$, $\sigma = 19.16$) comes third together with Sails ($\bar{x} = 253.97$, $\sigma = 38.64$). Both Django REST implementations underperformed a lot.

In conclusion, Metamorphic Async achieves better performances most of the times for GetAll operations followed by LoopBack, as shown by the rank sum in Table 5.2. Also, these operations on entities with lists are not recommended with both versions of Django REST, which had response times that could barely be shown in Figure 5.3.

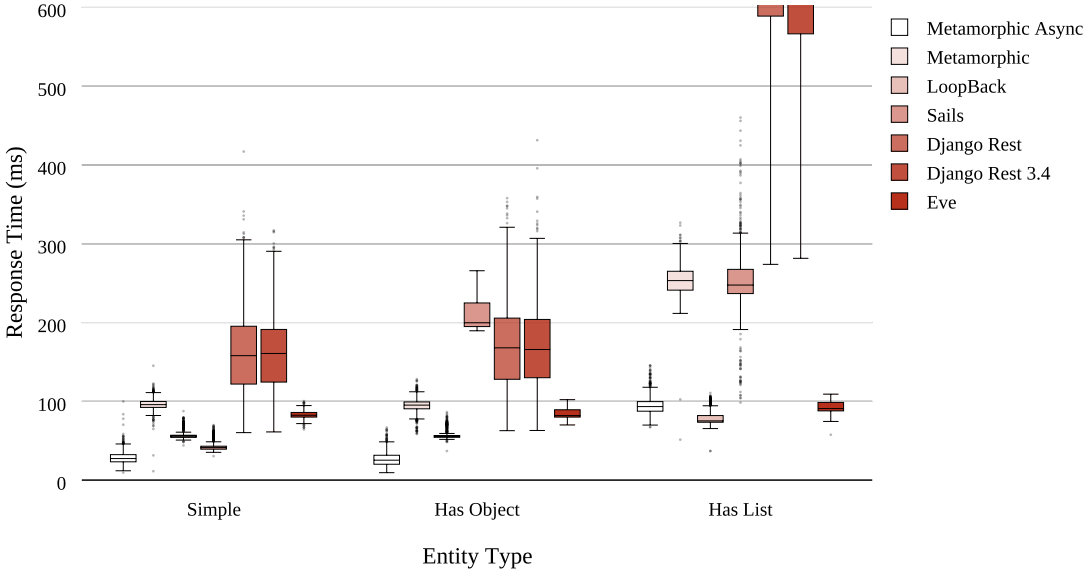


Figure 5.3: Frameworks' response time to GetAll operation by entity type

FRAMEWORK	SIMPLE	HAS OBJECT	HAS LIST	∑
Metamorphic Async	1	1	2	4
LoopBack	3	2	1	6
Metamorphic	4	3	3	10
Sails	2	6	4	12
Django REST 3.4	5	4	5	14
Django REST	6	5	6	17
Eve	4*	3*	2*	9*

Table 5.2: Frameworks' rank of GetAll operation by entity type and their rank sum

5.3.3 Get

This operations on **simple entities** performed better with LoopBack ($\bar{x} = 14.13$, $\sigma = 3.64$), while Metamorphic Async ($\bar{x} = 19.37$, $\sigma = 8.16$) and Sails ($\bar{x} = 19.93$, $\sigma = 4.53$) had the second best results, being the second being slightly faster. Metamorphic ($\bar{x} = 65.34$, $\sigma = 7.16$) underperformed all implementations except Django REST.

This operations on **entities with objects** performed better with LoopBack ($\bar{x} = 13.67$, $\sigma = 3.43$), while Metamorphic Async ($\bar{x} = 19.27$, $\sigma = 8.28$) had the second best results. Metamorphic ($\bar{x} = 66.18$, $\sigma = 8.44$) underperformed again all implementations except Django REST.

This operations on **entities with lists** performed better with LoopBack ($\bar{x} = 14.05$, $\sigma = 3.57$) again, while Metamorphic Async ($\bar{x} = 20.40$, $\sigma = 8.50$) also had the second best results. Metamorphic ($\bar{x} = 70.86$, $\sigma = 6.92$) had similar results outperformed only Django REST again.

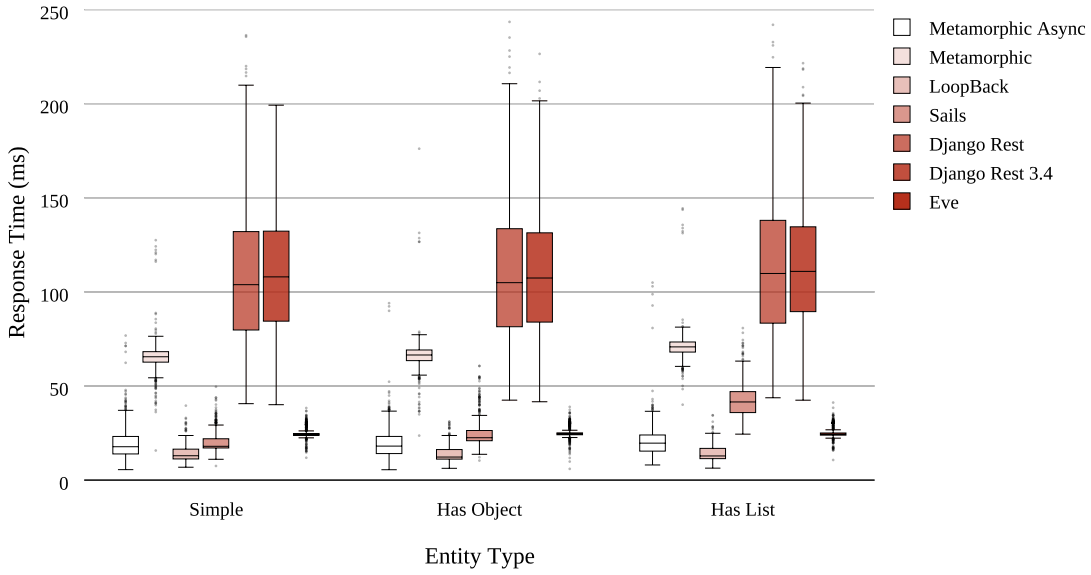


Figure 5.4: Frameworks' response time to Get operation by entity type

In conclusion, LoopBack achieves better performances for Get operations followed by Metamorphic Async and Sails, as shown by the rank sum in Table 5.3. Also, these operations are not recommended with both versions of Django REST, which had response times clearly worst than all others as shown in Figure 5.4.

FRAMEWORK	SIMPLE	HAS OBJECT	HAS LIST	$\sum$
LoopBack	1	1	1	3
Metamorphic Async	3	2	2	7
Sails	2	3	3	8
Metamorphic	4	4	4	12
Django REST 3.4	6	5	5	16
Django REST	5	6	6	17
Eve	4*	3*	3*	10*

Table 5.3: Frameworks' rank of Get operation by entity type and their rank sum

5.3.4 Replace

This operations on **simple entities** performed better with Metamorphic Async ($\bar{x} = 61.20$, $\sigma = 13.79$) and LoopBack ($\bar{x} = 61.44$, $\sigma = 14.53$) with the first being slightly faster. Metamorphic ($\bar{x} = 173.78$, $\sigma = 26.20$) was the one with worst results.

This operations on **entities with objects** also performed better with Metamorphic Async ($\bar{x} = 61.88$, $\sigma = 16.04$) and LoopBack ($\bar{x} = 62.12$, $\sigma = 16.87$) with the first being slightly faster again. Metamorphic ($\bar{x} = 175.42$, $\sigma = 26.42$) had the worst results again but with a wider interquartile range.

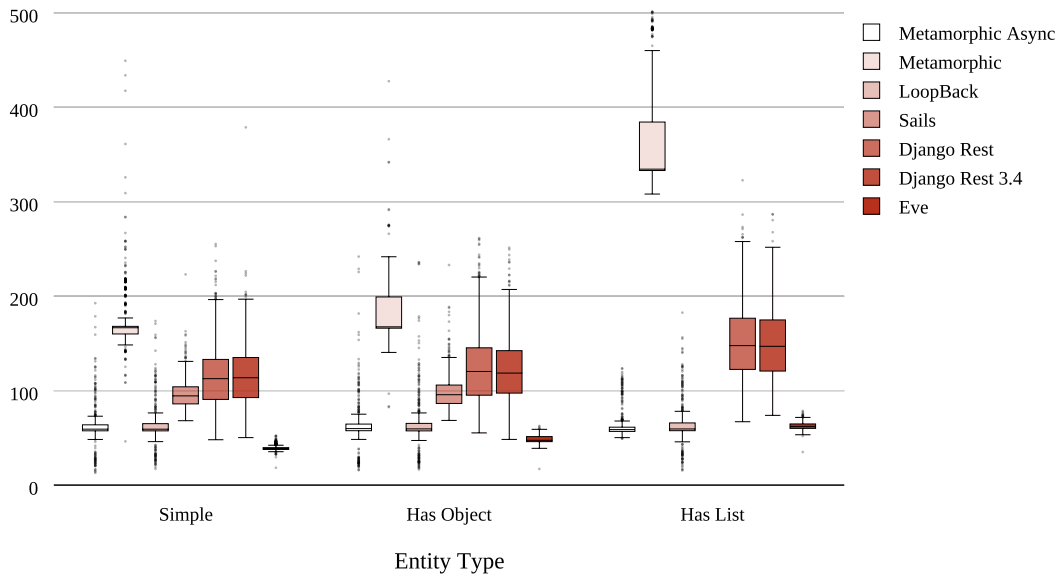


Figure 5.5: Frameworks' response time to Replace operation by entity type

This operations on **entities with lists** also performed better with Metamorphic Async ($\bar{x} = 61.60$, $\sigma = 11.42$) and LoopBack ($\bar{x} = 62.08$, $\sigma = 16.35$) with the first being slightly faster again. Metamorphic ($\bar{x} = 362.30$, $\sigma = 47.06$) had the second worst results as Sails ($\bar{x} = 2975.13$, $\sigma = 337.66$) underperformed a lot.

In conclusion, Metamorphic Async achieves better performances for Replace operations followed by LoopBack, as shown by the rank sum in Table 5.4. Also, these operations on entities with lists are not recommended with the Sails framework, which had response times that couldn't even be shown in Figure 5.5.

Framework	Simple	Has Object	Has List	$\sum$
Metamorphic Async	1	1	1	3
LoopBack	2	2	2	6
Sails	3	3	6	12
Django REST 3.4	5	4	3	12
Django REST	4	5	4	13
Metamorphic	6	6	5	17
Eve	1*	1*	2*	4*

Table 5.4: Frameworks' rank of Replace operation by entity type and their rank sum

5.3.5 Delete

This operations on **simple entities** performed better with LoopBack ($\bar{x} = 22.63$, $\sigma = 8.41$), while Metamorphic Async ($\bar{x} = 30.88$, $\sigma = 18.77$) had the second best results. Metamorphic ($\bar{x} =$

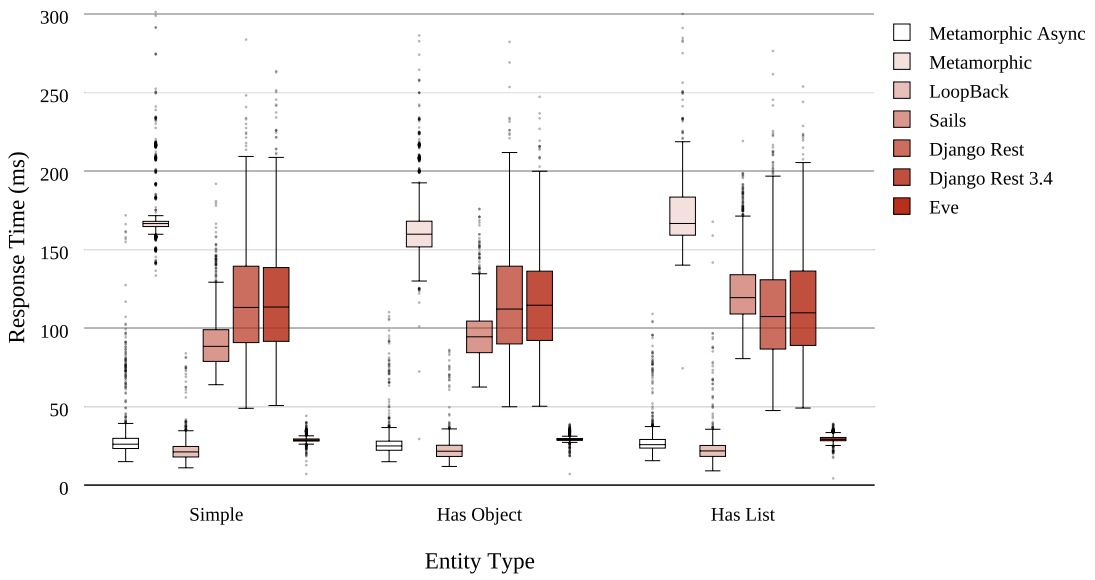


Figure 5.6: Frameworks' response time to Delete operation by entity type

173.66, $\sigma = 23.71$) was the one with worst results.

This operations on **entities with objects** also performed better with LoopBack ($\bar{x} = 23.08$, $\sigma = 8.69$), while Metamorphic Async ($\bar{x} = 27.39$, $\sigma = 11.86$) had the second best results again. Metamorphic ($\bar{x} = 166.31$, $\sigma = 25.16$) was again the one with worst results.

This operations on **entities with lists** also performed better with LoopBack ($\bar{x} = 23.37$, $\sigma = 11.17$), while Metamorphic Async ($\bar{x} = 28.07$, $\sigma = 11.18$) had the second best results again. Metamorphic ($\bar{x} = 174.85$, $\sigma = 26.38$) was again the one with worst results.

In conclusion, LoopBack achieves better performances for Delete operations followed by Metamorphic Async, as shown by the rank sum in Table 5.5.

FRAMEWORK	SIMPLE	HAS OBJECT	HAS LIST	$\sum$
LoopBack	1	1	1	3
Metamorphic Async	2	2	2	6
Sails	3	3	5	11
Django REST 3.4	4	4	4	12
Django REST	5	5	3	13
Metamorphic	6	6	6	18
Eve	3*	3*	3*	9*

Table 5.5: Frameworks' rank of Delete operation by entity type and their rank sum

5.4 Conclusion

This experiment attempted to assert that the Metamorphic framework has *better responses times* (section 1.3.4) when compared with current model-driven REST frameworks. To validate such assertion implemented services were categorized and individually tested following a set rules guarantee validity of the process.

From the results in section 5.3 it was built Table 5.6 that has the rank sum for each pair (Framework, Operation) and the rank sum for each framework. The final sums are indicators of the global performance of each framework. So it's possible to conclude that, in spite of not being the best performant framework, the asynchronous version of Metamorphic already achieves performances better than most model-driven frameworks. In fact, without almost no effort to optimize framework's implementation, its results are near to the most performant framework, LoopBack, and it's the best solution to implement GetAll and Replace operations.

Should be noted that the pseudo-rank for the framework Eve is merely indicative as the other ranks are computed discarding Eve.

Framework	Create	GetAll	Get	Replace	Delete	$\sum$
LoopBack	3	6	3	6	3	21
Metamorphic Async	6	4	7	3	6	26
Sails	12	12	8	12	11	55
Django REST 3.4	13	14	16	12	12	67
Django REST	12	17	17	13	13	72
Metamorphic	17	10	12	17	18	74
Eve	3*	9*	10*	4*	9*	35*

Table 5.6: Framework's rank sum by operation type and their rank sum

Chapter 6

Academic Quasi-Experiment

As planned in section 1.3.4, it was performed a synthetic envirnoment experiment in the form of an academic quasi-experiment. Such experiment is further detailed in this chapter, starting by section 6.1 that includes the treatments, the process, and questionnaires that were used. In section 6.2 the tasks of the experiment are described and explained.

Then the results of the questionnaires are analyzed in section 6.3 while the some objective measurements are explored in section 6.4. In section 6.5 some possible validation threats are identified and to finish conclusion of the experiment are summarized in section 6.6.

6.1 Research Design

For this experiment, 8 MSc students in their 5^{th} year of the Master in Informatics and Computing Engineering, lectured at the University of Porto, Faculty of Engineering, were asked to participate. With the given time and human resources, the experiment tested only one of the current model-driven frameworks (section 3.3) against the synchronous version of Metamorphic. Taking advantage of having subjects available for the experiment, they were asked to implement the same API using both approaches.

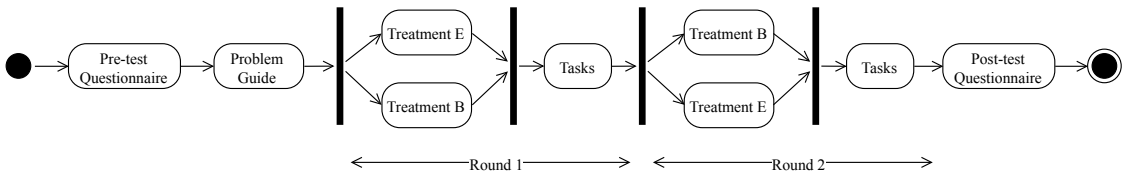


Figure 6.1: Experiment design. All subjects started by answering a questionnaire and reading a problem guide (Appendix B) common to the treatments. These were then split in two groups with different treatments and had to perform the same set of tasks (Round 1). Then they were object of the remaining treatment performing the same set of tasks as before (Round 2). The test finished by answering to another questionnaire.

6.1.1 Treatments

The experiment used two different treatments:

- **Baseline treatment.** The baseline is based in the use of Django REST, the most popular model-driven framework, in its Python 2.7 version. A guide (Appendix C) on how to use the framework was provided, which included a brief explanation of the Python syntax, in order to standardize knowledge. A default ready-to-use Django REST project configured to use a SQLite database was created.
- **Experimental treatment.** Similar to the baseline a guide (Appendix D) was provided, which also included an explanation of the Scala syntax, with the same goal. A default ready-to-use Metamorphic synchronous project configured to use a SQLite database was created.

The group that was firstly subjected to the baseline treatment shall be considered group 1, while the other shall be considered group 2. The guides were provided in an HTML form as most frameworks documentation nowadays are provided this way. This may reduce the risk of poor adaptability to the documentation.

6.1.2 Pre-test Evaluation

In the experiment it's considered that students may be valid representatives of the population of API developers. To ensure some validity to this statement the subjects provided their current average grade in the Master. This is an indicator of their base skills as programmers, and as shown in Table 6.1 or G.1 subjects can be considered well qualified. Comparing the two groups using an independent-samples t-test, with unequal variances, it was shown that there was **no significant** difference in the scores of group 1 ($\bar{x} = 15.75$, $\sigma = 2.217$) and group 2 ($\bar{x} = 16.75$, $\sigma = 1.500$); $\rho = 0.487$, within a significance level of 95%.

GROUP	N	MEAN	STD. DEVIATION	T-TEST
1	4	15.75	2.217	0.487
2	4	16.75	1.500	

Table 6.1: Subjects grades statistics by group

So despite the difference between the two groups, they can still be called admissible. A higher number of subjects would probably reduce these differences, as it would also improve the quality of the results.

6.1.3 Process

The experiment was built to assess several distinct claims, mainly related with the development process and the usability of the DSL. Each subject executed the test in a excluded area of a low

noise laboratory, with a single portable computer with internet access mimicking a real programming situation.

Due to lack of macro integration with IDEs the subjects could only use a text editor of their choice. Application running and testing had to be done using the terminal. The subjects were told before starting the test that they could clarify any doubts and that who was being were the frameworks and not them as programmers.

Data Collection

Before starting each test a *screencast* of the computer was initiated with the intent to correctly measure time and development metrics in a non-intrusive way. The number of requests for help was manually recorded.

6.1.4 Questionnaires

The questionnaires were designed using a Likert scale [Lik32] - see Appendices E and F. It were used 30 five-point Likert items with the following format: (1) strongly disagree; (2) somewhat disagree; (3) neither agree nor disagree; (4) somewhat agree; (5) strongly agree. These were grouped as follows:

1. **Background.** In spite of already having the average course grade as a term of comparison between the background of the groups, it is important to ensure that there is no difference regarding the skills required to complete the requested tasks.

2. **External Factors.** This information may help remove some validation threats regarding the experimental conditions, that would not probably exist in a valid situation.

3. **Problem Guide.** As the previous one, these may help remove validation threats regarding the source of errors during implementation.

4. **Framework Guide.** Following the same line, these may help remove validation threats regarding the source of errors during implementation.

5. **Overall Satisfaction.** This is the most subjective group as it aims to know the subjects' opinion about their own performance regarding the tasks at hand, and possible future tasks.

6. **Development Process.** These question pretend to access the type of problems they had to face, despite of their programming experience level.

7. **Future.** Aims to obtain an overall opinion about the experimental framework and how it could be used in the future.

6.2 Experiment Description

The following sections describe the experiment, especially the tasks given to the subjects, quoting the presented text. As referred in section 6.1 the problem was presented through a guide (Appendix B) which started by introducing the example used in section 4.2, as follows:

> *The online shop model as presented in the UML diagram below is composed by 5 entities (Address, Customer, Category, Product, and Order).*

Afterwards the full corresponding schema of the entities was presented. A possible implementation of the tasks using the experimental framework can be found in Appendix H.

6.2.1 Task 1 - Modeling

> *Create the necessary classes that represent the described model.*

Description

The first task consisted in creating classes that described the provided model. The subjects needed to use the UML diagram for understanding relation, the schema of the entities to know the types and the framework's guide to understand what would be the correct mapping. Implementations are expected to be similar, but in the baseline case there are slight details to be handled regarding foreign keys and default values. This task may have impact later on in Task 3 and during testing.

6.2.2 Task 2 - Create services

> *Implement the REST services for each of the entities as described in the table.*

Entity	Allowed Operations	Base path
Address	Create, List, Read, Update, Delete	/addresses
Customer	Create, Read, Update	/customers
Category	Create, List, Delete	/categories
Product	Create, List, Read, Update, Delete	/products
Order	Create, Read, Update	/orders

Table 6.2: Services specification for Task 2

Description

By the end of this task must exist a runnable but not complete API that exposes the modeled entities allowing only some operations. Implementation differs completely between treatments, being expected a higher implementation time for the baseline.

6.2.3 Task 3 - Customization

Reusing the previous services, implement the following constraints:

1. *When reading a list of products return only the available products.*
2. *On creation, an order must have at least one product.*
   - *If true use the default behavior.*
   - *Otherwise a BadRequest(400) response must be sent with the message 'Order must have at least one product'.*

Description

For completing the API it's requested the implementation of some customizations to the default behavior used by the end of Task 2. The main goal of this task is to test the usability of the DSL for this type of actions. It is expected a better result of the experimental framework as it unifies these kind of operations in one area of the code.

6.3 Data Analysis

In this section the subjects answers to the questionnaires (section 6.1.4) are analyzed. Let the null hypothesis be denoted as H_0, the alternative hypothesis as H_1, a group to compare as G_x, the compared group as G_y, and ρ the probability estimator of wrongly rejecting the null hypothesis. Then, the alternative hypothesis are either: (i) $H_1 : G_x \neq G_y$, the measure differs between groups, (ii) $H_1 : G_x < G_y$ or (iii) $H_1 : G_x > G_y$, the measure is greater in one of the groups. The following sections either compare the group 1 as G_1 with the group 2 as G_2, or a group under the baseline treatment as G_B with a group under the experimental treatment as G_E.

Due to the non-linear nature of the responses it is justified the use of non-parametric statistics. So the results are compared using the non-parametric, two-sample, rank-sum Wilcoxon-Mann-Whitney [HWC13] test, with $n1 = n2 = 4$ and significance level of 5%. Probability values of $\rho \leq 0.05$ are considered *significant*, and $\rho \leq 0.01$ considered *highly significant*. The raw data can be found in Appendix G.2.

6.3.1 Background

From these set of questions it's possible to reject the idea that the groups 1 and 2 present different skill levels to execute the requested tasks.

B1 I have considerable experience developing REST APIs

Let $H_1 : G_1 \neq G_2$, there was **no significant** difference ($\rho = 0.571$) in the scores of group 1 ($\bar{x} = 3.75$, $\sigma = 0.50$) and group 2 ($\bar{x} = 4.25$, $\sigma = 0.50$), as seen in Table 6.3. The subjects answered as expected as they had to develop REST APIs during some courses of the Master.

	Group 1			Group 2			Statistics		
	1 2 3 4 5	$\bar{x}$	σ	1 2 3 4 5	$\bar{x}$	σ	H_1	W	ρ
B1	_ _ ▬ ■ _	3.75	0.50	_ _ _ ■ ▬	4.25	0.50	$\neq$	04.5	0.571
B2	▬ _ ▬ ▬ _	2.25	1.50	■ _ _ _ ▬	2.00	2.00	$\neq$	09.0	1.000
B3	▬ _ ▬ ▬ _	2.25	1.50	■ _ _ _ _	1.00	0.00	$\neq$	12.0	0.429
B4	■ ▬ _ _ _	1.25	0.50	■ _ ▬ _ _	1.50	1.00	$\neq$	07.5	1.000
B5	_ _ ▬ ▬ _	3.50	0.58	_ ▬ ▬ _ ▬	3.75	1.50	$\neq$	07.0	1.000
B6	_ _ ▬ ▬ ▬	3.75	0.96	_ _ ▬ ■ _	3.75	0.50	$\neq$	07.5	1.000
B7	_ ▬ ▬ ▬ _	3.00	0.82	_ ▬ ■ _ _	2.75	0.50	$\neq$	09.5	1.000

Table 6.3: Summary of Background results

B2 I have considerable experience with Python

Let $H_1 : G_1 \neq G_2$, there was **no significant** difference ($\rho = 1.000$) in the scores of group 1 ($\bar{x} = 2.25$, $\sigma = 1.50$) and group 2 ($\bar{x} = 2.00$, $\sigma = 2.00$), as seen in Table 6.3. As the Python language is not mandatory in any of the Master's courses the low experience was expected. Having some admitted some experience, it was not sufficient to consider the groups different.

B3 I have considerable experience with the Django REST framework

Let $H_1 : G_1 \neq G_2$, there was **no significant** difference ($\rho = 0.429$) in the scores of group 1 ($\bar{x} = 2.25$, $\sigma = 1.50$) and group 2 ($\bar{x} = 1.00$, $\sigma = 0.00$), as seen in Table 6.3. In line with the previous question, it was not expected a great level of experience with the baseline framework. Despite of group 1 having the only two experienced subjects, it was not sufficient to reject the null hypothesis.

B4 I have considerable experience with Scala

Let $H_1 : G_1 \neq G_2$, there was **no significant** difference ($\rho = 1.000$) in the scores of group 1 ($\bar{x} = 1.25$, $\sigma = 0.50$) and group 2 ($\bar{x} = 1.50$, $\sigma = 1.00$), as seen in Table 6.3. Being the Scala language relatively new, it was not expected significant levels of experience.

B5 I have considerable experience with command-line terminals

Let $H_1 : G_1 \neq G_2$, there was **no significant** difference ($\rho = 1.000$) in the scores of group 1 ($\bar{x} = 3.50$, $\sigma = 0.58$) and group 2 ($\bar{x} = 3.75$, $\sigma = 1.50$), as seen in Table 6.3. This question intents to show that any of the results were influenced by uncomfortable use of the terminal as most developers use IDEs nowadays.

B6 I have considerable experience analyzing software documentation

Let $H_1 : G_1 \neq G_2$, there was **no significant** difference ($\rho = 1.000$) in the scores of group 1 ($\bar{x} = 3.75$, $\sigma = 0.96$) and group 2 ($\bar{x} = 3.75$, $\sigma = 0.50$), as seen in Table 6.3. The documentation patterns used to facilitate communication may require some time to learn. With these it has been

shown that the subjects had a moderate high level of experience with software documentation that didn't differ between the groups.

B7 I have considerable experience with test-driven development

Let $H_1 : G_1 \neq G_2$, there was **no significant** difference ($\rho = 1.000$) in the scores of group 1 ($\bar{x} = 3.00$, $\sigma = 0.82$) and group 2 ($\bar{x} = 2.75$, $\sigma = 0.50$), as seen in Table 6.3. Test-driven development was lectured in some courses but it's not always applied in practice by students. This question intended to understand if a possible different method for software development could influence the results.

6.3.2 External Factors

The results in this section reveal that the subjects were able to execute the tests comfortably with small or no effect of external factors.

	Group 1			Group 2			Statistics		
	1 2 3 4 5	$\bar{x}$	σ	1 2 3 4 5	$\bar{x}$	σ	H_1	W	ρ
EF1	▬ ▬ ▬ _ _	1.75	0.96	▬ ▬ _ _ _	1.25	0.50	$\neq$	10.5	0.714
EF2	_ _ ▬ ▬ ▬	4.00	0.82	_ _ ▬ _ ▬	4.50	1.00	$\neq$	05.0	0.486
EF3	▬ ▬ _ _ _	1.75	0.50	▬ ▬ _ _ _	1.50	0.58	$\neq$	10.0	1.000

Table 6.4: Summary of External Factors results

EF1 I felt disturbed and observed by the use of a screen-cast program

Let $H_1 : G_1 \neq G_2$, there was **no significant** difference ($\rho = 0.714$) in the scores of group 1 ($\bar{x} = 1.75$, $\sigma = 0.96$) and group 2 ($\bar{x} = 1.25$, $\sigma = 0.50$), as seen in Table 6.4. As the subjects were left isolated from the rest of the room and the screen-cast program was discrete, no relevant answers were expected due to this matter.

EF2 I enjoyed programming in the experiment

Let $H_1 : G_1 \neq G_2$, there was **no significant** difference ($\rho = 0.486$) in the scores of group 1 ($\bar{x} = 4.00$, $\sigma = 0.82$) and group 2 ($\bar{x} = 4.50$, $\sigma = 1.00$), as seen in Table 6.4. In general both groups enjoyed the experiment with one of the subjects being indifferent in reference to this matter.

EF3 I felt pressured to quickly finish the test

Let $H_1 : G_1 \neq G_2$, there was **no significant** difference ($\rho = 1.000$) in the scores of group 1 ($\bar{x} = 1.75$, $\sigma = 0.50$) and group 2 ($\bar{x} = 1.50$, $\sigma = 0.58$), as seen in Table 6.4. Despite being told that who was being tested were the frameworks, the subjects could fell the experiment as a challenge to prove themselves better than other participants. The low values and statistical equality enforces validity.

6.3.3 Problem Guide

The results in this section indicate that a validation threat w.r.t. the source of errors being caused by incomplete information shall be moderately considered.

	GROUP 1			GROUP 2			STATISTICS		
	1 2 3 4 5	$\bar{x}$	σ	1 2 3 4 5	$\bar{x}$	σ	H_1	W	ρ
PG1	_ ▬ ▬ _ ▬	3.75	1.50	_ ▬ _ ▬ ▬	4.00	1.41	$\neq$	07.5	1.000
PG2	_ _ _ ▬ ▬	4.50	0.58	_ _ _ ▬ ▬	4.75	0.50	$\neq$	06.0	1.000

Table 6.5: Summary of Problem Guide results

PG1 I read the complete guide

Let $H_1 : G_1 \neq G_2$, there was **no significant** difference ($\rho = 1.000$) in the scores of group 1 ($\bar{x} = 3.75$, $\sigma = 1.50$) and group 2 ($\bar{x} = 4.00$, $\sigma = 1.41$), as seen in Table 6.5. The result of this measure is unexpected as 37.5% admit not to have carefully read the guide. This may have no pratical effect as we can't reject the null hypothesis, which means that in case of negative effect that should be felt in both groups.

PG2 I completely understood what I read

Let $H_1 : G_1 \neq G_2$, there was **no significant** difference ($\rho = 1.000$) in the scores of group 1 ($\bar{x} = 4.50$, $\sigma = 0.58$) and group 2 ($\bar{x} = 4.75$, $\sigma = 0.50$), as seen in Table 6.5. These values demonstrate the quality of the provided documentation, despite whether it was used or not.

6.3.4 Framework Guide

The results in this section show that the provided framework guides don't pose a threat to the validity of the experiment.

	EXPERIMENTAL			BASELINE			STATISTICS		
	1 2 3 4 5	$\bar{x}$	σ	1 2 3 4 5	$\bar{x}$	σ	H_1	W	ρ
$FG1_1$	_ ▬ _ ▬ ▬	3.75	1.26	_ ▬ _ ▬ ▬	4.00	1.41	$\neq$	06.5	0.914
$FG1_2$	_ _ ▬ ▬ ▬	4.25	0.96	_ ▬ _ ▬ ▬	3.75	1.26		10.0	0.657
$FG2_1$	_ _ _ ▬ ▬	4.50	0.58	_ _ ▬ _ ▬	4.00	1.15	$\neq$	10.0	0.657
$FG2_2$	_ _ _ ▬ ▬	4.50	0.58	_ _ _ ▬ ▬	4.50	0.58		08.0	1.000

Table 6.6: Summary of Framework Guide results

FG1 I read the complete guide

Let $H_1 : G_E \neq G_B$: in the first round there was **no significant** difference ($\rho = 0.914$) in the scores for the experimental ($\bar{x} = 3.75$, $\sigma = 1.26$) and baseline ($\bar{x} = 4.00$, $\sigma = 1.41$) conditions; in

the second round there was **no significant** difference ($\rho = 0.657$) in the scores for the experimental ($\bar{x} = 4.25$, $\sigma = 0.96$) and baseline ($\bar{x} = 3.75$, $\sigma = 1.26$) conditions, as seen in Table 6.6. As the subjects had no prior experience with the experimental framework and almost none with the baseline, it was expected no difference that was shown in both rounds.

FG2 I completely understood what I read

Let $H_1 : G_E \neq G_B$: in the first round there was **no significant** difference ($\rho = 0.657$) in the scores for the experimental ($\bar{x} = 4.50$, $\sigma = 0.58$) and baseline ($\bar{x} = 4.00$, $\sigma = 1.15$) conditions; in the second round there was **no significant** difference ($\rho = 1.000$) in the scores for the experimental ($\bar{x} = 4.50$, $\sigma = 0.58$) and baseline ($\bar{x} = 4.50$, $\sigma = 0.58$) conditions, as seen in Table 6.6. These results ensures quality of the documentation despite the indifference of two subjects regarding this matter.

6.3.5 Overall Satisfaction

Below it's shown that developers find in Metamorphic an easier way to implement default behavior. Another four measures seem promising to be tested with larger subject groups.

	EXPERIMENTAL			BASELINE			STATISTICS		
	1 2 3 4 5	$\bar{x}$	σ	1 2 3 4 5	$\bar{x}$	σ	H_1	W	ρ
$OS1_1$	_ _ _ ▬ ■	4.75	0.50	_ ▬ _ ▬ _	3.00	1.15	$>$	15.0	0.043
$OS1_2$	_ _ ▬ ▬ ■	4.25	0.96	_ ▬ ■ _ ▬	3.25	1.26		12.0	0.200
$OS2_1$	_ _ _ _ ■	5.00	0.00	_ ▬ ■ ▬ _	3.00	0.82	$>$	16.0	0.014
$OS2_2$	_ _ _ ▬ ■	4.50	0.58	_ ▬ ■ _ _	2.75	0.50		16.0	0.014
$OS3_1$	_ _ _ _ ■	5.00	0.00	_ ▬ ■ _ _	2.50	0.58	$>$	16.0	0.014
$OS3_2$	_ _ _ ▬ ■	4.50	0.58	_ ▬ ■ _ ▬	3.25	1.26		13.0	0.100
$OS4_1$	_ _ ▬ _ ■	4.50	1.00	_ ▬ ▬ ▬ ▬	3.50	1.29	$>$	12.0	0.186
$OS4_2$	_ ▬ _ ▬ ▬	3.75	1.26	▬ ▬ _ ▬ ▬	3.00	1.83		10.0	0.386
$OS5_1$	_ _ _ ▬ ■	4.75	0.50	▬ ▬ ▬ ▬ _	2.50	1.29	$>$	15.5	0.029
$OS5_2$	_ _ _ _ ■	5.00	0.00	▬ ▬ ▬ _ ▬	2.75	1.71		14.0	0.071
$OS6_1$	_ _ _ ■ ▬	4.25	0.50	_ _ ■ ■ _	3.50	0.58	$>$	13.0	0.143
$OS6_2$	_ _ ▬ ■ ▬	4.00	0.82	_ _ ▬ ■ _	3.75	0.50		09.5	0.500

Table 6.7: Summary of Overall Satisfaction results

OS1 I found it easy and intuitive to specify a model

Let $H_1 : G_E > G_B$: in the first round there was **significant** difference ($\rho = 0.043$) in the scores for the experimental ($\bar{x} = 4.75$, $\sigma = 0.50$) and baseline ($\bar{x} = 3.00$, $\sigma = 1.15$) conditions; in the second round there was **no significant** difference ($\rho = 0.200$) in the scores for the experimental ($\bar{x} = 4.25$, $\sigma = 0.96$) and baseline ($\bar{x} = 3.25$, $\sigma = 1.26$) conditions, as seen in Table 6.7. In round 1 the null hypothesis is rejected, meaning that the experimental framework was considered better

to specify models. However, in the second round one would have to raise the significance level to 20% to show the same.

OS2 I found it easy and intuitive to specify default behavior

Let $H_1 : G_E > G_B$: in the first round there was **significant** difference ($\rho = 0.014$) in the scores for the experimental ($\bar{x} = 5.00$, $\sigma = 0.00$) and baseline ($\bar{x} = 3.00$, $\sigma = 0.82$) conditions; in the second round there was **significant** difference ($\rho = 0.014$) in the scores for the experimental ($\bar{x} = 4.50$, $\sigma = 0.58$) and baseline ($\bar{x} = 2.75$, $\sigma = 0.50$) conditions, as seen in Table 6.7. As expected in both rounds it was shown that the experimental framework was considered better to implement default behavior.

OS3 I found it easy and intuitive to specify custom behavior

Let $H_1 : G_E > G_B$: in the first round there was **significant** difference ($\rho = 0.014$) in the scores for the experimental ($\bar{x} = 5.00$, $\sigma = 0.00$) and baseline ($\bar{x} = 2.50$, $\sigma = 0.58$) conditions; in the second round there was **no significant** difference ($\rho = 0.100$) in the scores for the experimental ($\bar{x} = 4.50$, $\sigma = 0.58$) and baseline ($\bar{x} = 3.25$, $\sigma = 1.26$) conditions, as seen in Table 6.7. In round 1 the null hypothesis is rejected while in round 2 it is not, but with a promising probability of wrongly rejecting it.

OS4 I found it easy to make changes and rapidly test them

Let $H_1 : G_E > G_B$: in the first round there was **no significant** difference ($\rho = 0.186$) in the scores for the experimental ($\bar{x} = 4.50$, $\sigma = 1.00$) and baseline ($\bar{x} = 3.50$, $\sigma = 1.29$) conditions; in the second round there was **no significant** difference ($\rho = 0.386$) in the scores for the experimental ($\bar{x} = 3.75$, $\sigma = 1.26$) and baseline ($\bar{x} = 3.00$, $\sigma = 1.83$) conditions, as seen in Table 6.7. In both rounds the null hypothesis is accepted, but the higher means of the experimental seem to indicate that an experiment with a larger number of subjects would have different results.

OS5 I was able to create the application at least as rapidly as I could normally have created

Let $H_1 : G_E > G_B$: in the first round there was **significant** difference ($\rho = 0.029$) in the scores for the experimental ($\bar{x} = 4.75$, $\sigma = 0.50$) and baseline ($\bar{x} = 2.50$, $\sigma = 1.29$) conditions; in the second round there was **no significant** difference ($\rho = 0.071$) in the scores for the experimental ($\bar{x} = 5.00$, $\sigma = 0.00$) and baseline ($\bar{x} = 2.75$, $\sigma = 1.71$) conditions, as seen in Table 6.7. Similar to the previous questions, larger groups could result in stronger results.

OS6 I found that the resulting application could be used in production-level environments with minimal or no change

Let $H_1 : G_E > G_B$: in the first round there was **no significant** difference ($\rho = 0.143$) in the scores for the experimental ($\bar{x} = 4.25$, $\sigma = 0.50$) and baseline ($\bar{x} = 3.50$, $\sigma = 0.58$) conditions; in the second round there was **no significant** difference ($\rho = 0.500$) in the scores for the experimental ($\bar{x} = 4.00$, $\sigma = 0.82$) and baseline ($\bar{x} = 3.75$, $\sigma = 0.50$) conditions, as seen in Table 6.7.

Despite expecting the experimental to perform better than the baseline in this measure, the results indicate that Metamorphic transmits them at least the same level of confidence of Django REST, an already mature framework.

6.3.6 Development Process

In this section it's possible to infer that the experimental framework, compared with the baseline, leads to less difficulties when used for the first time. Also applications implemented with the experimental framework presented almost no runtime errors.

	EXPERIMENTAL			BASELINE			STATISTICS		
	1 2 3 4 5	$\bar{x}$	σ	1 2 3 4 5	$\bar{x}$	σ	H_1	W	ρ
$DP1_1$	▬ ▬ _ _ _	1.50	0.58	_ ▬ ▬ ▬ _	2.75	0.96	$<$	02.0	0.086
$DP1_2$	▬ ▬ ▬ _ _	1.75	0.96	▬ ▬ _ _ _	1.25	0.50		10.5	0.929
$DP2_1$	▬ ▬ _ _ _	1.25	0.50	_ ▬ ▬ _ _	2.50	0.58	$<$	01.0	0.043
$DP2_2$	▬ ▬ _ _ _	1.50	0.58	▬ ▬ ▬ _ _	1.75	0.96		07.0	0.500
$DP3_1$	▬ ▬ _ _ _	1.25	0.50	_ ▬ ▬ _ _	2.50	0.58	$<$	01.0	0.043
$DP3_2$	_ ▬ ▬ _ _	2.50	0.58	▬ _ _ ▬ _	2.50	1.73		08.0	0.629
$DP4_1$	_ ▬ _ ▬ ▬	3.75	1.26	_ _ ▬ _ ▬	4.00	1.15	$>$	07.0	0.629
$DP4_2$	_ ▬ ▬ ▬ _	2.75	0.96	▬ ▬ _ _ ▬	2.25	1.89		11.0	0.200
$DP5_1$	▬ ▬ _ _ _	1.25	0.50	_ _ _ ▬ ▬	4.25	0.50	$<$	00.0	0.014
$DP5_2$	▬ ▬ _ _ _	1.50	0.58	_ _ _ ▬ ▬	4.75	0.50		00.0	0.014
$DP6_1$	▬ _ _ ▬ ▬	3.50	1.73	▬ _ ▬ ▬ ▬	3.25	1.71	$\neq$	09.0	0.971
$DP6_2$	▬ ▬ _ _ ▬	2.25	1.89	_ _ _ ▬ ▬	4.50	0.58		03.0	0.171
$DP7_1$	▬ _ _ ▬ _	1.75	1.50	▬ ▬ ▬ _ _	2.00	0.82	$\neq$	05.5	0.486
$DP7_2$	▬ ▬ ▬ _ _	1.75	0.96	▬ ▬ ▬ _ _	1.75	0.96		08.0	1.000
$DP8_1$	▬ _ ▬ _ _	1.50	1.00	▬ ▬ _ _ _	1.50	0.58	$\neq$	07.0	1.000
$DP8_2$	▬ ▬ _ _ _	1.50	0.58	▬ _ _ ▬ _	1.75	1.50		09.0	1.000

Table 6.8: Summary of Development Process results

DP1 Most of my difficulties were implementing the model

Let $H_1 : G_E < G_B$: in the first round there was **no significant** difference ($\rho = 0.086$) in the scores for the experimental ($\bar{x} = 1.50$, $\sigma = 0.58$) and baseline ($\bar{x} = 2.75$, $\sigma = 0.96$) conditions; in the second round there was **no significant** difference ($\rho = 0.929$) in the scores for the experimental ($\bar{x} = 1.75$, $\sigma = 0.96$) and baseline ($\bar{x} = 1.25$, $\sigma = 0.50$) conditions, as seen in Table 6.8. The experimental framework leads to less modeling difficulties when using for the first time, as in round 1 the null hypothesis is almost rejected and in round 2 is far from being rejected.

DP2 Most of my difficulties were implementing the default behavior

Let $H_1 : G_E < G_B$: in the first round there was **significant** difference ($\rho = 0.043$) in the scores for the experimental ($\bar{x} = 1.25$, $\sigma = 0.50$) and baseline ($\bar{x} = 2.50$, $\sigma = 0.58$) conditions; in the second round there was **no significant** difference ($\rho = 0.500$) in the scores for the experimental ($\bar{x} = 1.50$, $\sigma = 0.58$) and baseline ($\bar{x} = 1.75$, $\sigma = 0.96$) conditions, as seen in Table 6.8. In line with the last question, when used for the first time the experimental framework leads to less difficulties when implementing default behavior.

DP3 Most of my difficulties were implementing the custom behavior

Let $H_1 : G_E < G_B$: in the first round there was **significant** difference ($\rho = 0.043$) in the scores for the experimental ($\bar{x} = 1.25$, $\sigma = 0.50$) and baseline ($\bar{x} = 2.50$, $\sigma = 0.58$) conditions; in the second round there was **no significant** difference ($\rho = 0.629$) in the scores for the experimental ($\bar{x} = 2.50$, $\sigma = 0.58$) and baseline ($\bar{x} = 2.50$, $\sigma = 1.73$) conditions, as seen in Table 6.8. Just like in the previous question, it's possible to reject the null hypothesis only in the first round.

DP4 Most of my difficulties were fixing compile errors

Let $H_1 : G_E > G_B$: in the first round there was **no significant** difference ($\rho = 0.629$) in the scores for the experimental ($\bar{x} = 3.75$, $\sigma = 1.26$) and baseline ($\bar{x} = 4.00$, $\sigma = 1.15$) conditions; in the second round there was **no significant** difference ($\rho = 0.200$) in the scores for the experimental ($\bar{x} = 2.75$, $\sigma = 0.96$) and baseline ($\bar{x} = 2.25$, $\sigma = 1.89$) conditions, as seen in Table 6.8. These results seem to indicate that the subjects either didn't understand the question or were not able distinguish that the baseline framework didn't require compilation. As so the results cannot be properly interpreted.

DP5 Most of my difficulties were fixing runtime errors

Let $H_1 : G_E < G_B$: in the first round there was **significant** difference ($\rho = 0.014$) in the scores for the experimental ($\bar{x} = 1.25$, $\sigma = 0.50$) and baseline ($\bar{x} = 4.25$, $\sigma = 0.50$) conditions; in the second round there was **significant** difference ($\rho = 0.014$) in the scores for the experimental ($\bar{x} = 1.50$, $\sigma = 0.58$) and baseline ($\bar{x} = 4.75$, $\sigma = 0.50$) conditions, as seen in Table 6.8. Despite the confusion in the previous question, the results for this seem consistent. In both rounds, the experimental framework shown itself not to trouble developers with runtime errors.

DP6 Most of my difficulties were understanding failed tests

Let $H_1 : G_E \neq G_B$: in the first round there was **no significant** difference ($\rho = 0.971$) in the scores for the experimental ($\bar{x} = 3.50$, $\sigma = 1.73$) and baseline ($\bar{x} = 3.25$, $\sigma = 1.71$) conditions; in the second round there was **no significant** difference ($\rho = 0.171$) in the scores for the experimental ($\bar{x} = 2.25$, $\sigma = 1.89$) and baseline ($\bar{x} = 4.50$, $\sigma = 0.58$) conditions, as seen in Table 6.8. Understanding of failed tests should not depend on the framework being used as concluded.

DP7 Most of my difficulties were related with documentation

Let $H_1 : G_E \neq G_B$: in the first round there was **no significant** difference ($\rho = 0.486$) in the scores for the experimental ($\bar{x} = 1.75$, $\sigma = 1.50$) and baseline ($\bar{x} = 2.00$, $\sigma = 0.82$) conditions; in the second round there was **no significant** difference ($\rho = 1.000$) in the scores for the experimental ($\bar{x} = 1.75$, $\sigma = 0.96$) and baseline ($\bar{x} = 1.75$, $\sigma = 0.96$) conditions, as seen in Table 6.8. These results come in line with the results in sections 6.3.3 and 6.3.4, meaning that documentation doesn't pose a threat to validity.

DP8 Most of my difficulties were related with the development environment (terminal, text editor, etc)

Let $H_1 : G_E \neq G_B$: in the first round there was **no significant** difference ($\rho = 1.000$) in the scores for the experimental ($\bar{x} = 1.50$, $\sigma = 1.00$) and baseline ($\bar{x} = 1.50$, $\sigma = 0.58$) conditions; in the second round there was **no significant** difference ($\rho = 1.000$) in the scores for the experimental ($\bar{x} = 1.50$, $\sigma = 0.58$) and baseline ($\bar{x} = 1.75$, $\sigma = 1.50$) conditions, as seen in Table 6.8. Unable to reject the null hypothesis in both rounds, the development environment should not pose as a threat to validity.

6.3.7 Future

Both groups are consistent in considering the experimental framework as a good tool not only for prototyping but also for production-level applications.

	Group 1			Group 2			Statistics		
	1 2 3 4 5	$\bar{x}$	σ	1 2 3 4 5	$\bar{x}$	σ	H_1	W	ρ
F1	_ _ _ ■ ■	4.50	0.58	_ _ _ ▬ ■	4.75	0.50	$\neq$	06.0	1.000
F2	_ _ _ ■ ▬	4.25	0.50	_ _ _ ■ ▬	4.25	0.50	$\neq$	08.0	1.000

Table 6.9: Summary of Future results

F1 In the future I would use the Metamorphic framework for rapid prototyping of a REST API

Let $H_1 : G_1 \neq G_2$, there was **no significant** difference ($\rho = 1.000$) in the scores of group 1 ($\bar{x} = 4.50$, $\sigma = 0.58$) and group 2 ($\bar{x} = 4.75$, $\sigma = 0.50$), as seen in Table 6.9. Both groups are clear and admit to use the framework for prototyping a REST API.

F2 In the future I would use the Metamorphic framework for developing a production-level REST API

Let $H_1 : G_1 \neq G_2$, there was **no significant** difference ($\rho = 1.000$) in the scores of group 1 ($\bar{x} = 4.25$, $\sigma = 0.50$) and group 2 ($\bar{x} = 4.25$, $\sigma = 0.50$), as seen in Table 6.9. In comparison with the previous question the subjects are more moderate in both groups but still trust in its application in production.

6.4 Objective Measurement

Using the screen-cast recordings and some manually take notes a set of measurement were taken in order to validate the framework more objectively. In the following sections are presented measurements related to errors, time, lines of code and others. The raw data for this measurements can be found in Appendix G.3.

6.4.1 Errors Measurement

To measure if the framework is in fact error preventive (section 1.3.4) the means of runtime errors and non-runtime errors were compared, as shown in Table 6.10. For non-runtime errors it is meant compile errors and errors that occur on applications initialization. As expected the experimental conditions leads to almost no runtime errors in contrast with baseline conditions. The opposite happens for non-runtime errors but being the mean amount of errors slightly lower.

Measurement	Round	$\bar{x}_E$	σ_E	$\bar{x}_B$	σ_B
# Non-runtime errors	1	02.8	2.06	01.5	1.29
	2	04.3	1.71	00.3	0.50
# Runtime errors	1	00.0	0.00	05.8	4.99
	2	00.3	0.50	05.5	1.73

Table 6.10: Statistics of error measurements

6.4.2 Time Measurement

Development time is a rich measurement as it is an highly dependent variable, depends of experience, depends of the language syntax, depends of the language type system, and depends of framework's usability. With this in mind the time of each task and the total time were registered. Task were considered to be finished when a developer tried to test the application or started the next task.

Measurement	Round	$\bar{x}_E$	σ_E	$\bar{x}_B$	σ_B	$\bar{x}_B - \bar{x}_E$	$(\bar{x}_B - \bar{x}_E)/\bar{x}_B$
Task 1	1	11.44	03.11	20.13	09.18	08.69	43.2%
	2	07.94	01.48	08.81	03.06	00.87	09.9%
Task 2	1	08.31	06.46	22.94	13.73	14.63	63.8%
	2	10.69	05.76	17.25	01.49	06.56	38.0%
Task 3	1	05.06	02.12	03.92	02.67	-1.14	-29.1%
	2	05.31	01.91	03.44	00.97	-1.87	-54.4%
Total	1	45.81	08.19	70.56	12.14	24.75	35.1%
	2	31.50	01.86	51.38	08.29	19.88	38.7%

Table 6.11: Full time and tasks time measurements in minutes

The results in Table 6.11 confirms the conclusion from section 6.3.6 that on a first use the experimental framework is easier to use for implementing the first and the second task. However the times for the third task are better in baseline conditions, contradicting some previous conclusions, which is believed to be the result of requesting customization of two operations. But still the overall time, tasks plus debugging, was confirmed to decrease on average by at least 35% in experimental conditions.

6.4.3 Lines of Code Measurement

Another possible indicator of a framework's usability is the amount of produced code, which can be represented in terms of lines of code. As shown in Table 6.12, in experimental conditions the number of lines of code may reduced on average at least 28.8%, and this number increases to 42% if lines that only contain are discard from the count.

MEASUREMENT	ROUND	$\bar{x}_E$	σ_E	$\bar{x}_B$	σ_B	$\bar{x}_B - \bar{x}_E$	$(\bar{x}_B - \bar{x}_E)/\bar{x}_B$
# Lines of code	1	65.5	5.00	92.0	9.35	26.5	28.8%
	2	62.5	1.00	86.3	9.64	29.5	34.2%
# Lines of code (no braces)	1	49.3	0.96	92.0	9.35	42.7	46.4%
	2	49.5	1.00	86.3	9.64	36.8	42.6%

Table 6.12: Statistics of lines of code measurements

6.4.4 Other Measurement

The remainder measurements statistics are presented in Table 6.13. These indicated that there were no significant difference between the number of requests for help and the number of times application was started. However the number of times a subject had to run the tests was lower in experimental conditions, which also was lower when just comparing rounds.

MEASUREMENT	ROUND	$\bar{x}_E$	σ_E	$\bar{x}_B$	σ_B
# Requests for help	1	03.0	2.16	04.0	2.45
	2	02.0	1.41	02.0	1.63
# Application runs	1	06.0	2.00	07.8	4.50
	2	07.0	3.16	07.3	3.40
# Tests runs	1	04.8	2.22	09.0	3.74
	2	02.3	2.50	07.5	2.65

Table 6.13: Statistics of miscellaneous of measurements

6.5 VALIDATION THREATS

Validation threats are conditions or situations that in some way may incorrectly change the results of the study. This means that if they are not correctly handled the study may be invalid. The

identified validation threats are as follows:

- **Difference of experience with the languages.** It could happen that each of groups had different experience using either Python or Scala, which would be translated into biased results in favor of one of the treatments. The results of questions B2 and B4 in section 6.3.1 discard this threat.

- **Difference of experience with the frameworks.** The subjects had no previous experience with the experimental framework but could have experience with the baseline framework. The results of question B3 in section 6.3.1 discard this threat.

- **Unsuitable development environment.** As subjects could only use a text editor and the terminal instead of an IDE, they could fell uncomfortable and not perform the tasks as if it was a real situation. The results of B5 in section 6.3.1 discard this threat.

- **Reduced number of subjects.** The sample size is very small mainly due to lack of time and human resources. This threat was diminished by executing two rounds so each subject would receive the two treatments.

- **Intrusive observation procedures.** Being subjects aware that the screen was being recorded, their performance could be limited as they might feel observed of judged. The results of questions EF1 and EF3 in section 6.3.2 discard this threat.

- **Incomplete information.** Subjects may misinterpret the guides or not completely read them and as so were executing the tasks fully informed. The results of questions in sections 6.3.3 and 6.3.4 discard this threat.

6.6 Conclusion

As already pointed out, this experiment attempted to assert that the Metamorphic framework is *quick and easy to use* and that is *error preventive* (section 1.3.4).

The first statement is favorably supported by the results to questions OS1, OS2, OS3, and OS5 in section 6.3.5 which conclude that implementations of default behavior are easier and more intuitive to do using Metamorphic. These also conclude that: there is a high chance that implementations of models and customizations are easier and more intuitive; and implementations can be created faster. This last is also corroborated by time measurements (section 6.4.2) which indicate that development time may decrease 35%, and by lines of code measurements (section 6.4.3) that indicate that the quantity of code may decrease 28%.

The second statement is favorably supported by the results to question DP5 in section 6.3.6 which indicates that the applications built with the Metamorphic framework barely have runtime errors. This is also corroborated by the runtime errors measurement (section 6.4.1) which concludes a large difference to the baseline framework. At last, the number of test runs measurement

(section 6.4.4) indicates that application built with Metamorphic require less iterations to validate all the tests.

To conclude, some experiment subjects commented that the Metamorphic framework is capable of reducing lost of boilerplate code which is positive. Probably with that in mind, subjects showed great confidence in the framework for rapid prototyping of a REST API or develop a production-level REST API.

Chapter 7

Conclusions

To conclude it is presented a summary of the document (section 7.1), this work contributions (section 7.2) and some possible future work (section 7.3).

7.1 Summary

This document starts by explaining the context of model-driven REST frameworks and what are the identified problems. From these problems, a set of goals were established that were explored throughout this dissertation. Underlying concepts to the research problem were established and explored which enabled the design and implementation of Metamorphic, a model-driven REST framework in Scala, that served as a proof of concept to the research problem.

Validation was achieved through benchmarks and an academic quasi-experiment. The first revealed that with almost no effort spent in optimization, it was possible to implement a framework with performance near to be the most performant model-driven REST framework. The second concluded that the characteristics of Metamorphic enable faster development, approximately 35% less time, due to its error preventive nature. This experiment also accessed that the specific developed framework is easy to use, as on average it can be assumed gains of 42% in terms of lines of code.

7.2 Contributions

Development of this dissertation may be resumed in the following results:

1. **Research on model-driven REST frameworks.** This information, and here gathered, has not been systematized, treated, analyzed, and distributed in a public repository. The scientific community may now have a source of information in which they can base their work on, without having to do general concept related searches.

2. **The specification of a meta-architecture.** Both the metametamodel and application logic model, may be used in other circumstances as they are independent from any programming

language. They also may be used as a starting point for something with greater complexity, given their proven basic applicabilities.

3. **A verified implementation of a model-driven REST framework in Scala.** Metamorphic besides being a reference implementation of the research problem, may help researchers and industry partners to improve their development process. Due to the framework's architecture developers may easily implement generators that fit their needs without having to address modeling issues and maintainability associated with it. The framework is publicly available at github.com/frroliveira/metamorphic.

4. **Benchmarks.** Developers may now choose, with objective information, between any of the tested model-driven REST frameworks. Depending on their requirements, i.e. type of entities and operations, the results in section 5.3 can help make a decision.

5. **Academic quasi-experiment.** The results from the experiment may reveal what to expect from a framework with such configuration. Information contained in the results may help answer questions like: "What does developers think about the approach?"; "Will it be accepted by the community?". Depending on the work one might have, this may be useful information.

7.3 Future Work

Validation of the proof of concept revealed that the response to the research problem (section 1.3) is positive. This means that it is possible to improve the development process and execution performance of model-based REST services, through a framework that is written in a statically type-safe programming language that enables code generation in compile time. Further research of this problem would be connected with Metamorphic as it's not yet a full featured framework. So some future work could be:

1. **Test models flexibility.** Knowledge on Metamorphic's flexibility to other generators is empirical. To ensure such information another repository and service generators could be implemented based in other libraries.

2. **Extension of models.** For a framework to be useful, it may contain most features developers will need. Metamorphic has the basic features, so new features would be welcome such as entity inheritance, authentication, database migrations, filtering, and pagination.

3. **Profiling.** Understanding any possible bottleneck in generated applications could be done trough profiling. This would aim to fully validate the performance goal initially established.

4. **Experiments.** Having a new and more mature version of the framework, its validity should be tested in industrial environments with samples that are more size significant. This time both versions, synchronous and asynchronous, should be tested.

5. **Swagger definition.** Swagger [Sma] defines a "a standard, language-agnostic interface to REST APIs which allows both humans and computers to discover and understand the capabilities of the service without access to source code". This would allow faster testing of the generated services.

It would be wiser to finish this set of activities before attempting to prove the research problem for any considerable language.

References

[Art] Artima, Inc. ScalaTest. http://scalatest.org/ [Online; accessed June 27, 2015]. Cited on page 38.

[Ash09] Kevin Ashton. That 'internet of things' thing. *RFiD Journal*, 22(7):97–114, 2009. Cited on page 1.

[AZW06] Uwe Aßmann, Steffen Zschaler, and Gerd Wagner. Ontologies, meta-models, and the model-driven paradigm. In *Ontologies for software engineering and software technology*, pages 249–273. Springer, 2006. Cited on page 9.

[BGW93] Daniel G Bobrow, Richard P Gabriel, and Jon L White. Clos in context-the shape of the design space. *Object Oriented Programming: The CLOS Perspective*, pages 29–61, 1993. Cited on page 9.

[Blu] BlueEyes. BlueEyes. https://github.com/jdegoes/blueeyes [Online; accessed February 15, 2015]. Cited on page 26.

[BOV+13] Eugene Burmako, Martin Odersky, Christopher Vogt, Stefan Zeiger, and Adriaan Moors. Scala macros, November 2013. http://scalamacros.org/paperstalks/2013-11-25-ScalaMacrosPoster.pdf [Online; accessed February 13, 2015]. Cited on pages 12 and 13.

[Bur] Eugene Burmako. Scala Macros. http://scalamacros.org/ [Online; accessed June 11, 2015]. Cited on page 11.

[Bur13] Eugene Burmako. Scala macros: Let our powers combine!: On how rich syntax and static types work with metaprogramming. In *Proceedings of the 4th Workshop on Scala*, SCALA '13, pages 3:1–3:10, New York, NY, USA, 2013. ACM. Cited on pages 11, 12, and 13.

[Bur14] Eugene Burmako. Roadmap for Scala macros, July 2014. http://scalamacros.org/news/2014/07/16/roadmap-for-scala-macros.html [Online; accessed January 27, 2015]. Cited on page 10.

[Car96] Luca Cardelli. Type systems. *ACM Computing Surveys*, 28(1):263–264, 1996. Cited on page 13.

[Chr] Tom Christie. Django REST framework. http://www.django-rest-framework.org/ [Online; accessed January 15, 2015]. Cited on pages 2, 22, and 41.

[CI84] Robert D. Cameron and M. Robert Ito. Grammar-based definition of metaprogramming systems. *ACM Trans. Program. Lang. Syst.*, 6(1):20–54, January 1984. Cited on page 9.

[CR91] William Clinger and Jonathan Rees. Macros that work. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '91, pages 155–162, New York, NY, USA, 1991. ACM. Cited on page 10.

[Dav] David Pollak. Simply Lift. http://simply.liftweb.net/ [Online; accessed February 15, 2015]. Cited on page 26.

[DS10] Lisa Dusseault and James Snell. PATCH Method for HTTP, March 2010. http://tools.ietf.org/html/rfc5789 [Online; accessed January 22, 2015]. Cited on page 16.

[EPF] École Polytechnique Fédérale de Lausanne - EPFL. The Scala Programming Language. http://www.scala-lang.org/ [Online; accessed February 5, 2015]. Cited on page 3.

[Exp] Express. Express - Node.js web application framework. http://expressjs.com/ [Online; accessed February 15, 2015]. Cited on page 24.

[Fac] Facebook. Graph API. https://developers.facebook.com/docs/graph-api [Online; accessed February 4, 2015]. Cited on page 1.

[FAF09] H.S. Ferreira, A. Aguiar, and J.P. Faria. Adaptive object-modelling: Patterns, tools and applications. In *Software Engineering Advances, 2009. ICSEA '09. Fourth International Conference on*, pages 530–535, Sept 2009. Cited on page 19.

[FB96] Ned Freed and Nathaniel Borenstein. Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types, November 1996. http://tools.ietf.org/html/rfc2046 [Online; accessed January 22, 2015]. Cited on page 17.

[Fie00] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000. Cited on pages 1, 14, and 15.

[Fie08a] Roy Thomas Fielding. No REST in CMIS, September 2008. http://roy.gbiv.com/untangled/2008/no-rest-in-cmis [Online; accessed January 20, 2015]. Cited on page 15.

[Fie08b] Roy Thomas Fielding. REST APIs must be hypertext-driven, October 2008. http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven [Online; accessed January 20, 2015]. Cited on page 15.

[Fie14] Roy Thomas Fielding. Roy Fielding on Versioning, Hypermedia, and REST, December 2014. http://www.infoq.com/articles/roy-fielding-on-versioning [Online; accessed January 20, 2015]. Cited on page 15.

[FLR14] Roy Thomas Fielding, Yves Lafon, and Julian Reschke. Hypertext Transfer Protocol (HTTP/1.1): Range Requests, June 2014. http://tools.ietf.org/html/rfc7233 [Online; accessed January 22, 2015]. Cited on page 16.

[FNR14] Roy Thomas Fielding, Mark Nottingham, and Julian Reschke. Hypertext Transfer Protocol (HTTP/1.1): Caching, June 2014. http://tools.ietf.org/html/rfc7234 [Online; accessed January 22, 2015]. Cited on pages 16 and 18.

[For15] IETF (Internet Engineering Task Force). Hypertext Transfer Protocol (HTTP) Authentication Scheme Registry, January 2015. http://www.iana.org/assignments/http-authschemes/http-authschemes.xhtml [Online; accessed January 26, 2015]. Cited on page 18.

[Foua] Django Software Foundation. Django. https://www.djangoproject.com/ [Online; accessed January 15, 2015]. Cited on page 22.

[Foub] Node.js Foundation. About | Node.js. https://nodejs.org/about/ [Online; accessed June 18, 2015]. Cited on page 24.

[FR14a] Roy Thomas Fielding and Julian Reschke. Hypertext Transfer Protocol (HTTP/1.1): Authentication, June 2014. http://tools.ietf.org/html/rfc7235 [Online; accessed January 22, 2015]. Cited on pages 16 and 18.

[FR14b] Roy Thomas Fielding and Julian Reschke. Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests, June 2014. http://tools.ietf.org/html/rfc7232 [Online; accessed January 22, 2015]. Cited on pages 16 and 17.

[FR14c] Roy Thomas Fielding and Julian Reschke. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing, June 2014. http://tools.ietf.org/html/rfc7230 [Online; accessed January 22, 2015]. Cited on page 16.

[FR14d] Roy Thomas Fielding and Julian Reschke. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content, June 2014. http://tools.ietf.org/html/rfc7231 [Online; accessed January 22, 2015]. Cited on page 16.

[fra] Django REST framework. Quickstart. http://www.django-rest-framework.org/tutorial/quickstart/ [Online; accessed February 14, 2015]. Cited on page 22.

[Goo] Google. Google Calendar API. https://developers.google.com/google-apps/calendar/ [Online; accessed February 4, 2015]. Cited on page 1.

[Gro] The PHP Group. PHP: PDO - Manual. http://php.net/manual/en/class.pdo.php [Online; accessed February 10, 2015]. Cited on page 20.

[HWC13] Myles Hollander, Douglas A Wolfe, and Eric Chicken. *Nonparametric statistical methods*. John Wiley & Sons, 2013. Cited on page 55.

[Iara] Nicola Iarocci. Python REST API Framework — Eve 0.5 documentation. http://python-eve.org/ [Online; accessed January 15, 2015]. Cited on pages 23 and 42.

[Iarb] Nicola Iarocci. Quickstart. http://python-eve.org/quickstart.html [Online; accessed February 14, 2015]. Cited on page 23.

[Inc] Typesafe Inc. Slick. http://slick.typesafe.com/ [Online; accessed February 12, 2015]. Cited on page 37.

[Jaz07] Mehdi Jazayeri. Some trends in web application development. In *Future of Software Engineering, 2007. FOSE'07*, pages 199–213. IEEE, 2007. Cited on page 2.

[Jef] Jeff Barczewski. jeffbski/bench-rest. https://github.com/jeffbski/bench-rest [Online; accessed June 23, 2015]. Cited on page 42.

[JF88] Ralph E Johnson and Brian Foote. Designing reusable classes. *Journal of object-oriented programming*, 1(2):22–35, 1988. Cited on page 19.

[KFB14] Jacek Kopecký, Paul Fremantle, and Rich Boakes. A history and future of web apis. *it - Information Technology*, 56(3):90–97, 2014. Cited on pages 1 and 2.

[LFA+05] Daniel Lucrédio, Renata PM Fortes, Alexandre Alvaro, ESd Almeida, and Silvio RL Meira. Towards a model-driven reuse process. In *31st IEEE EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), Work in Progress Session, Porto, Portugal. IEEE Computer Society, Los Alamitos*, 2005. Cited on page 8.

[Lik32] Rensis Likert. A technique for the measurement of attitudes. *Archives of psychology*, 1932. Cited on page 53.

[MBH] Heather Miller, Eugene Burmako, and Philipp Haller. Reflection - Overview - Scala Documentation. http://docs.scala-lang.org/overviews/reflection/overview.html [Online; accessed January 27, 2015]. Cited on page 10.

[McN] Mike McNeil. Sails.js | Realtime MVC Framework for Node.js. http://sailsjs.org/ [Online; accessed June 17, 2015]. Cited on pages 24 and 41.

[Mic] Microsoft. Entity Framework. http://www.asp.net/entity-framework [Online; accessed February 10, 2015]. Cited on page 20.

[OAC+04] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. The scala language specification, 2004. Cited on page 3.

[Obj] Object Management Group, Inc. MOF. http://www.omg.org/spec/MOF/ [Online; accessed July 20, 2015]. Cited on pages 9 and 29.

[Otwa] Taylor Otwell. HTTP Routing. http://laravel.com/docs/5.0/routing [Online; accessed February 10, 2015]. Cited on page 20.

[Otwb] Taylor Otwell. Schema Builder. http://laravel.com/docs/5.0/schema [Online; accessed February 10, 2015]. Cited on page 20.

[Par] Parse. Parse. https://parse.com/ [Online; accessed February 4, 2015]. Cited on page 1.

[PAUP12] Pavan Kumar Potti, Sanjay Ahuja, Karthikeyan Umapathy, and Zornitza Prodanoff. Comparing performance of web service interaction styles: Soap vs. rest. In *Proceedings of the Conference on Information Systems Applied Research ISSN*, volume 2167, page 1508, 2012. Cited on page 2.

[Rac] Rackspace. Transactional Email API Service for Developers - Mailgun. http://www.mailgun.com/ [Online; accessed February 4, 2015]. Cited on page 1.

[Res13] ABI Research. More Than 30 Billion Devices Will Wirelessly Connect to the Internet of Everything in 2020, May 2013. https://www.abiresearch.com/press/more-than-30-billion-devices-will-wirelessly-conne [Online; accessed February 3, 2015]. Cited on page 1.

[RFBLO01] Dirk Riehle, Steven Fraleigh, Dirk Bucka-Lassen, and Nosa Omorogbe. The architecture of a uml virtual machine. *SIGPLAN Not.*, 36(11):327–341, October 2001. Cited on page 8.

[Ron] Armin Ronacher. Flask. http://flask.pocoo.org/ [Online; accessed February 14, 2015]. Cited on page 23.

[Sch06] Douglas C. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):25–31, 2006. Cited on page 7.

[Sma] SmartBear. Swagger | The World's Most Popular Framework for APIs. http://swagger.io/ [Online; accessed June 28, 2015]. Cited on page 71.

[Soc] Internet Society. Global Internet Report 2014. http://www.internetsociety.org/sites/default/files/Global_Internet_Report_2014_0.pdf [Online; accessed February 3, 2015]. Cited on page 1.

[SRGT14] Jonathan Sprinkle, Matti Rossi, Jeff Gray, and Juha-Pekka Tolvanen. Dsm'14: The 14th workshop on domain-specific modeling. In *Proceedings of the Companion Publication of the 2014 ACM SIGPLAN Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '14, pages 73–74, New York, NY, USA, 2014. ACM. Cited on page 9.

[Stra] StrongLoop. Create a simple API. http://docs.strongloop.com/display/public/LB/Create+a+simple+API [Online; accessed February 15, 2015]. Cited on page 24.

[Strb] StrongLoop. LoopBack. http://loopback.io/ [Online; accessed January 15, 2015]. Cited on pages 24 and 41.

[Tuk77] John W Tukey. Exploratory data analysis. 1977. Cited on page 43.

[Typa] Typesafe Inc. Akka. http://akka.io/ [Online; accessed February 15, 2015]. Cited on page 26.

[Typb] Typesafe Inc. spray | REST/HTTP for your Akka/Scala Actors. http://spray.io/ [Online; accessed February 15, 2015]. Cited on page 26.

[Typc] Typesafe Inc. Typesafe gets Spray(ed). http://www.typesafe.com/blog/typesafe-gets-sprayed [Online; accessed February 15, 2015]. Cited on page 26.

[Typd] Typesafe Inc. typesafehub/config. https://github.com/typesafehub/config [Online; accessed June 24, 2015]. Cited on page 33.

[Type] Typesafe Inc and Zengularity. Play Framework - Build Modern & Scalable Web Apps with Java and Scala. https://www.playframework.com/ [Online; accessed February 15, 2015]. Cited on page 26.

[Unf] Unfiltered. Unfiltered. http://unfiltered.databinder.net/Unfiltered.html [Online; accessed February 15, 2015]. Cited on page 26.

[Van] Vance Lucas. REST API Testing—Frisby.js. http://frisbyjs.com/ [Online; accessed June 29, 2015]. Cited on page 39.

[VDKV00] Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *Sigplan Notices*, 35(6):26–36, 2000. Cited on page 9.

[W3C14] W3C. Cross-Origin Resource Sharing, January 2014. http://www.w3.org/TR/cors/ [Online; accessed February 10, 2015]. Cited on page 22.

[WBJ90] Rebecca J Wirfs-Brock and Ralph E Johnson. Surveying current research in object-oriented design. *Communications of the ACM*, 33(9):104–124, 1990. Cited on page 19.

[WPR10] Jim Webber, Savas Parastatidis, and Ian Robinson. *REST in Practice: Hypermedia and Systems Architecture*. O'Reilly Media, Inc., 1st edition, 2010. Cited on page 15.

[Xit] Xitrum. Xitrum. http://xitrum-framework.github.io/ [Online; accessed February 15, 2015]. Cited on page 26.

[ZW98] Marvin V Zelkowitz and Dolores R. Wallace. Experimental models for validating technology. *Computer*, 31(5):23–31, 1998. Cited on page 4.

Appendix A

Benchmarks Statistics

Ent. Type	Framework	$\bar{x}$	σ	*min*	*p*25	$\widetilde{x}$	*p*75	*max*
Simple	MA	0062.78	0027.26	0014.08	0057.22	0059.27	0064.26	0510.83
	MT	0173.12	0028.13	0022.19	0158.43	0166.52	0170.25	0500.59
	LB	0059.60	0018.85	0014.34	0055.69	0058.27	0060.84	0394.58
	SL	0082.32	0015.92	0062.54	0074.84	0080.32	0085.34	0363.03
	DJ	0112.39	0070.06	0046.48	0089.15	0107.97	0129.35	2254.94
	D3	0113.32	0074.38	0045.84	0088.98	0108.48	0129.65	2281.71
	EV	0026.11	0002.57	0012.41	0025.31	0025.66	0026.09	0044.83
Has Object	MA	0061.55	0013.22	0009.97	0057.87	0059.56	0064.82	0168.81
	MT	0175.21	0023.62	0012.41	0166.06	0166.66	0168.81	0397.33
	LB	0059.83	0014.82	0008.96	0056.51	0058.47	0060.73	0202.36
	SL	0083.28	0011.56	0063.53	0075.45	0082.26	0087.17	0171.84
	DJ	0116.57	0032.56	0046.81	0091.36	0111.83	0136.31	0304.67
	D3	0117.18	0031.77	0049.97	0093.32	0114.43	0137.31	0308.56
	EV	0034.82	0002.63	0004.66	0033.68	0034.19	0035.16	0054.48
Has List	MA	0063.72	0024.21	0022.89	0041.43	0072.57	0081.91	0190.84
	MT	0262.90	0039.06	0051.96	0242.08	0249.94	0274.74	0741.89
	LB	0059.47	0012.62	0011.34	0056.91	0058.64	0061.06	0213.90
	SL	1487.77	0830.53	0129.07	2351.01	2560.76	2752.53	3948.83
	DJ	0146.93	0038.85	0067.74	0121.43	0141.81	0165.85	0453.71
	D3	0145.15	0032.59	0062.05	0121.01	0142.69	0166.24	0332.79
	EV	0050.96	0003.41	0009.47	0048.93	0049.79	0052.03	0067.24

Table A.1: Benchmark statistics for the Create operation including means, standard deviations, minimum values, 25^{th} percentiles, medians, 75^{th} percentiles, and maximum values. Framework implementations are identified as follows: (MA) Metamorphic Async, (MT) Metamorphic, (LB) LoopBack, (SL) Sails, (DJ) Django REST, (D3) Django REST 3.4, and (EV) Eve.

Benchmarks Statistics

ENT. TYPE	FRAMEWORK	$\bar{x}$	σ	*min*	*p*25	$\widetilde{x}$	*p*75	*max*
Simple	MA	0029.23	0008.59	0009.04	0023.19	0027.35	0032.31	0100.23
	MT	0096.40	0007.28	0011.27	0092.25	0095.79	0099.82	0146.42
	LB	0057.09	0005.67	0020.91	0054.22	0055.70	0056.94	0093.88
	SL	0043.44	0006.43	0015.44	0039.26	0041.75	0043.03	0075.58
	DJ	0159.91	0054.77	0060.25	0121.85	0157.98	0195.33	0431.67
	D3	0160.49	0052.20	0060.38	0124.36	0160.73	0191.27	0425.68
	EV	0083.47	0005.24	0010.04	0080.00	0082.32	0085.88	0106.04
Has Object	MA	0027.04	0009.11	0009.33	0020.12	0025.27	0031.57	0076.89
	MT	0094.73	0009.31	0012.98	0090.51	0095.19	0099.22	0156.28
	LB	0057.30	0005.98	0012.76	0054.30	0055.42	0056.61	0086.32
	SL	0209.80	0020.69	0046.29	0194.89	0199.73	0224.98	0268.82
	DJ	0168.48	0068.15	0061.37	0127.91	0167.91	0205.64	1880.86
	D3	0170.33	0089.13	0063.00	0129.90	0165.76	0204.03	2633.30
	EV	0084.49	0005.89	0011.00	0079.98	0081.91	0089.28	0103.66
Has List	MA	0094.68	0011.61	0050.04	0087.45	0093.32	0099.64	0156.55
	MT	0254.03	0019.16	0025.28	0241.12	0253.26	0265.11	0329.10
	LB	0078.95	0009.13	0016.61	0073.44	0074.92	0081.95	0112.56
	SL	0253.97	0038.64	0077.20	0236.77	0247.62	0267.53	0464.32
	DJ	0708.32	0182.57	0273.88	0588.75	0714.22	0834.51	1267.33
	D3	0693.44	0181.36	0265.46	0566.47	0694.85	0809.82	1339.59
	EV	0093.13	0006.64	0010.77	0087.87	0090.80	0098.61	0117.99

Table A.2: Benchmark statistics for the GetAll operation including means, standard deviations, minimum values, 25^{th} percentiles, medians, 75^{th} percentiles, and maximum values. Framework implementations are identified as follows: (MA) Metamorphic Async, (MT) Metamorphic, (LB) LoopBack, (SL) Sails, (DJ) Django REST, (D3) Django REST 3.4, and (EV) Eve.

Benchmarks Statistics

ENT. TYPE	FRAMEWORK	$\bar{x}$	σ	*min*	*p*25	$\widetilde{x}$	*p*75	*max*
Simple	MA	019.37	008.16	005.55	013.90	017.74	023.20	088.50
	MT	065.34	007.16	008.65	062.68	065.54	068.25	133.00
	LB	014.13	003.64	005.53	011.24	012.94	016.44	039.57
	SL	019.93	004.53	007.51	017.07	017.96	021.96	049.59
	DJ	106.40	034.75	036.12	079.76	103.85	132.05	253.03
	D3	106.84	034.16	040.06	084.45	107.99	132.31	227.05
	EV	025.05	002.84	004.98	023.79	024.15	024.75	040.67
Has Object	MA	019.27	008.28	005.41	014.09	018.06	023.20	114.55
	MT	066.18	008.44	009.98	063.49	066.48	069.16	182.91
	LB	013.67	003.43	005.27	011.15	012.25	016.23	032.58
	SL	024.26	005.82	009.18	020.92	022.51	026.35	065.04
	DJ	107.84	035.01	040.59	081.44	104.92	133.67	263.88
	D3	108.68	034.59	039.27	083.92	107.40	131.40	246.62
	EV	025.22	002.74	005.67	024.09	024.63	025.07	039.92
Has List	MA	020.40	008.50	006.72	015.40	019.64	023.97	109.98
	MT	070.86	006.92	010.84	068.03	070.78	073.40	147.10
	LB	014.05	003.57	006.16	011.40	012.83	016.85	034.55
	SL	042.01	008.78	022.06	035.87	041.52	047.01	101.17
	DJ	113.47	037.67	043.48	083.32	109.80	138.08	265.53
	D3	113.44	044.14	041.64	089.49	110.91	134.61	1929.89
	EV	025.25	002.84	006.99	023.90	024.34	025.04	041.20

Table A.3: Benchmark statistics for the Get operation including means, standard deviations, minimum values, 25^{th} percentiles, medians, 75^{th} percentiles, and maximum values. Framework implementations are identified as follows: (MA) Metamorphic Async, (MT) Metamorphic, (LB) LoopBack, (SL) Sails, (DJ) Django REST, (D3) Django REST 3.4, and (EV) Eve.

ENT. TYPE	FRAMEWORK	$\bar{x}$	σ	*min*	*p*25	$\widetilde{x}$	*p*75	*max*
	MA	061.20	013.79	012.92	057.46	059.21	063.81	218.71
	MT	173.78	026.20	028.30	159.96	166.57	168.04	451.65
	LB	061.44	014.53	015.67	057.41	059.19	065.04	187.20
Simple	SL	095.62	014.65	066.97	085.95	094.42	104.04	233.36
	DJ	114.48	033.68	046.07	090.69	112.67	133.05	280.81
	D3	114.66	032.09	048.96	092.61	113.64	135.13	378.68
	EV	039.10	002.68	005.85	037.89	038.56	039.63	058.81
	MA	061.88	016.04	014.23	057.57	059.80	064.58	280.64
	MT	175.42	026.42	021.81	166.09	166.81	199.27	500.47
	LB	062.12	016.87	014.08	057.39	059.49	065.33	238.17
Has Object	SL	097.72	016.60	068.48	086.40	095.66	105.91	251.93
	DJ	121.15	036.23	049.62	095.12	120.26	145.38	303.40
	D3	121.33	034.55	048.19	097.36	118.58	142.32	270.94
	EV	048.35	003.53	007.10	045.99	047.02	051.33	064.91
	MA	061.60	011.42	047.28	056.83	058.74	061.30	128.39
	MT	362.30	047.06	041.14	333.01	334.45	384.34	541.68
	LB	062.08	016.35	013.07	057.59	059.41	065.82	216.35
Has List	SL	2975.13	337.66	1150.71	2834.94	3004.09	3194.69	4636.68
	DJ	155.11	081.13	067.06	122.41	147.66	176.73	1173.58
	D3	150.71	038.05	066.33	120.70	146.91	174.87	327.72
	EV	062.72	003.59	011.12	060.05	062.14	064.73	078.60

Table A.4: Benchmark statistics for the Replace operation including means, standard deviations, minimum values, 25^{th} percentiles, medians, 75^{th} percentiles, and maximum values. Framework implementations are identified as follows: (MA) Metamorphic Async, (MT) Metamorphic, (LB) LoopBack, (SL) Sails, (DJ) Django REST, (D3) Django REST 3.4, and (EV) Eve.

ENT. TYPE	FRAMEWORK	$\bar{x}$	σ	*min*	*p*25	$\widetilde{x}$	*p*75	*max*
Simple	MA	030.88	018.77	012.43	023.30	026.04	029.77	171.80
	MT	173.66	023.71	021.06	164.65	166.56	167.97	325.00
	LB	022.63	008.41	010.99	017.91	021.13	024.60	087.04
	SL	091.03	017.74	063.93	078.77	088.36	098.96	191.86
	DJ	117.60	036.75	048.91	090.73	113.15	139.41	283.72
	D3	118.48	034.26	048.93	091.49	113.41	138.55	296.70
	EV	029.27	002.91	006.90	028.06	028.37	029.39	045.93
Has Object	MA	027.39	011.86	013.79	022.17	024.93	027.97	110.51
	MT	166.31	025.16	015.99	151.68	159.77	168.07	286.32
	LB	023.08	008.69	009.81	018.24	021.56	025.39	087.75
	SL	095.44	017.71	062.47	084.33	094.43	104.46	183.04
	DJ	117.10	036.57	047.66	089.88	112.12	139.46	282.25
	D3	116.89	033.72	047.47	092.12	114.56	136.25	331.02
	EV	029.28	002.22	004.65	028.61	028.91	029.64	040.47
Has List	MA	028.07	011.18	014.00	023.56	025.75	029.07	109.43
	MT	174.85	026.38	017.37	159.14	166.60	183.44	434.25
	LB	023.37	011.17	008.93	018.27	021.76	025.24	170.09
	SL	124.09	022.16	078.79	108.96	119.38	133.99	232.83
	DJ	113.35	045.88	046.75	086.54	107.33	130.78	2249.29
	D3	113.18	038.75	046.07	088.95	109.71	136.39	1639.16
	EV	029.59	002.94	004.27	028.26	028.68	030.34	046.48

Table A.5: Benchmark statistics for the Delete operation including means, standard deviations, minimum values, 25^{th} percentiles, medians, 75^{th} percentiles, and maximum values. Framework implementations are identified as follows: (MA) Metamorphic Async, (MT) Metamorphic, (LB) LoopBack, (SL) Sails, (DJ) Django REST, (D3) Django REST 3.4, and (EV) Eve.

Appendix B

Problem Guide

Below is a printed version of the HTML document provided to the quasi-experiment subjects, which contains a description of the problem, the requested tasks, and information on running the tests. More about the experiment can be seen in Chapter 6.

Introduction

This document is part of a master thesis in Informatics and Computer Engineering entitled "Exploring the Scala Macro System for Compile Time Model-Based Generation of Statically Type-Safe REST Services" (http://paginas.fe.up.pt/~ei10038/dissert).

One of the results of this thesis is a framework denominated **Metamorphic**, that based on a **model** generates **REST** services that implement **CRUD** operations.

In order to validate it's relevance and the established goals, the framework will be compared against another model-based framework, the **Django REST** framework. This will be achieved by implementing a **simplified scenario** of a REST API for an online shop.

This document also contains:

- A quick guide (django/) to learn how to use the **Django REST framework**.
- A quick guide (metamorphic/) to learn how to use the **Metamorphic** framework.

How to start

You can **create** and access a development **session** by running:

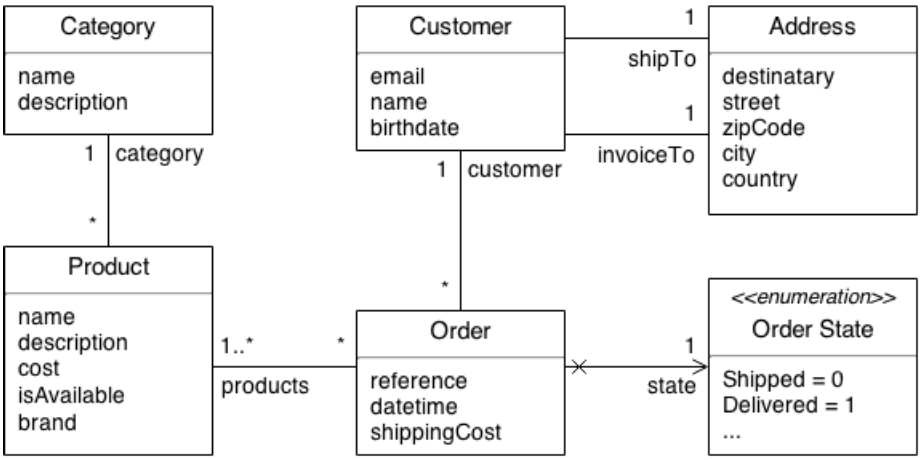
```
./init <session-name>
cd <session-name>
```

This will generate three folders:

- /**django-rest** that contains an empty Django REST project
- /**metamorphic** that contains an empty Metamorphic project
- /**tests** that contains the API tests.

Model

The online shop model as presented in the UML diagram below is composed by **5 entities** (Address, Customer, Category, Product, and Order).



The schema to use by the services for each entity is described below. This includes the types of each property and the **id property**.

Address

```
id: Int,
destinatary: String,
street: String,
zipCode: String,
city: String,
country: String
```

Customer

```
id: Int,
email: String,
name: String,
birthdate: Date,
shipTo: Int,
invoiceTo: Int
```

Category

```
id: Int,
name: String,
description: String
```

Product

```
id: Int,
name: String,
description: String,
cost: Float,
isAvailable: Boolean,
brand: String,
category: Int
```

Order

```
id: Int,
reference: String,
datetime: DateTime,
shippingCost: Float,
state: Int,
products: Int[],
customer: Int
```

Tasks

Follow this set of tasks for each of the frameworks. A set of tests to the expected result can be found in your session folder and used has specified in here.

Task 1: Modeling

Create the necessary classes that represent the described model.

Task 2: Create services

Implement the REST services for each of the entities as described in the table.

Entity	Allowed Operations	Base path
Address	Create, List, Read, Update, Delete	/addresses
Customer	Create, Read, Update	/customers
Category	Create, List, Delete	/categories
Product	Create, List, Read, Update, Delete	/products

Entity	Allowed Operations	Base path
Order	Create, Read, Update	/orders

Task 3: Customization

Reusing the previous services, implement the following constraints:

1. When reading a list of products return only the available products.
2. On creation, an order must have at least one product.
    - If true use the default behavior.
    - Otherwise a BadRequest(400) response must be sent with the message `'Order must have at least one product'`.

Run the tests

The tests are written using frisby.js (http://frisbyjs.com/) that runs Jasmine (http://jasmine.github.io/) for Node.js.

To **run** the **tests** use:

```
jasmine-node ./tests/integration
```

To run the tests without stack trace on test failure use:

```
jasmine-node ./tests/integration --noStack
```

Appendix C

Baseline Guide

Below is a printed version of the HTML document provided to the quasi-experiment subjects under the baseline treatment. This has a brief explanation of Python syntax using examples, a small example on how to implement model-driven API with Django REST, and instructions for running the API. More about the experiment can be seen in Chapter 6.

Python (by example)

Some **basic python syntax** is presented below using examples.

More information can be found here (https://docs.python.org/2/tutorial/index.html).

If statement

```
x = ...

if x < 0:
    x = 0
    print 'Negative changed to zero'
elif x == 0:
    print 'Zero'
elif x == 1:
    print 'Single'
else:
    print 'More'
```

For statement

```
words = ['cat', 'window', 'defenestrate']
for w in words:
    print w, len(w)
```

Function and while statement

```
def fib(n):
    a, b = 0, 1
    while a < n:
        print a,
        a, b = b, a + b
```

Classes and inheritance

```
class Clock:
    ...

class Calendar:
    ...

class CalendarClock(Clock, Calendar):
    ...
```

Boolean values and operators

Operations and built-in functions that have a Boolean result always return `0` or `False` for false and `1` or `True` for true.

Operation	Result
x or y	if x is false, then y, else x
x and y	if x is false, then x, else y
not x	if x is false, then True, else False

Django REST

In this section are described the **key concepts** for implementing model-based REST services with Django REST.

Models

Entities can be defined using the `models` package of Django (https://docs.djangoproject.com/en/1.8/topics/db/models/). Each entity is defined by extending `django.db.models.Model` which may define `Field` properties.

These are usually implemented in file `models.py`. The example below ilustrates possible field uses. The `Field` reference can be found here (https://docs.djangoproject.com/en/1.8/ref/models/fields/).

```
from django.db import models

class Author(models.Model):
    birthdate = models.DateField()
    ...

class Category(models.Model):
    ...

class Book(models.Model):
    name = models.TextField()
    price = models.FloatField()
    nrCopies = models.IntegerField()
    isAvailable = models.BooleanField(default = True)
    created = models.DateTimeField()
    author = models.ForeignKey(Author, related_name = 'author')
    categories = models.ManyToManyField(Category)
```

Serializers

The framework can handle generic requests and as so responses are not built from entities but using serializers. As illustrated below for defining a serializer one can extend a `ModelSerializer` and indicate the entity in the field `model` of the inner class `Meta`.

Serializers also allow custom validation of entities by implementing the `validate` method that if raises some `ValidationError` will cause the service to answer with a **BadRequest (400)**.

These are usually implemented in file `serializers.py`. More information can be found here (http://www.django-rest-framework.org/api-guide/serializers/).

```
from rest_framework import serializers

from shop.rest import models

class AuthorSerializer(serializers.ModelSerializer):
    class Meta:
        model = models.Author

class CategorySerializer(serializers.ModelSerializer):
    class Meta:
        model = models.Category

class BookSerializer(serializers.ModelSerializer):
    class Meta:
        model = models.Book

    def validate(self, data):
        if data['nrCopies'] <= 0:
            raise serializers.ValidationError("A book must have a positive number
of copies.")
        return data
```

Views

The handling of requests is done by instances of `View` that implement the HTTP methods. A `ViewSet` implements these methods by defining higher abstractions: `create`, `list`, `retrieve`, `update`, and `destroy`.

The class `ModelViewSet` implements all of these abstractions using the properties `queryset` and `serializer_class` as illustrated by the `AuthorViewSet` and the `CategoryViewSet`. For implementing only some of the abstractions `mixins` must be used together with `GenericViewSet` as illustrates `BookViewSet`. These may be:

- CreateModelMixin
- ListModelMixin
- RetrieveModelMixin
- UpdateModelMixin
- DestroyModelMixin

These are usually implemented in file `views.py`. More information can be found here (http://www.django-rest-framework.org/api-guide/viewsets/).

```
from django.shortcuts import render
from rest_framework import mixins, viewsets
from rest_framework.response import Response

from shop.rest import models
from shop.rest import serializers

class AuthorViewSet(viewsets.ModelViewSet):
    queryset = models.Author.objects.all()
    serializer_class = serializers.AuthorSerializer

class CategoryViewSet(viewsets.ModelViewSet):
    queryset = models.Category.objects.all()
    serializer_class = serializers.CategorySerializer

class BookViewSet(mixins.CreateModelMixin,
                  viewsets.GenericViewSet):
    queryset = models.Book.objects.all()
    serializer_class = serializers.BookSerializer

    def list(self, request):
        cheap = models.Book.objects.filter(price < 10).all()
        serializer = self.serializer_class(cheap, many = True)
        return Response(serializer.data)
```

URLs (Routers)

The link between views and URLs is made by `urlpatterns` that can be constructed with the help of a router. These are usually implemented in file `urls.py`.

```
from django.conf.urls import patterns, include, url
from rest_framework import routers

from rest import views

router = routers.DefaultRouter()
router.register(r'authors', views.AuthorViewSet)
router.register(r'categories', views.CategoryViewSet)
router.register(r'books', views.BookViewSet)

urlpatterns = patterns('',
    url(r'^', include(router.urls))
)
```

How to start

Project Structure

Your project structure should look like this:

```
shop/
  rest/
    admin.py
    __init__.py
    models.py
    serializers.py
    tests.py
    views.py
  __init__.py
  settings.py
  urls.py
  wsgi.py
manage.py
```

If it matches you are ready to implement the **model**, **serializers**, **views** and **routers**.

Running the API

Before starting the server, **create** the database:

```
cd django-rest
python manage.py migrate
```

A SQLite database will be created in file `db.sqlite3`. After changing the model this file should be **deleted** and previous command rerun.

```
rm db.sqlite3
```

To **start** the server just run:

```
python manage.py runserver 8080
```

Appendix D

Experimental Guide

Below is a printed version of the HTML document provided to the quasi-experiment subjects under the experimental treatment. This has a brief explanation of Scala syntax using examples, a small example on how to implement model-driven API with Metamorphic, and instructions for running the API. Doesn't contain the full documentation as subjects could follow links for the full reference documentation. More about the experiment can be seen in Chapter 6.

Scala (by example)

Scala runs in the JVM and in some aspects has a syntax similar with Java. Anyway some **basic scala syntax** is presented below using examples.

More information can be found here (http://www.scala-lang.org/files/archive/spec/2.11/).

Variables

```
var a = 0 // mutable (can be changed)
val b = 2 // immutable (can't be changed)
```

If, for, and while statements

```
if (foo) bar else baz

for (i <- 0 to 10) { ... }

while (true) { println("Hello, World!") }
```

Functions

The **return value** is the **last expression** to be evaluated, in this case `sum`.

```
def addInt(a: Int, b: Int): Int = {
  val sum: Int = a + b
  sum
}
```

Classes and inheritance

```
class Base {
  def value(x: Int): Int = 2 * x
}

class Foo extends Base {
  def value(x: Int): Int = super.value(x) / 4
}
```

Objects

Objects can be treated as **variables** but may implement functionality.

```
object EmptySet extends IntSet {
  def contains(x: Int): Boolean = false
  def incl(x: Int): IntSet = new NonEmptySet(x, EmptySet, EmptySet)
}

EmptySet.contains(0)
```

Metamorphic

In this section are described the **key concepts** for implementing model-based REST services with Metamorphic.

Entities

Entities are identified by using the annotation `@entity` in a class. Fields can be indicated by using definitions that return a `metamorphic.dsl.Field`. All possible types of fields are represented below. More insight can be found here (http://176.111.107.16/api/#metamorphic.dsl.Field).

```
@entity class Author {
  def birthdate = DateField()
  ...
}

@entity class Category {
  ...
}

@entity class Book {
  def name = StringField()
  def price = DoubleField()
  def nrCopies = IntegerField()
  def isAvailable = BooleanField()
  def created = DateTimeField()
  def author = ObjectField(Author)
  def categories = ListField(Category)
}
```

Services

A `Service` represents a set of HTTP request handlers. `EntityService[T]` may implement some handlers providing the following operations: `GetAll`, `Create`, `Get`, `Replace`, and `Delete`. The services will be available at `/<entity-plural>` path, `/books` in the example below.

If defined, the variable `operations` contains the operations that should be implemented with a **default behavior**. If not defined, the globally defined set of operations will be used. In this case, three operations are implemented.

All 5 operations can be **customized** by overriding the signature defined in here (http://176.111.107.16/api/#metamorphic.dsl.EntityService). These return a `Response` which contains on the of the `StatusCode` defined in here (http://176.111.107.16/api/#metamorphic.dsl.StatusCode). This customizations may use a `repository` to retrieve data as defined in here (http://176.111.107.16/api/#metamorphic.dsl.Repository).

```
class BookService extends EntityService[Book] {

  val operations = List(Replace)

  def getAll = {
    val result = repository.getAll.filter(book => book.price < 10)
    Response(result, Ok)
  }

  def create(book: Book) = {
    if (book.nrCopies <= 0)
      Response("A book must have a positive number of copies.", BadRequest)
    else
      super.create(order)
  }
}
```

App

An application is an object with a `@app` annotation. The object may contain entities and services.

If defined, the variable `operations` contains the operations that each entity may implement with a **default behavior**. As seen before the value can be override by services. If not defined, **all operations** are considered.

```
@app
object App {

  ... // entities
  ... // services

  val operations = List(GetAll, Get, Create)
}
```

How to start

Project Structure

Your project structure should look like this:

```
conf/
  application.conf
project/
  build.properties
  Build.scala
src/
  main/
    scala/
      App.scala // implementation
```

If it matches you are ready to implement the **entities** and **services**.

Running the API

To **start** the server just run:

```
cd metamorphic
sbt run
```

This command will **load** the build settings, **compile**, and **run** the project.

A SQLite database will be created in file `file.db`. After changing the model this file should be **deleted**:

```
rm file.db
```

Appendix E

Pre-Test Questionnaire

Below is a printed version of the HTML form used to collect subjects answers before starting the test. More about the experiment can be seen in Chapter 6.

Pre-test Questionnaire

Thank you for your interest in this experiment.
Please fill this form to start.

* Required

Personal Information

Your education and grade are requested by statistical reasons, but you are still free not to provide them.

1. **Identifier** *
   Already provided

   .....

2. **Education**
   Mark only one oval.

   - ◯ 3rd year
   - ◯ 4th year
   - ◯ 5th year
   - ◯ Doctoral degree
   - ◯ Other: .....

3. **Grade (average)**

   .....

Background

This information helps characterize the subjects of the experiment and improves it's validity.

Answer each question as follows:
1 - Strongly Disagree
2 - Somewhat Disagree
3 - Neither Agree nor Disagree
4 - Somewhat Agree
5 - Strongly Agree

4. **B1. I have considerable experience developing REST APIs.** *
   Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

5. **B2. I have considerable experience with Python.** *
Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

6. **B3. I have considerable experience with the Django REST framework.** *
Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

7. **B4. I have considerable experience with Scala.** *
Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

8. **B5. I have considerable experience with command-line terminals.** *
Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

9. **B6. I have considerable experience analyzing software documentation.** *
Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

10. **B7. I have considerable experience with test-driven development.** *
Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

Appendix F

Post-Test Questionnaire

Below is a printed version of the HTML form used to collect subjects answers after they have finished the tests. By logistics reasons the questions preceded by *OS*, *DP*, and *FG* in section 6.3 are in this document preceded by *SM*, *PM*, and *GM*, respectively, when referring to the Metamorphic framework, and preceded by *SD*, *PD*, and *GD*, respectively, when referring to the baseline framework. More about the experiment can be seen in Chapter 6.

Post-test Questionnaire

Thank you for participating in this experiment.
You may now take a deep breath as this will only take you 5 more minutes.

Each question relates to issues regarding your perception about the experiment. The questionnaire is divide into sections with questions. The sections in pages 2 and 3 are the same but applying each to a different framework.

Please answer each question as follows:
1 - Strongly Disagree
2 - Somewhat Disagree
3 - Neither Agree nor Disagree
4 - Somewhat Agree
5 - Strongly Agree

* Required

1. **Identifier** *
   Already provided

Problem Guide

2. **PG1. I read the complete guide.** *
   Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

3. **PG2. I completely understood what I read.** *
   Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

External Factors

4. **EF1. I felt disturbed and observed by the use of a screen-cast program.** *
   Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

5. **EF2. I enjoyed programming in the experiment.** *
   Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

6. **EF3. I felt pressured to quickly finish the test.** *
   Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

Metamorphic

Please answer each question as follows:
1 - Strongly Disagree
2 - Somewhat Disagree
3 - Neither Agree nor Disagree
4 - Somewhat Agree
5 - Strongly Agree

Overall Satisfaction

7. **SM1. I found it easy and intuitive to specify a model.** *
   Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

8. **SM2. I found it easy and intuitive to specify default behavior.** *
   Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

9. **SM3. I found it easy and intuitive to specify custom behavior.** *
   Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

10. **SM4. I found it easy to make changes and rapidly test them.** *
Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

11. **SM5. I was able to create the application at least as rapidly as I could normally have created.** *
Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

12. **SM6. I found that the resulting application could be used in production-level environments with minimal or no change.** *
Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

Development Process

13. **PM1. Most of my difficulties were implementing the model.** *
Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

14. **PM2. Most of my difficulties were implementing the default behavior.** *
Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

15. **PM3. Most of my difficulties were implementing the custom behavior.** *
Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

16. **PM4. Most of my difficulties were fixing compile errors.** *
Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

17. **PM5. Most of my difficulties were fixing runtime errors.** *
Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

18. **PM6. Most of my difficulties were understanding failed tests.** *
Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

19. **PM7. Most of my difficulties were related with documentation.** *
Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

20. **PM8. Most of my difficulties were related with the development environment (terminal, text editor, etc).** *
Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

Guide

21. **GM1. I read the complete guide.** *
Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

22. **GM2. I completely understood what I read.** *
Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

Django REST

Please answer each question as follows:
1 - Strongly Disagree
2 - Somewhat Disagree
3 - Neither Agree nor Disagree
4 - Somewhat Agree
5 - Strongly Agree

Overall Satisfaction

23. **SD1. I found it easy and intuitive to specify a model.** *
Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

24. **SD2. I found it easy and intuitive to specify default behavior.** *
Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

25. **SD3. I found it easy and intuitive to specify custom behavior.** *
Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

26. **SD4. I found it easy to make changes and rapidly test them.** *
Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

27. **SD5. I was able to create the application at least as rapidly as I could normally have created.** *
Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

28. **SD6. I found that the resulting application could be used in production-level environments with minimal or no change.** *
Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

Development Process

29. **PD1. Most of my difficulties were implementing the model.** *
Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

30. **PD2. Most of my difficulties were implementing the default behavior.** *
Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

31. **PD3. Most of my difficulties were implementing the custom behavior.** *
Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

32. **PD4. Most of my difficulties were fixing compile errors.** *
Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

33. **PD5. Most of my difficulties were fixing runtime errors.** *
Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

34. **PD6. Most of my difficulties were understanding failed tests.** *
Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

35. **PD7. Most of my difficulties were related with documentation.** *
Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

36. **PD8. Most of my difficulties were related with the development environment (terminal, text editor, etc).** *
Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

Guide

37. **GD1. I read the complete guide.** *
Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

38. **GD2. I completely understood what I read.** *
Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

Please answer each question as follows:
1 - Strongly Disagree
2 - Somewhat Disagree
3 - Neither Agree nor Disagree

4 - Somewhat Agree
5 - Strongly Agree

Future

39. **F1. In the future I would use the Metamorphic framework for rapid prototyping of a REST API.** *
Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

40. **F2. In the future I would use the Metamorphic framework for developing a production-level REST API.** *
Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

Comments

If you wish to leave any further comments, they are welcome.

41. .....
.....
.....
.....
.....

Appendix G

Quasi-Experiment Data

The Mann-Whitney-Wilcoxon tests were done using the *exactRankTests* library for R.

G.1 Subjects Characterization

	Group 1				$\bar{x}$	σ	Group 2				$\bar{x}$	σ	H_1	W	$\rho \neq$	$\rho <$	$\rho >$
Grade	17	18	13	15	15.75	2.217	18	16	18	15	16.75	1.500	$\neq$	5.5	0.571	0.286	0.800

Table G.1: Subjects average grades in Master with corresponding means, standard deviation, probability values and the non-parametric significance of the Mann-Whitney-Wilcoxon test.

G.2 Questionnaires Results

	Group 1				$\bar{x}$	σ	Group 2				$\bar{x}$	σ	H_1	W	$\rho \neq$	$\rho <$	$\rho >$
B1	4	4	3	4	3.75	0.50	4	5	4	4	4.25	0.50	$\neq$	04.5	0.571	0.286	1.000
B2	3	4	1	1	2.25	1.50	1	1	5	1	2.00	2.00	$\neq$	09.0	1.000	0.643	0.500
B3	3	4	1	1	2.25	1.50	1	1	1	1	1.00	0.00	$\neq$	12.0	0.429	1.000	0.214
B4	2	1	1	1	1.25	0.50	1	1	3	1	1.50	1.00	$\neq$	07.5	1.000	0.500	0.786
B5	4	3	3	4	3.50	0.58	3	5	5	2	3.75	1.50	$\neq$	07.0	1.000	0.500	0.714
B6	5	3	3	4	3.75	0.96	4	4	3	4	3.75	0.50	$\neq$	07.5	1.000	0.500	0.671
B7	4	2	3	3	3.00	0.82	3	3	2	3	2.75	0.50	$\neq$	09.5	1.000	0.786	0.500
PG1	2	5	3	5	3.75	1.50	4	5	5	2	4.00	1.41	$\neq$	07.5	1.000	0.500	0.671
PG2	4	5	4	5	4.50	0.58	5	4	5	5	4.75	0.50	$\neq$	06.0	1.000	0.500	0.929
EF1	3	1	2	1	1.75	0.96	1	1	1	2	1.25	0.50	$\neq$	10.5	0.714	0.929	0.357
EF2	4	4	3	5	4.00	0.82	5	5	3	5	4.50	1.00	$\neq$	05.0	0.486	0.243	0.871
EF3	2	2	2	1	1.75	0.50	1	1	2	2	1.50	0.58	$\neq$	10.0	1.000	0.929	0.500
F1	4	4	5	5	4.50	0.58	5	5	5	4	4.75	0.50	$\neq$	06.0	1.000	0.500	0.929
F2	4	4	4	5	4.25	0.50	5	4	4	4	4.25	0.50	$\neq$	08.0	1.000	0.786	0.786

Table G.2: Questionnaires results of round independent questions with corresponding means, standard deviation, probability values and the non-parametric significance of the Mann-Whitney-Wilcoxon test; see section 6.3.

	Experimental				$\bar{x}$	σ	Baseline				$\bar{x}$	σ	H_1	W	$\rho \neq$	$\rho <$	$\rho >$
OS1	5	5	5	4	4.75	0.50	2	4	2	4	3.00	1.15	$>$	15.0	0.086	1.000	0.043
OS2	5	5	5	5	5.00	0.00	2	3	3	4	3.00	0.82	$>$	16.0	0.029	1.000	0.014
OS3	5	5	5	5	5.00	0.00	2	3	2	3	2.50	0.58	$>$	16.0	0.029	1.000	0.014
OS4	5	5	5	3	4.50	1.00	2	5	3	4	3.50	1.29	$>$	12.0	0.371	0.929	0.186
OS5	5	5	4	5	4.75	0.50	2	4	1	3	2.50	1.29	$>$	15.5	0.057	1.000	0.029
OS6	5	4	4	4	4.25	0.50	3	4	3	4	3.50	0.58	$>$	13.0	0.286	1.000	0.143
DP1	1	1	2	2	1.50	0.58	4	2	2	3	2.75	0.96	$<$	02.0	0.171	0.086	1.000
DP2	1	1	1	2	1.25	0.50	3	3	2	2	2.50	0.58	$<$	01.0	0.086	0.043	1.000
DP3	1	1	1	2	1.25	0.50	3	3	2	2	2.50	0.58	$<$	01.0	0.086	0.043	1.000
DP4	5	2	4	4	3.75	1.26	5	3	5	3	4.00	1.15	$>$	07.0	0.829	0.414	0.629
DP5	1	1	1	2	1.25	0.50	5	4	4	4	4.25	0.50	$<$	00.0	0.029	0.014	1.000
DP6	5	1	4	4	3.50	1.73	5	1	4	3	3.25	1.71	$\neq$	09.0	0.971	0.686	0.486
DP7	1	4	1	1	1.75	1.50	3	1	2	2	2.00	0.82	$\neq$	05.5	0.486	0.243	0.814
DP8	1	1	1	3	1.50	1.00	2	1	2	1	1.50	0.58	$\neq$	07.0	1.000	0.500	0.643
FG1	4	5	4	2	3.75	1.26	2	5	4	5	4.00	1.41	$\neq$	06.5	0.914	0.457	0.800
FG2	5	4	5	4	4.50	0.58	3	5	3	5	4.00	1.15	$\neq$	10.0	0.657	0.757	0.329

Table G.3: Post-test questionnaire results in Round 1 with corresponding means, standard deviation, probability values and the non-parametric significance of the Mann-Whitney-Wilcoxon test; see section 6.3.

	Experimental				$\bar{x}$	σ	Baseline				$\bar{x}$	σ	H_1	W	$\rho \neq$	$\rho <$	$\rho >$
OS1	4	5	3	5	4.25	0.96	2	3	5	3	3.25	1.26	$>$	12.0	0.400	0.943	0.200
OS2	4	5	4	5	4.50	0.58	2	3	3	3	2.75	0.50	$>$	16.0	0.029	2.000	0.014
OS3	4	5	4	5	4.50	0.58	2	3	5	3	3.25	1.26	$>$	13.0	0.200	0.943	0.100
OS4	5	2	4	4	3.75	1.26	2	1	5	4	3.00	1.83	$>$	10.0	0.771	0.800	0.386
OS5	5	5	5	5	5.00	0.00	2	1	3	5	2.75	1.71	$>$	14.0	0.143	1.000	0.071
OS6	5	3	4	4	4.00	0.82	4	4	4	3	3.75	0.50	$>$	09.5	1.000	0.786	0.500
DP1	3	1	2	1	1.75	0.96	1	1	1	2	1.25	0.50	$<$	10.5	0.714	0.929	0.357
DP2	2	1	2	1	1.50	0.58	1	1	3	2	1.75	0.96	$<$	07.0	1.000	0.500	0.757
DP3	2	3	2	3	2.50	0.58	1	4	1	4	2.50	1.73	$<$	08.0	1.000	0.629	0.629
DP4	2	3	2	4	2.75	0.96	1	5	1	2	2.25	1.89	$>$	11.0	0.400	0.843	0.200
DP5	2	1	2	1	1.50	0.58	5	5	4	5	4.75	0.50	$<$	00.0	0.029	0.014	1.000
DP6	5	1	2	1	2.25	1.89	5	4	4	5	4.50	0.58	$\neq$	03.0	0.171	0.086	0.957
DP7	2	1	3	1	1.75	0.96	1	3	1	2	1.75	0.96	$\neq$	08.0	1.000	0.671	0.671
DP8	2	1	2	1	1.50	0.58	1	1	1	4	1.75	1.50	$\neq$	09.0	1.000	0.643	0.500
FG1	4	5	3	5	4.25	0.96	4	5	4	2	3.75	1.26	$\neq$	10.0	0.657	0.800	0.329
FG2	4	5	4	5	4.50	0.58	5	4	5	4	4.50	0.58	$\neq$	08.0	1.000	0.757	0.757

Table G.4: Post-test questionnaire results in Round 2 with corresponding means, standard deviation, probability values and the non-parametric significance of the Mann-Whitney-Wilcoxon test; see section 6.3.

G.3 Objective Measurement

Measurement	Experimental				$\bar{x}$	σ	Baseline				$\bar{x}$	σ
# Requests for help	06	01	03	02	03.0	2.16	06	01	06	03	04.0	2.45
# Lines of code	73	63	63	63	65.5	5.00	78	96	97	97	92.0	9.35
# Lines of code (no braces)	48	50	49	50	49.3	0.96						
# Application runs	07	03	07	07	06.0	2.00	04	05	08	14	07.8	4.50
# Tests runs	06	02	04	07	04.8	2.22	08	05	09	14	09.0	3.74
# Non-runtime errors	05	01	04	01	02.8	2.06	00	03	01	02	01.5	1.29
# Runtime errors	00	00	00	00	00.0	0.00	03	02	05	13	05.8	4.99
Task 1 time (min)	12.00	12.50	07.00	14.25	11.44	03.11	31.25	09.75	23.00	16.50	20.13	09.18
Task 2 time (min)	04.25	07.25	17.75	04.00	08.31	06.46	41.75	13.75	11.75	24.50	22.94	13.73
Task 3 time (min)	04.25	07.75	02.75	05.50	05.06	02.12	-	02.50	02.25	07.00	03.92	02.67
Total time (min)	56.25	36.75	43.00	47.25	45.81	08.19	72.00	53.00	77.75	79.50	70.56	12.14

Table G.5: Objective measurements in Round 1 with corresponding means and standard deviation; see section 6.4. The missing value happened as one of the subjects didn't finish task 3.

Measurement	Experimental				$\bar{x}$	σ	Baseline				$\bar{x}$	σ
# Requests for help	02	01	01	04	02.0	1.41	04	00	02	02	02.0	1.63
# Lines of code	63	63	61	63	62.5	1.00	96	89	73	87	86.3	9.64
# Lines of code (no braces)	50	50	48	50	49.5	1.00						
# Application runs	10	09	03	06	07.0	3.16	07	04	06	12	07.3	3.40
# Tests runs	06	01	01	01	02.3	2.50	08	05	06	11	07.5	2.65
# Non-runtime errors	04	06	02	05	04.3	1.71	00	00	01	00	00.3	0.50
# Runtime errors	01	00	00	00	00.3	0.50	05	04	05	08	05.5	1.73
Task 1 time (min)	07.00	06.50	08.50	09.75	07.94	01.48	05.50	10.75	07.00	12.00	08.81	03.06
Task 2 time (min)	15.50	15.50	07.75	04.00	10.69	05.76	15.50	16.75	19.00	17.75	17.25	01.49
Task 3 time (min)	08.00	03.50	05.00	04.75	05.31	01.91	02.50	03.00	04.75	03.50	03.44	00.97
Total time (min)	29.75	34.00	30.50	31.75	31.50	01.86	48.50	43.50	50.50	63.00	51.38	08.29

Table G.6: Objective measurements in Round 2 with corresponding means and standard deviation; see section 6.4.

Appendix H

Experimental Tasks Implementation

```
import metamorphic.dsl._

@app
object App {

  @entity class Address {
    def destinatary = StringField()
    def street = StringField()
    def zipCode = StringField()
    def city = StringField()
    def country = StringField()
  }

  @entity class Customer {
    def email = StringField()
    def name = StringField()
    def birthdate = DateField()
    def shipTo = ObjectField(Address, R.Object)
    def invoiceTo = ObjectField(Address, R.Object)
  }

  @entity class Category {
    def name = StringField()
    def description = StringField()
  }

  @entity class Product {
    def name = StringField()
    def description = StringField()
    def cost = DoubleField()
    def isAvailable = BooleanField()
    def brand = StringField()
    def category = ObjectField(Category)
```

```
34    }
35
36    @entity class Order {
37      def reference = StringField()
38      def datetime = DateTimeField()
39      def shippingCost = DoubleField()
40      def state = IntegerField()
41      def products = ListField(Product)
42      def customer = ObjectField(Customer)
43    }
44
45    @entity class Shop {
46      def opens = DateTimeField()
47      def closes = DateTimeField()
48      def products = ListField(Product)
49    }
50
51    class CustomerService extends EntityService[Customer] {
52      val operations = List(Get, Create, Replace)
53    }
54
55    class CategoryService extends EntityService[Category] {
56      val operations = List(GetAll, Create, Delete)
57    }
58
59    class OrderService extends EntityService[Order] {
60      val operations = List(Get, Replace)
61
62      def create(order: Order) = {
63        if (order.products.length == 0)
64          Response("Order must have at least one product", BadRequest)
65        else
66          super.create(order)
67      }
68    }
69
70    class ProductService extends EntityService[Product] {
71
72      def getAll = {
73        val result = repository.getAll.filter(product => product.isAvailable)
74        Response(result, Ok)
75      }
76    }
77  }
```

Source H.1: Possible implementation of the quasi-experiment tasks using the experimental framework.