Concern-Oriented and Model-Driven Migration of Legacy Java Applications to RESTful Web Services

Bowen Li

A thesis submitted to McGill University in partial
fulfillment of the requirements of the degree of

Masters of Science

School of Computer Science
McGill University
Montréal, Québec, Canada

June 2022

Abstract

With modern advances in internet technologies, many existing *native* software systems have been extended to *web* services to enable the querying and modification of data from anywhere in the globe across the network. REpresentational State Transfer (REST) proposes a standard architectural style for web services and defines a collection of constraints for their behaviour. Nowadays, developers can integrate the REST interface in their software system by incorporating an appropriate framework that implements REST. Popular REST implementation technologies include Spring Boot, Eclipse Jersey, JBoss RESTEasy and Apache CXF.

There is significant technical complexity in the development process of RESTful web services. The typical web service development first involves designing a REST resource tree consisting of Uniform Resource Identifiers (URIs) and HTTP request methods for the designated controller operations. Next, a compatible REST framework is selected to be integrated as a *dependency* to the existing software system. Now in order to accommodate for the requirements of the selected REST technology, the existing build pipeline is updated, and new source code files are implemented correspondingly to the specification of the new *dependency*. Finally, the designated controller classes, operations and applicable parameters are decorated with the defined REST annotations from the framework technology.

In this thesis we propose an alternative approach to building REST applications based on Model-Driven Engineering (MDE), Domain-Specific Modelling Languages (DSMLs) and Concern-Oriented Reuse (CORE). We encapsulate the technical complexity of REST frameworks in a reusable unit called the RESTify concern. With the RESTify concern, the interface to the existing software system is represented at a high level of abstraction with a Reusable Aspect Model (RAM) composed of a UML Class Diagram design model. The design of the REST Resource tree is accomplished with a domain-specific Resource Tree Language (ResTL). The connection

between the REST interface and the application functionality is done by providing mappings from ResTL to RAM, avoiding the tedious placement of REST Java annotations across multiple Java files. Finally, a functional and ready-to-deploy RESTful web service can be generated from the models simply by selecting the concrete REST implementation framework to use.

To implement the RESTify pipeline, this thesis delivers several contributions: 1) an extension of the CORE metamodel to support reuse and selection of features of the RESTify concern, 2) a graphical user editor for the ResTL language as well as a 3) a split-view capable of displaying ResTL and RAM design models simultaneously and establish coherent mappings between them, 4) a refactoring and modularization of the previously monolithic code generation transformations in order to maximize transformation reuse among the different REST implementations, and finally 5) an algorithm for composing Maven build configuration files.

Abrégé

En vertu des percées modernes dans les technologies internet, de nombreux systèmes logiciels *natifs* existants ont été élargis aux services *web* afin de permettre la consultation et la modification de données depuis n'importe où au monde via un réseau. REpresentational State Transfer (REST) propose un style architectural standard pour les services web et définit un ensemble de contraintes pour leur fonctionnement. De nos jours, les développeurs peuvent intégrer l'interface REST dans leur système logiciel en incorporant un framework approprié permettant de l'implémenter. Les technologies les plus populaires permettant l'implémentation de REST sont Spring Boot, Eclipse Jersey, JBoss RESTEasy et Apache CXF.

Le processus de développement des services Web RESTful est d'une grande complexité technique. En effet, le développement classique d'un service web implique tout d'abord la conception d'un arbre de ressources REST regroupant des identificateurs de ressources uniformes (URI) et des méthodes de requête HTTP pour les opérations du contrôleur désigné. Ensuite, un framework REST compatible est retenu pour être intégré en tant que dépendance au système logiciel existant. Afin de répondre aux exigences de la technologie REST choisie, le pipeline de build existant est mis à jour et de nouveaux fichiers de code source sont implémentés conformément à la spécification de la nouvelle dépendance. Pour finir, les classes de contrôleur désignées, les opérations et les paramètres applicables sont agrémentés des annotations REST définies de la technologie du framework.

Dans le cadre de cette thèse, nous proposons une approche alternative au développement d'applications REST basée sur une ingénierie dirigée par modèles (MDE), des langages de modélisation spécifiques au domaine (DSML) et une réutilisation axée sur les préoccupations (CORE). Nous intégrons la complexité technique des frameworks REST dans une unité réutilisable appelée RESTify préoccupation. Avec le RESTify préoccupation, l'interface du système logiciel existant

est représentée à un haut niveau d'abstraction avec un Reusable Aspect Model (RAM) constitué d'un modèle de conception de diagramme de classe UML. La conception de l'arbre de ressources REST est réalisée avec un langage d'arbre de ressources (ResTL) spécifique au domaine. La connexion entre l'interface REST et la fonctionnalité de l'application se fait en prévoyant des mappings entre ResTL et RAM, évitant ainsi le placement pénible des annotations Java REST dans plusieurs fichiers Java. Pour finir, un service web RESTful fonctionnel et prêt à être déployé peut être généré depuis les modèles en sélectionnant simplement le framework d'implémentation REST spécifique à exploiter.

Afin d'implémenter le pipeline RESTify, cette thèse propose plusieurs contributions : 1) une extension du métamodèle CORE afin de supporter la réutilisation et la sélection des caractéristiques du RESTify préoccupation, 2) un éditeur graphique pour le langage ResTL ainsi qu'un 3) une vue partagée permettant d'afficher simultanément les modèles de conception ResTL et RAM ainsi que d'établir des mappings pertinents entre ces derniers, 4) un refactorisation et une modularisation des transformations de génération de code précédemment monolithiques afin de maximiser la réutilisation des transformations au sein des différentes implémentations REST, et pour finir 5) un algorithme pour la compilation des fichiers de configuration de construction Maven.

Acknowledgements

I would like to thank my supervisor, Jörg, for the opportunity to work on such a fascinating project and his tremendous support, guidance, and encouragement throughout the entire journey. This thesis would not be possible without him. I would also like to thank my mentor, Max, for inviting me to partake in his research, patiently answering my questions and constantly teaching me about new technologies. He is the greatest mentor I could ever ask for. Throughout the years, I have learned so much from you two, as a researcher, software developer and as a person. Thank you.

I express my deepest gratitude to my family for their unconditional love and encouragement. To my parents, thank you for being here for me during every moment in my life. I will never forget the times when we have celebrated my accomplishments and embraced my failures. To my grandfather, Liangzheng Xia, and grandmother, Guifen Du, thank you for always being involved in my education and career. I owe my success to your invaluable guidance. To my grandmother, Xiulan Yang, thank you for always encouraging and supporting me throughout my life. I appreciate your regular phone calls to check in on my well-being and ask about my current ventures. To my uncle Hui, thank you for helping me look at my predicaments from a different perspective.

I thank my good friend, Suko, for assisting me with French for the Abrégé chapter. I thank my friends and colleagues for their companionship and encouragement throughout my studies. These years have been much more enjoyable with them around.

Thank you everyone.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

Since the advent of the internet, web applications are rapidly growing in popularity. In particular, e-commerce platforms have been overwhelming traditional brick and mortar businesses, due to their accessibility and convenience. The coronavirus (COVID-19) pandemic continues to play a significant influence on e-commerce and online consumer behaviour around the world [Cop22]. As millions of people stayed home in early 2020 to contain the spread of the virus, digital channels have become the most popular alternative to crowded stores and in-person shopping [Cop22]. In June 2020, global retail e-commerce traffic stood at a record 22 billion monthly visits, with demand being exceptionally high for everyday items such as groceries, clothing, but also retail tech items [Cop22].

Web applications require a web service to act as a server, listening and responding to requests from clients over the network. The most dominant approaches for web services are SOAP and REST. Simple Object Access Protocol (SOAP) is a messaging protocol specification for the exchange of structured information for web services. SOAP uses extensible markup language (XML) as its messaging format and relies on application layer protocols. On the other hand, REpresentational State Transfer (REST) [Fie00] proves a standard software architectural style for web service interfaces and defines a set of constraints for how they should behave. RESTful web services define a combination of uniform resource identifiers (URIs) and HTTP request methods (GET, PUT, POST, DELETE) to expose the defined service endpoints to the internet. Throughout this thesis, we will be highlighting the development process of RESTful web services, due to their simplicity, flexible selection of message data formats, and superior performance.

The standard procedure of transforming an existing software system to a RESTful web service involves defining a REST resource tree, choosing an implementation technology, updating build files or scripts to accommodate for the selected framework, providing additional files as required by the framework, and annotating controller classes and operations with appropriate URIs, HTTP request methods, parameter-mappings, and payload format and encodings. When the above devel-

opment procedures are complete, the resulting RESTful web service can be deployed by simply running the build files or scripts. We elaborate on the technical complexity of each of the steps illustrated above and illustrate some of the challenges of traditional RESTful web service development.

The first and commonly overlooked step in the development process is the definition of the underlying REST resource tree. Developers often devise REST resource endpoints spontaneously and focus on the actual development of the RESTful web service system. This can result in a poorly defined resource architecture, that does not follow the standard REST naming and behaviour conventions. REST standardizes web service interfaces, and familiar users can infer the business logic of an endpoint simply from the URI and HTTP verb. By properly following the REST conventions and meticulously designing the endpoint architecture in the RESTful web service system, developers can observe a significant improvement in the design, reusability, and extensibility of the software.

REST is an established architectural style and there are numerous implementation technologies that offer a framework to develop a RESTful web service. For the Java programming language, a commonly used specification for supporting RESTful web services is Jakarta RESTful Web Services (JAX-RS) [Gui]. Popular implementations of JAX-RS include Eclipse Jersey [Fou21], JBoss RESTEasy [Hat], and Apache CXF [Fou]. Alternatively for Java, Spring Boot [Edu] provides a framework for stand-alone, production grade Spring-applications that complies and implements the REST architecture. There are plentiful REST implementations for other programming and scripting languages as well, including Django REST for Python, Express.js for Node.js etc. It can be difficult to select the most applicable technology for the software system, given the significant options that exist and the various trade-offs each of them offers. After the selection of a REST implementation framework, developers need to read the technical documentation extensively to understand how to incorporate the framework into their system.

Software systems typically define build scripts or files to build and deploy the resulting application. Developers need to update the existing build pipeline, typically using a build automation tool, to incorporate the selected REST implementation framework by adding the dependencies and reconfiguring the build lifecycle. Build automation tools can help simplify the complex development of manual build scripts, and help teams work more efficiently and with more flexibility. Popular build automation tools include Gradle and Apache Maven. Gradle introduces a Groovy based Domain Specific Language (DSL) for project configuration and is widely used due to its high performance. On the other hand, Apache Maven [PZL] focuses on standardizing the development of software in a specific layout and uses XML for structuring the application. Developers need to include a specific version of the selected framework that is compatible with the rest of the

software system as a dependency to the project configuration files. In addition, they need to update the deployment section of the project configuration files to deploy the software to an application server. Commonly used Java application servers include Jetty and Tomcat. There are often different methods to define the build life-cycle, and it can be difficult to select the most applicable approach and update the current build to accommodate for the new framework.

Most implementations of the REST architecture style require additional files to be defined. For Java, this can include launcher classes or application configuration classes that specify the REST controller classes. Developers are required to study the technical documentation thoroughly to ensure that all of the required additional files are functional and present during deployment.

The most technically and conceptually challenging task is decorating the software system with appropriate annotations. The annotations can specify the URI, HTTP verb, parameter-mappings, and payload format and encodings. The conceptualization of the REST resource tree is a prerequisite for this step. A well-defined resource tree that complies with REST conventions can simplify this procedure. The URI and HTTP request method annotations specify the resulting address of the REST endpoint. Parameter-mappings can help define query parameters, which assigns values from a part of the URI to specified parameters. REST supports a variety of payload formats, but JavaScript Object Notation (JSON) and XML are the most common. Payload format and encoding annotations are necessary to ensure that the exchange of payloads between the web service and the users are in the proper format. Different REST implementations define the annotations differently, developers need to ensure they are using the correct annotations for the selected REST implementation.

The deployment step can be quite simple if the above development procedures have been completed correctly. A functional RESTful web service implementation can be run within a few simple commands from the build configuration to compile and package the software system and deploy it on the internet.

After the development transformation of an existing software system to a functional RESTful web service, changing implementation technologies can be quite demanding and troublesome. This can happen when a different implementation technology is required due to the evolution of the software system and user specifications, in which the developers need to migrate to a different technology. The entire development process illustrated above is required to be executed again with a different framework, which can be technically challenging and vulnerable to software defects.

1.1 Contributions

This thesis incorporates Concern-Oriented Reuse (CORE) [AKM13] to streamline the development of REST web services for software systems by providing a modelling language, Resource

Tree Language (ResTL) specifically designed for expressing REST interfaces. We establish a reusable concern, namely RESTify, to encapsulate the technological implementation details of REST frameworks. By defining clear modularization of design decisions, our approach offers designs of well-structured architectures, encapsulations of technical complexity, and a simple transformation pipeline. Concretely, this thesis makes the following contributions:

1. We extend the CORE metamodel to allow an application developer to reuse the RESTify concern and select an implementation technology by means of the CORE-provided feature model interface. The RESTify concern supports Spring Boot and implementations of JAX-RS, including Eclipse Jersey, JBoss RESTEasy, and Apache CXF.

2. We develop an interactive graphical user interface for creating and modifying a ResTL model. A ResTL model visualizes the REST architecture as a tree with endpoints defined by its URI and HTTP request methods. This interface supports the ResTL model through various gestures and buttons in CORE.

3. We implement a split view that can display a REST tree model and a class diagram model, as defined in Unified Modeling Language (UML) simultaneously and create mappings between the two models. Additional efforts are applied to make the split view generic in which it can view, edit, and add or remove mappings of two arbitrary models concurrently. We also define a mapping validator that ensures the validity of the mappings when a mapping is added or removed, either model is edited, and when the transformation pipeline is executed.

4. We modularize the existing code generation transformation pipeline in CORE for Spring Boot and provide support for annotation and code generation transformations with JAX-RS technologies. We redesign the transformation pipeline for REST implementations such that when the technologies share common implementation details, some of the transformations can be reused. Newly supported JAX-RS technologies include Eclipse Jersey, JBoss RESTEasy, and Apache CXF.

5. We propose an algorithm for combining build configuration files for the Apache Maven build automation tool in order to support the modularization of transformations for different REST implementation technologies. Highlights of the algorithm include dynamic merging of *dependencies* and *build plugins* from the input build configuration files.

1.2 Thesis Outline

In the subsequent chapters, we will introduce our RESTify approach which incorporates MDE and CORE technologies to model and automate the development extension of an existing software

system to a RESTful web service with a sophisticated transformation pipeline. We illustrate the new technologies developed in CORE to enable the accessibility, practicality and customizability of the RESTify approach. We then present the implementation specifications for each individual transformation in the RESTify transformation pipeline. Throughout the thesis, we illustrate the example of a BookStore application [Sch20a] to demonstrate the advantages and effectiveness of our method.

Chapter 2 highlights the relevant background knowledge for our proposed method. Chapter 3 summarizes the foundation of RESTify, including the user workflow with CORE, and the underlying transformations. Chapter 4 presents a new method of concern reuse to reuse a concern in directly in another concern via a dummy artefact. Chapter 5 presents a domain-specific language for the REST architecture and its graphical interface in CORE. Chapter 6 presents a new view in CORE to visualize two arbitrary models at the same time and create inter-model mappings. Chapter 7 elaborates on the redesign of the existing transformation pipeline with Spring Boot and addition of new transformations to accommodate for different JAX-RS implementations. Chapter 8 presents a new algorithm for parsing, weaving, and outputting Maven build configuration files, namely pom.xml files, that validates with the Maven pom schema file. Chapter 9 elaborates on alternative approaches for modelling languages for REST and code generation transformations of RESTful web service implementations. Chapter 10 concludes the thesis by summarizing the contributions of our approach, and discussing the potential future extensions of our method.

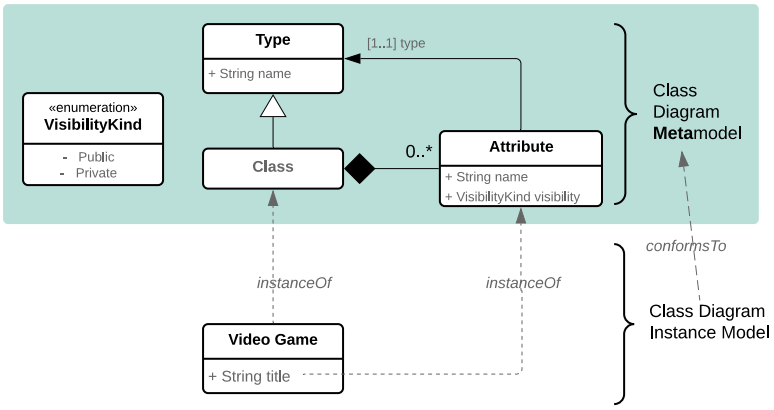
2 Background

2.1 Model-Driven Engineering

Model-Driven Engineering (MDE) is a software engineering methodology that addresses the increasing complexity of modern software systems by the use of technology agnostic models. Recent advances in software languages and platforms during the decent decades have raised the level of software abstractions available to developers with expressive object-oriented languages, and reusable class libraries and application framework platforms to minimize the need to reinvent common and domain-specific middleware services [Sch06]. However, despite these advances, several problems remain, including the growth of platform complexity, and complication of key integration activities, such as system deployment, configuration, and quality assurance. Platform complexity has evolved much faster than the ability of general-purpose languages to mask it, resulting in convoluted platforms with intricate dependencies and subtle side effects. Furthermore, new platforms are regularly introduced where developers need to spend considerable effort manually porting application code to different platforms or newer versions of the same platform. The complication of key integration activities stems from the semantic gap between the design intent and the expression of this intent in the actual implementation. The lack of an integrated view often leads developers to implement suboptimal solutions [Sch06].

MDE technologies address platform complexities and the inability of programming languages to alleviate this complexity and express domain concepts effectively by introducing Domain-Specific Modelling Languages (DSMLs) and transformation engines and generators. DSMLs formalize the application structure, behaviour, and requirements within particular domains with *metamodels*, which defines the relationships among concepts in a domain and precisely specify the key semantics and constraints associated with these domain concepts. Transformation engines and generators analyze aspects of models and synthesize various types of artifacts including source code, simulation inputs, XML deployment descriptions, or alternative model representations. This

Figure 2.1: Example of Metamodelling



transformation process automates the conventional software development process, which is tedious and often error prone. By combining DSMLs and transformation engines, MDE tools impose domain-specific constraints and perform model checking that can detect and prevent many errors early in the life cycle.

In summary, MDE provides an abstraction layer higher than the implementation level and can be incorporated in the software development lifecycle by enabling developers to focus on the behavioural and structural aspects of the software system, as opposed to the implementation concerns.

2.2 Metamodelling

Model-driven engineering introduces the concept of a *metamodel* as a model for describing models, or simply a model of models [KÖ06]. A metamodel elaborates the accepted individual model elements and the relationships between them, such that model instances instantiate upon them. The practice of designing a metamodel is called metamodelling.

Figure 2.1 provides an example of metamodelling and illustrates the distinctions from metamodels and their model instances. The Figure depicts a simple class diagram metamodel, and an instance model that conforms to the metamodel. Within the instance model, the *Video Game* element is an instance of the *Class* model element. The *title* identifier is an instance of the *Attribute* model element with the identifier *name* containing the value “title”, and the identifier *visibility* containing the value “Public”. The *title* identifier is also attached to a *Type* model element with the identifier *name* as “String”.

It is possible to design a model at an even higher modelling level with a *metametamodel*.

A metametamodel is a modelling language for expressing metamodels. Ecore from the Eclipse Modelling Framework (EMF) provides an example of a metametamodel in which metamodels can be realized from its various model elements and relationships.

2.3 CORE

Concern-Oriented REuse (CORE) is a new software development paradigm inspired by the ideas of multi-dimensional separation of concerns and builds on the disciplines of MDE, Software Product Lines (SPL), goal modelling, and advanced modularization techniques offered by aspect-orientation to define flexible software modules that enable broad-scale model-based software reuse [SAKM16].

In CORE, the primary unit of modularization, abstraction, construction, and reasoning is a *concern* [SAKM16]. A concern can include multiple *realization models* to address a specific domain of interest during software development. In order to exploit the maximum reuse potential, a concern typically provides multiple variations to address a particular domain of interest during software development. Additionally, the realization models within a concern can encompass various phases of software development and levels of abstraction. A concern can be constructed by specifying its three interfaces, namely the *variation interface*, *customization interface*, and *usage interface*. We elaborate on the specifications of each of the interfaces below.

- The *variation interface* describes the available variations of the concern and the impact of different variants on high-level stakeholder goals, qualities, and non-functional requirements [AKM13]. The default visualization for a concern and its variation interface is with a *feature model* that specifies individual *features* that the concern offers. The feature model enables the idea of *modularization along features* which decomposes the creation of large and monolithic models of each possible configuration of a concern into various smaller models named *realization models* and links them to features. The variation interface also provides an *impact model* to establish the impact of selecting a feature on non-functional goals and qualities. This is achieved by assigning a numerical value to features for the relevant goals and qualities.

- The *customization interface* increases the potential reusability of concerns by describing realization models as abstract as possible. To elaborate, some model elements in a concern can be *partially* specified and require additional complementation with concrete model elements from the reusing concern during concern reuse. These model elements can be specified in CORE with a *partiality* identifier. The customization interface of a concern thus consists of all of the model elements in the realization models that have the value *public* for the *partiality*

identifier and have to be adapted to the context of the reusing concern to be functional.

- The *usage interface* specifies which model elements from the concern are accessible when it is reused. In particular, each model element in CORE has a *visibility* identifier and by setting it to *public*, the concern designer can expose model elements of the realization model to the outside world. Thus, the usage interface of a reused concern can include a subset of the model elements contained in the concern.

For the RESTify approach, we incorporate CORE to package our different REST implementation strategies in a straightforward and reusable way. We propose a concern to encapsulate the technical complexity of REST frameworks with only the *variation interface*. The RESTify concern is visualized with a *feature model* in Figure 3.2. Each terminal *feature* specifies a singular REST implementation technology. Intermediary *features* can define an additional layer of specification for implemented REST frameworks to inherit. A valid reuse of the RESTify concern includes a selection of only a singular terminal feature and the applicable intermediary features. The RESTify concern is different from traditional CORE concerns in which it only defines the variation interface with a feature model, and does not include any realization models, thus eliminating the need for the customization and usage interfaces.

2.3.1 Concern Reuse Process

After the construction of a concern by a concern designer, concern users and software engineers can import the functionality from that concern by *reuse*. The reuse process of an existing concern is simple and consists of the following steps [SAKM16]:

1. The concern user selects the relevant *features* from the *variation interface* of the reused concern with consideration of the impact on important stakeholder goals and system qualities. Based on the selected configuration, the modelling tool merges the *realization models* from the selected features to yield new models.

2. The concern user adapts the newly generated models to the application context by mapping abstract model elements from the *customization interface* to concrete application-specific model elements in the reusing concern.

3. A software engineer incorporates the functionality provided by the selected concern *features* and exposed in the *usage interface* of the concern within their own application models.

The traditional concern reuse process as articulated above stores the reuse information with a *COREExternalArtefact*, which encapsulates an instance of a metamodel and its modelling language. The envisioned reuse process for the RESTify concern enables concern reuse directly from

a concern, thus eliminating the requirement of the *COREExternalArtefact*. To support this particular concern reuse process, we introduce a *COREReuseArtefact* to serve as a dummy artefact for storing the reuse information. Chapter 4 elaborates on the implementation of the proposed concern reuse process with a *COREReuseArtefact*.

2.3.2 Languages in CORE

In CORE, *languages* provide modelling languages as metamodels for realization models. Previous releases of TouchCORE, the modelling software realization of CORE, support languages by encoding them in a single metamodel. The limitation of this approach is that the evolution of a singular integrated metamodel can yield an overly complex metamodel. Subsequently, TouchCORE now supports various modelling languages as *plug-in languages*.

A plug-in language can be integrated in TouchCORE by providing a *metamodel*, *language actions*, *graphical user interface*, and a *weaver*. The *metamodel* defines the modelling language using ECore from the Eclipse Modelling Framework (EMF). In order to integrate the metamodel for the plug-in language, a separate instance of the CORE metamodel is also required to provide additional information for the concern-oriented reuse of models. The *language actions* are implemented as Java controller classes to execute EMF commands for creating and updating models of the plug-in language. The *graphical user interface* provides an editor in for visualizing models of the plug-in language and executing language actions from the supported controls. The *weaver* is an implementation of a model-to-model transformation that takes as input a *source* model and a *target* model of the plug-in language, as a composition specification consisting of a set of mappings that relate model elements from the *source* model with model elements of the *target* model [SLL+21]. The weaver produces a single model that combines the source model with the target models accordingly to the composition specification.

2.3.3 Perspectives in CORE

In MDE, a system under development is typically modelled at multiple levels of abstraction and from different points of view [SLL+21]. Each realization model is expressed with a modelling language, as articulated above, to provide the appropriate language concepts for expressing the desired properties. Although the concept of languages promotes modularity and separation of concerns, there can be inconsistencies between the numerous realization models for the software system. In order to maintain coherence between the realization models that describe the system, *perspectives* are introduced in CORE.

A perspective groups different languages for a modelling purpose and defines the role of each participating language [AMK22]. Furthermore, a perspective defines composite actions for constructing a coherent multi-model system and maintaining the links between different model el-

ements. These actions are specified by re-exposing, combining, or redefining existing language actions offered by the language as *language actions*.

Perspectives can be realized with Perspectives for Multi-Language systems (PML), a framework for assembling multi-language systems based on existing, independent languages [AMK22]. PML supports a proactive approach for preventing the occurrence of inconsistencies by monitoring and augmenting the language actions to create and update a model. PML maintains consistency conditions including equivalency, equality, and multiplicity constraints across different model elements of different languages. PML promotes modular combination of languages and facilitates consistency and reuse of existing languages across other languages and software systems.

2.3.4 Reusable Aspect Model

Reusable Aspect Models (RAMs) provides an example of a *language* and *perspective* in CORE. RAM is an aspect-oriented multi-view modelling language for describing a software system from the structural and behaviour points of view [KAAK09]. RAM supports the structural point of view of a software system with UML class diagrams and behaviour point of view with UML state and sequence diagrams. A RAM model can consist of a structural view (UML class diagram), various state views, and various message views (UML state and sequence diagrams respectively) and maintain consistency across the different views and models with the RAM metamodel. With the concern reuse process, RAM supports aspect dependency chains, which enables an aspect providing complex functionality to reuse the functionality provided by the other aspects. RAM provides a weaver to create weaved views of composed RAM models for debugging, simulation, or code generation purposes, as well as perform consistency checks during the weaving process and on the weaved model to detect inconsistencies of the composition. The *Design Modelling with RAM* perspective enables the creation and modification of RAM models in TouchCORE.

For the RESTify approach, we represent the existing software system with a RAM model and import its structure with *ImplementationClass* model elements. We also define the Maven coordinates of the generated RESTful web service and dependency to existing software system with the *ArtifactSignature* model elements, which specifies the *groupId*, *artifactId*, and *version*.

2.4 FIDDLR

For the RESTify approach, we construct a RESTify concern to encapsulate REST frameworks, design a ResTL modelling language for expressing REST endpoints, realize inter-model mappings to decorate REST annotations, and implement a transformation pipeline for the code generation of the RESTful web service. It can be extremely difficult to devise the exact technologies required for the RESTify methodology. Furthermore, the design and implementation of each of the steps

is nontrivial and significantly complex. In order to facilitate the design and construction of the technologies required for the RESTify approach, the Framework for the Integration of Domain specific moDelling Languages with concern-oriented Reuse (FIDDLR) [SKK21] is introduced concurrently with this thesis to structure the modularization of our approach.

FIDDLR integrates the ideas of domain-specific modelling language, concern-oriented reuse, and model-driven engineering to modularize concerns that cross-cut multiple levels of abstractions of the software development process and streamline the reuse process [SKK21]. The FIDDLR framework is illustrated in Figure 2.2. The FIDDLR procedure can be separated into three steps, namely *Concern Design*, *Concern Us*e and *Concern Composition*.

The *Concern Design* step emphasizes the step of designing a concern that encapsulates multiple MDE disciplines such as DSML design, and model transformations. The complex task of designing a concern can be further simplified in FIDDLR to four smaller and independent steps, namely *Concern Realization*, *CSML Design*, *CSML to GPML Transformation*, and *CSML to Composition Specification*.

The first step *Concern Realization* illustrates the process of realizing the most appropriate General Purpose Modelling Language (GPML) at the correct levels of abstraction. This step is illustrated in Figure 2.2 as Step 1. The specific type of models that are needed are dependent on the MDE process, and the nature of the concern.

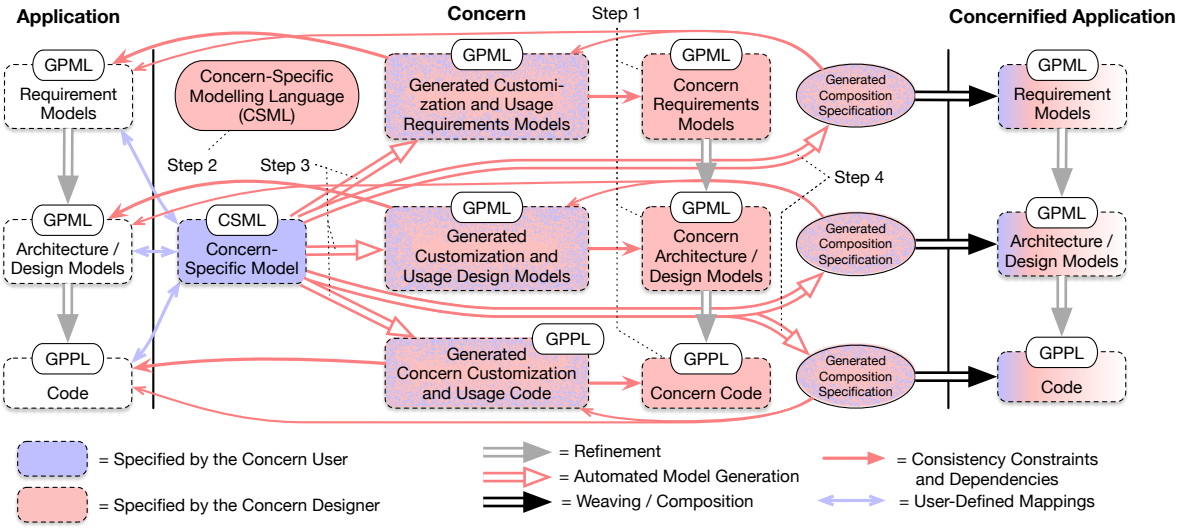
The second step *CSML Design* is applied when the nature of a concern and its properties do not align or can not easily be expressed with GPMLs, or when a concern covers several MDE abstraction layers. Concern-Specific Modelling Language (CSML) is targeted at exposing the concepts of a concern, similar to how a Domain-Specific Modelling Language (DSML) exposes the concepts of a domain. This step is illustrated in Figure 2.2 as Step 2.

The third step *CSML to GPML Transformations*, defines a series of model transformations that when given an input CSML, can generate the appropriate GPML models or code that can customize and make use of the developed GPML realization models or code of the concern for each relevant level of abstraction. This step is illustrated in Figure 2.2 as Step 3.

The last step *CSML to Composition Specification* involves deciding the appropriate level of abstraction the concern-specific model is best composed with the application's realization models and designing a transformation that given a CSML model and mappings as input, produces a composition specification for the customized GPML models in the third step. This step is illustrated in Figure 2.2 as Step 4.

Concern Use illustrates the reuse process of a model packaged with its own CSML that describes the concern-related properties in the context of the application in which it is reused. This is shown in blue in Figure 2.2. Customization and usage of the concern-specific model requires

Figure 2.2: The FIDDLR Framework



only linking the appropriate model elements from the created CSML model to model elements in the GPML models of the application as illustrated with the blue arrows.

Concern Composition specifies the automatically generated GPML models that contain application-specific customization mappings and usage of the concern API (Step 3) with the CSML models and model transformations, as well as the composition specifications that connect the generated models with the application models at each relevant level of abstraction (Step 4). The automatically generated models and composition specifications are highlighted in Figure 2.2 in specked blue and red. The composition specifications and models are then provided as input to the CORE model weavers, which generate the "concernified" application, i.e., the GPML models in which the concern-specific and application-specific structure and behaviour have been combined [SKK21].

2.5 REST

REpresentational State Transfer (REST) [Fie00] is a software architectural style for distributed hypermedia systems initially proposed by Roy Fielding in his Doctor of Philosophy thesis in 2000. In the following decades, REST has become a standard interface for web services, and numerous organizations have developed their own implementation frameworks based on the architectural style. REST defines six guiding constraints, namely *uniform interface*, *client-server*, *stateless*, *cacheable*, *layered system*, and *code on demand*.

The most fundamental constraint of REST is *uniform interface*, it simplifies and decouples the architecture, allowing each component to evolve independently. The uniform interface defines

four additional constraints to be satisfied, namely *resource identification in requests*, *resource manipulation through representations*, *self-descriptive messages*, and *hypermedia as the engine of application state*. *Resource identification in requests* illustrates that individual resources can be identified in requests, including resource URI, HTTP request method, and payloads. *Resource manipulation through representations* indicate that when a client holds a representation of a resource, it already has sufficient information to manipulate or delete the resource's state. *Self-descriptive messages* specifies that each exchanged message between the client and server includes enough information to describe how the message can be processed. An example can be a specific payload media type, such as JSON, XML etc. *Hypermedia as the engine of application state* (HATEOAS) allows users to dynamically navigate to appropriate resources by traversing hypermedia links defined in a server response. Through this approach, clients do not need any prior knowledge regarding the structure of the underlying architecture of the application. The uniform interface constraint defines a standardized approach for designing RESTful web services.

The *client-server* distributed systems design pattern enforces the principle of separation of concerns by decoupling the user interface concerns from the data storage concerns. This property can allow both the client and server to evolve independently, provided that the interface between them is not altered [Gup22b].

The *statelessness* property of RESTful web services mandates that each request from the client to the server must contain all of the information necessary to understand and complete the request [Gup22b]. Clients are required to maintain the session state as opposed to the server. A stateless server can service any client at any time without the session affinity issue. A key advantage of stateless servers is that it can be scaled for significantly more clients as opposed to stateful servers, since the server does not maintain any of its client session information.

The *cacheable* constraint in REST enables the capability of storing copies of frequently requested server responses in the client itself, to minimize the traffic to the server. A server response should provide information in the HTTP headers for whether the response can be cached, the components that are permitted to cache the response, and the maximum duration for a cache to persist. An effective cache can minimize the number of client-server interactions, and directly optimizes the server network, resulting in an improvement in the performance of the system.

The *layered system* constraint in REST can encapsulate the underlying server architecture, in which the client cannot distinguish whether it is connected to the end server or an intermediary server. Intermediary servers are typically used to improve system scalability by enabling load balancing between servers through shared caches. Another application of an intermediary server can be for security purposes, to separate the business logic from the security logic in a web service and enforce security policies. Intermediary servers can call each other to generate an appropriate

response to a client.

Code on demand is an optional constraint for RESTful webservices. It states that the server can provide extensions to the client's functionality by directly sending executable code to the client. For example, in the context of the web, the server can send JavaScript code to a client interacting with the server with a browser. The code that the client receives can now be executed to achieve new functionality, such as animations. Code on demand is optional because it can reduce the visibility of the client system and most web applications do not require this form of flexibility.

In addition to the above constraints for REST, there are also naming and behavioural conventions for endpoints. A RESTful web service defines an endpoint with a resource URI and a HTTP request method. The best practice for resource URI naming is using nouns to represent resources as opposed to verbs, as the action performed should be interpreted by the HTTP request method. Resource URI paths should be structured hierarchically, increasing in specificity from left to right. The commonly used HTTP request methods include *POST*, *GET*, *PUT*, and *DELETE*, in which they correspond to the Create, Read, Update, and Delete (CRUD) operations respectively. *POST* operations submit an entity to the specified resource, often resulting in a change of state in the server. *GET* operations request a representation of the specified resource and should only retrieve data. *PUT* operations replace all the current representations of the target resource with the request payload. *DELETE* operations delete the specified resource. It is essential for the behaviour of a REST endpoint to be defined as indicated by its HTTP request method.

There are numerous REST implementation frameworks designed for different platforms. With the RESTify approach, we provide code generation of RESTful web services in the Java programming language. Consequently, we provide a selection of popular REST frameworks for Java, including Spring Boot, Eclipse Jersey, JBoss RESTEasy, and Apache CXF. The following subsections elaborate on the technical details of these REST frameworks.

REST frameworks designed for Java generally integrate Java annotations to provide a simple procedure for incorporating the framework and signify elements of the REST architectural style. Java annotations have three uses, including information for the compiler, compile-time and deployment-time processing, and runtime processing [Cor]. Table 2.1 illustrates common REST annotations and the implementation differences of these annotations from Spring and JAX-RS.

Instead of implementing extensive build scripts for the build life-cycle of the generated RESTful web service, we incorporate Apache Maven [PZL], a build automation tool primarily used for Java projects. Maven streamlines the build life-cycle of a software system by configuring a Project Object Model (POM) which is stored as a singular file, namely pom.xml. POM defines the required build configuration for the entire software project.

2.5.1 Spring Boot

Java Spring Boot (Spring Boot) [Edu] is a popular implementation of REST for the Java programming language based on the Java Spring Framework (Spring Framework) for developing web applications and microservices. The Spring Framework, initially developed by Rod Johnson, is an open source, enterprise-level framework for creating standalone, production-grade applications that run on the Java Virtual Machine (JVM). Spring Boot offers three core capabilities, including *autoconfiguration*, *an opinionated approach to configuration*, and *the ability to create standalone applications*. These features provide a tool that allows developers to set up a Spring-based application with minimal configuration and setup.

Autoconfiguration allows Spring Boot applications to be initialized with pre-set dependencies, such that users do not have to manually configure them [Edu]. In particular, it automatically configures the underlying Spring Framework and other required third-party packages. The autoconfiguration feature allows developers to focus on the development of Spring-based applications, rather than the configuration of various required dependencies.

Spring Boot uses *an opinionated approach to add and configure starter dependencies*, based on the needs of the project [Edu]. Spring Boot offers a simple web form at `https://start.spring.io/` in which users can define the needs of the project, and Spring Boot selects the appropriate dependencies according to the provided requirements. After selection of the required functionality and dependencies, users can simply generate the project directory as a template from the web form and begin development.

One of Spring Boot's key features is its *capability of creating standalone applications* that can run on its own, without relying on external web servers, such as Tomcat or Jetty [Edu]. Spring Boot achieves this by embedding a web server, typically Tomcat, directory in its build configuration, thus eliminating the requirement to package and deploy Web Application Resource (WAR) files. As a result, users can deploy the application on any platform with a simple Java execution command.

With support from the Maven build automation tool, Spring Boot applications can be packaged and deployed as a Java ARchive (JAR). Within the Maven build configuration file, Spring Boot requires the appropriate *parent*, *dependencies*, and *build plugins* to be specified. Additionally, Spring Boot requires the definition of a launcher class which bootstraps and initializes the framework.

As an implementation of REST, the Spring Framework provides annotations for decorating controller endpoints. We highlight the most commonly used annotations in Table 2.1. The Resource URI annotation is not an annotation by itself; however, it is contained within the HTTP request method annotation. Spring Boot defines individual annotations for each HTTP Request Method. The URI Match Pattern with Parameter annotation can provide mappings from wildcard URI patterns within the resource URI to a specific *parameter* in the controller operation. The

HTTP Request Body and HTTP Response Body annotations define the input and output payloads respectively. The Controller Class annotation specifies that the decorated Java class is a web controller which contains REST endpoint operations. The *@RestController* annotation is used for both the HTTP Response Body and Controller Class annotations as it conveniently combines the *@ResponseBody* and *@Controller* annotations respectively. Since Spring Boot defines a launcher class, a Launcher Class annotation is also required to mark the class and trigger the auto-configuration and component scanning features.

2.5.2 JAX-RS

Jakarta RESTful Web Services (JAX-RS) [Gui], developed by the Eclipse Foundation, is a Jakarta EE API specification that provides support for creating RESTful web services using Java. JAX-RS provides a specification interface for REST implementations to follow, by defining its own annotations and support class for configuring REST controllers. Popular implementations of JAX-RS include Eclipse Jersey, JBoss RESTEasy, and Apache CXF, as documented below.

Table 2.1 illustrates the commonly used annotations that JAX-RS defines, and how they differ from Spring. JAX-RS defines a Resource URI annotation, as opposed to Spring, where it is contained within the HTTP Request Method annotation. In JAX-RS, a HTTP Response Body annotation is placed on an endpoint operation, while in Spring, it placed on the controller class, as it is contained with the *@RestController* annotation. JAX-RS does not define a launcher class, as implementations of JAX-RS are typically deployed as a WAR file, and run on a Java application server, such as Eclipse Jetty. Consequently, JAX-RS does not define an explicit annotation for launcher classes. However, JAX-RS implementations define an *Application* class with the *@ApplicationPath* annotation which specifies the components of a JAX-RS application and supplies additional meta-data.

2.5.2.1 Eclipse Jersey

Eclipse Jersey [Fou21], developed by Oracle Corporation and Eclipse Foundation, is an open-source framework that implements the JAX-RS specification for developing RESTful web services. Eclipse Jersey applications are commonly deployed as a WAR file and run on a Java application server. Jersey implements the same annotations and *Application* class as JAX-RS, and developers need to add the appropriate *dependencies* and *build plugins* to the Maven configuration file.

2.5.2.2 JBoss RESTEasy

JBoss RESTEasy [Hat], developed by Red Hat Incorporated, is another open-source implementation of the JAX-RS specification. Similar to Eclipse Jersey, JBoss RESTEasy applications are typically deployed as a WAR file, and executed on a Java application server. JBoss RESTEasy

Table 2.1: Spring VS JAX-RS Annotations Table

Annotation Type	**Spring Annotation**	**Spring Annotation Location**	**JAX-RS Annotation**	**JAX-RS Annotation Location**
Resource URI	path="/"	HTTP Request Method Annotation	@Path	Operation
GET HTTP Request Method	@GetMapping	Operation	@GET	Operation
PUT HTTP Request Method	@PutMapping	Operation	@PUT	Operation
POST HTTP Request Method	@PostMapping	Operation	@POST	Operation
DELETE HTTP Request Method	@DeleteMapping	Operation	@DELETE	Operation
URI Matching Pattern with Parameter	@PathVariable	Parameter	@PathParam	Parameter
HTTP Request Body	@RequestBody	Parameter	@Consumes	Operation
HTTP Response Body	@RestController	Class	@Produces	Operation
Controller Class	@RestController	Class	@Path	Class
Launcher Class	@SpringBoot Application	Class	N/A	N/A
Application Class	N/A	N/A	@ApplicationPath	Class

incorporates the same annotations and *Application* class as JAX-RS. With Maven, RESTEasy can be configured by injecting the necessary *dependencies* and *build plugins* in the configuration file.

2.5.2.3 Apache CXF

Apache CXF [Fou], developed by the Apache Software Foundation, is another open-source web services framework that implements the JAX-RS specification. Apache CXF integrates the JAX-RS annotations and *Application* class as well. Apache CXF applications generally implement a launcher class that instantiates and executes the built-in application server to expose the specified endpoints in the *Application* class. Consequently, Apache CXF applications are commonly deployed as a self-contained JAR. With Maven, CXF is configured by including the associated *dependencies* and *build plugins* in the build configuration file.

3
Overview of RESTify

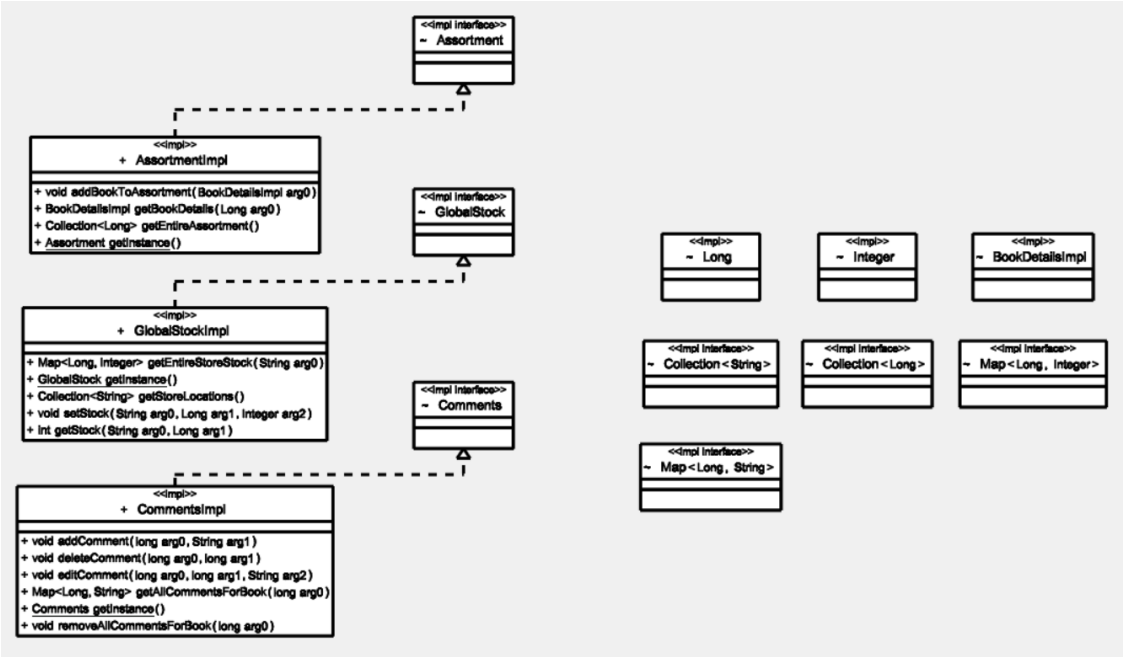
In this chapter, we present an overview of RESTify, a concern that incorporates MDE and CORE techniques to automate the development process of RESTful web services with a series of model-to-model, model-to-code, and code-to-code transformations. We currently view the underlying transformations as black boxes, in which the implementation details will be elaborated in the following chapters.

Throughout the thesis, we illustrate the use of RESTify with a simple Java application of a BookStore [Sch20a] from a user's perspective. The BookStore application implements the fundamental functionality of a book store, e.g., managing the stock of books available at each of the store locations. Each *book* stored in the BookStore application is indexed by an ISBN, and contains information regarding its price, title, author, and description. The BookStore chain has *agencies* in different cities in which there can be only one agency in the same city, and each city has a particular stock for each of the indexed books. The BookStore chain also stores *reader feedback* for the indexed books, in which each comment is anonymous and has no author, comments can be updated, and comments must not be empty. In the current state of the BookStore application, users can query and manipulate the database with its defined public methods. With the RESTify approach, we transform the original BookStore application to a RESTful web service with the TouchCORE modelling tool such that users can now consult or modify the database remotely. This process involves 6 steps, outlined in the remaining sections of this chapter.

3.1 Step 1: Importation of Existing Software System with RAM

In TouchCORE, we can represent the RESTification of the BookStore application as a concern and model the existing application with a Reusable Aspect Model (RAM). The existing BookStore application is modelled in Figure 3.1. On the left-hand side, interfaces, classes and methods from the BookStore application are imported into the RAM model. In particular, the classes *AssortmentImpl*, *GlobalStockImpl*, and *CommentsImpl* are controller classes that contain methods

Figure 3.1: RAM Model Representation of BookStore Application in TouchCORE

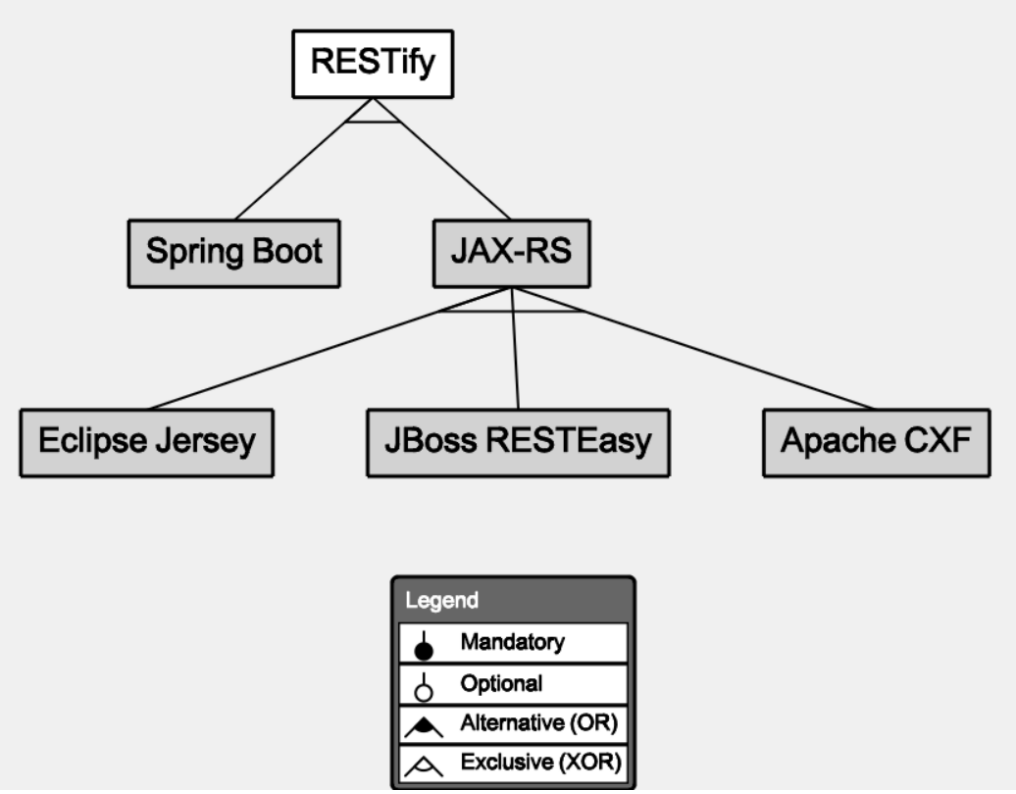


to query and modify the underlying database. The business logic of the generated RESTful web service is contained within the imported methods as defined in the source code of the controller classes. The generated REST endpoints will delegate the client requests to these methods. On the right-hand side, helper Java types and data structures are included to support their use in the parameter and return types of operations in the imported classes from the BookStore application. Now, to generate a functional Maven project, we need to define a *dependency* to the existing software system by specifying its Maven coordinates with the *ArtifactSignature* model element, which includes the *groupId*, *artifactId* and *version* identifiers. Additionally, we establish a new Maven *ArtifactSignature* for the generated RESTful web service with Maven coordinates.

3.2 Step 2: Reuse of RESTify Concern

As part of this thesis, we have implemented a RESTify concern encapsulating a REST-specific modelling language and several REST implementation platforms. The platforms are exposed as *features* of the concern as illustrated in Figure 3.2. The feature model is part of the *variation interface* of a concern and provides different possible variations of addressing the concern's domain of interest. In the case of RESTify, we provide different REST implementation technologies including Spring Boot, and Jax-RS implementations - Eclipse Jersey, JBoss RESTEasy, and Apache

Figure 3.2: RESTify Concern



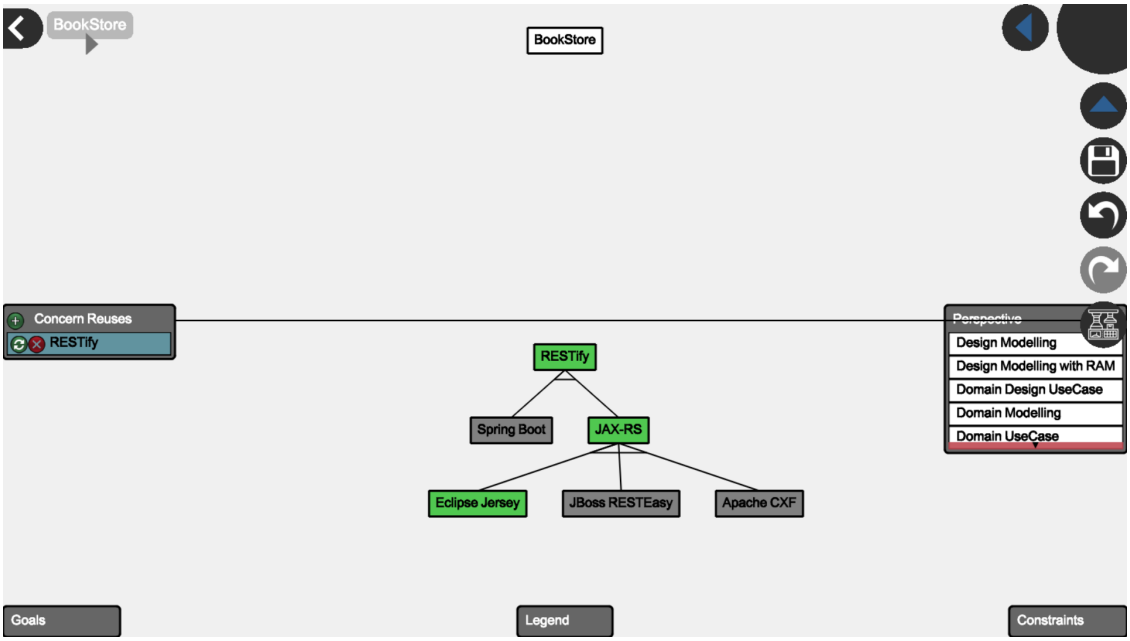
CXF and encapsulate each technology within a feature in the feature model. With the use of the XOR relationship, visualized by the white arc between features, we force users to select a single implementation from either Spring Boot or one of the JAX-RS implementations.

In order to create a REST interface for the BookStore application as well as generate an implementation, and in line with the CORE reuse process explained in section 4.2, a user must *reuse* the RESTify concern that we provide and select a valid configuration with the desired technology. In Figure 3.3, we illustrate the reuse of the RESTify concern within the BookStore concern with the selected *Eclipse Jersey* implementation.

3.3 Step 3: Design of a REST Resource Tree with ResTL

Thanks to the reuse of the RESTify concern, the user can then design the architecture of the BookStore RESTful web service's endpoints with a REST-specific modelling language called Resource Tree Language (ResTL). The ResTL model visually represents a resource layout for the REST interface. Each node in the resource tree defines an individual fragment in the resulting URI. The

Figure 3.3: Reuse of RESTify Concern in BookStore Concern



four circular buttons within a resource tree node illustrated as *G*, *PU*, *PO*, and *D*, represent each of the HTTP request methods - *GET*, *PUT*, *POST*, and *DELETE* respectively. With ResTL, a REST endpoint is defined by a HTTP request method button, and its elaborated URI from the ancestors.

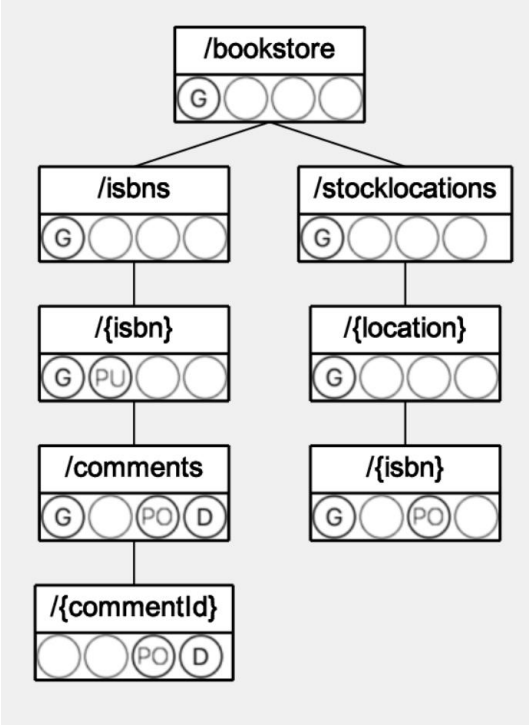
We illustrate a simple ResTL model for the BookStore application in Figure 3.4 that contains the appropriate REST endpoints to decorate the BookStore application. For example, the REST endpoint with */bookstore/isbns/{isbn}* URI and *GET* HTTP request method is modelled in the */isbns* node with the *G* button and will retrieve a list of the ISBNs of all of the books in the BookStore application. The business logic of this endpoint will be provided by the original BookStore Java implementation.

3.4 Step 4: Specifying Inter-Model Mappings Between RAM and ResTL

After the import of the Java implementation into a RAM model, which encapsulates the Java implementation of the business logic, and the construction of a ResTL model, which specifies the BookStore REST interface, we need to specify a mapping between the implementation and the interface. This is done by placing inter-model mappings between each REST endpoint and the appropriate method in the BookStore application. If the behaviour requires parameters to

3.4 Step 4: Specifying Inter-Model Mappings Between RAM and ResTL

Figure 3.4: BookStore REST Resource Tree with ResTL Model



be passed, additional parameter mappings between, e.g., URI segments, and method parameters also have to be specified. To support the creation of these inter-model mappings, we developed a generic split view to view a RAM and ResTL model simultaneously and establish mappings between them. We currently support two types of mappings for the RAM-ResTL split view - HTTP request method to operation mappings, and URI matching pattern to parameter mappings. A HTTP request method button in the ResTL model contains enough information to represent a REST endpoint, by elaborating its URI with its ancestor nodes. The HTTP request method to operation mapping maps a REST endpoint from the ResTL model to a controller operation in the RAM model. Consequently, a REST endpoint is now correlated to a particular controller method that contains the business logic to query and modify the BookStore application. The URI matching pattern to parameter mappings map a *dynamic* node in the ResTL model as illustrated with the curly brackets in the node name to a parameter of a controller operation in the RAM model. The purpose of the parameter mapping is to capture the dynamic information stored in the dynamic section of the URI in a parameter.

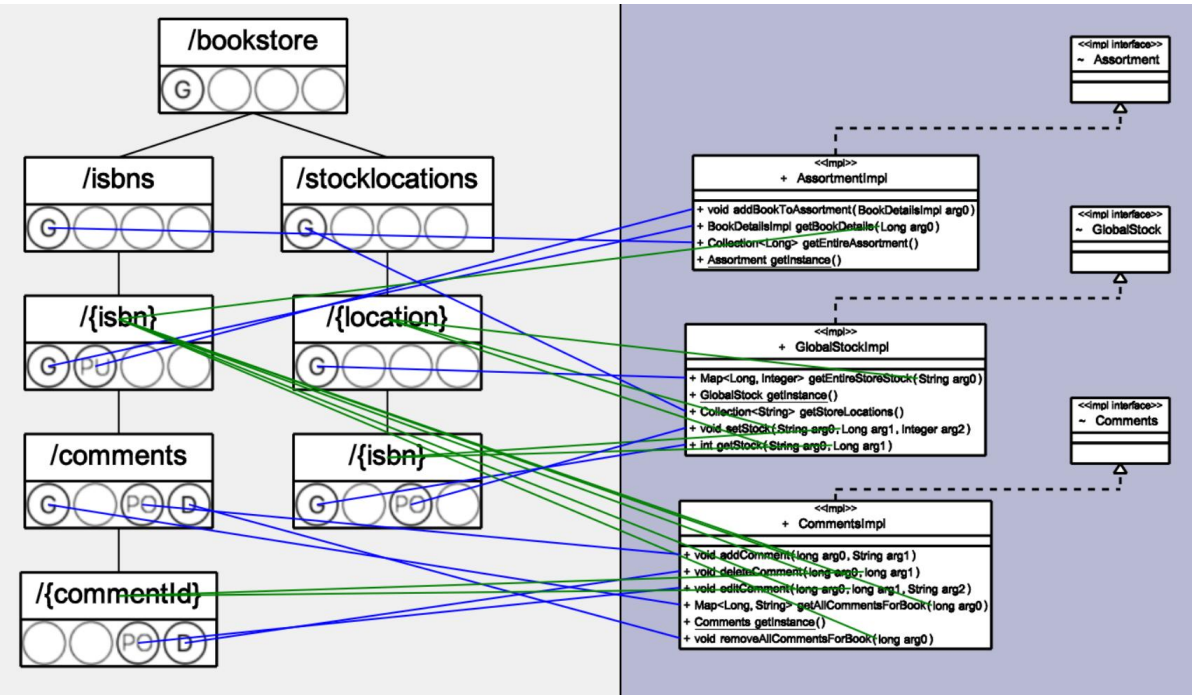
An instance of the generic split view that captures the RAM and ResTL models of the BookStore concern and their inter-model mappings is illustrated in Figure 3.5. The REST endpoint to controller operation mappings is visualized in blue, while the URI matching pattern to parameter mappings are visualized in green. In particular, we highlight an example with the mappings that correspond to the *void removeAllCommentsForBook(long arg0)* method in the *CommentsImpl* controller class from the RAM model. First, there is a REST endpoint to controller operation mapping from the following REST endpoint - */bookstore/isbns/{isbn}/comments DELETE* in the ResTL model to the *removeAllCommentsForBook* operation. This mapping links the REST endpoint to the controller operation to delegate the business logic of a HTTP request to the original BookStore application. In addition, there is a mapping from the dyanmic *{isbn}* node to the *arg0* parameter in the *removeAllCommentsForBook* operation. This instructs the REST runtime to take the string from the URI at the location {isbn}, transform it into a long value and use it as the *arg0* parameter when invoking the removeAllCommentsForBook method. Chapter 6 elaborates on the implementation details of the generic split view, and the mapping functionalities.

3.5 Step 5: Execution of RESTify Transformations

Finally, the BookStore concern now contains two models, namely RAM and ResTL, inter-model mappings between them, and the reused RESTify concern, and is ready to execute the RESTify transformations. The input to the RESTify transformation pipeline is the complete BookStore concern as developed from the above steps, and the output is a functional, and ready to be deployed RESTful web service that uses the selected REST framework in the RESTify concern.

3.5 Step 5: Execution of RESTify Transformations

Figure 3.5: RAM-ResTL Split View with Mappings for BookStore Concern



An intermediate artefact is generated during the RESTify transformation, which is a *functionality poor BookStore RAM model* that clones the controller classes and operations, annotates them with the appropriate REST annotations based on the selected implementation technology, and contain *sequence diagram* models to delegate the business logic of each REST endpoint to the original BookStore application. This intermediate RAM model also contains *maven dependencies* to the selected REST implementation technology. Next, the original BookStore RAM model is weaved with the intermediate functionality poor BookStore RAM model to generate a *weaved* RAM model with the standard CORE weaver in TouchCORE. The weaved RAM model is now used to generate the source code files. These contain the Java files, and the Maven configuration file, namely pom.xml. Chapter 7 expands upon the underlying transformations within the RESTify transformation pipeline. Chapter 8 elaborates an algorithm for the weaving operation of Maven configuration files.

3.6 Step 6: Deployment of Generated RESTful Web Service Application

After the execution of the RESTify transformations, the generated RESTful web service project is located in the concern directory, in a new subdirectory named *generated-maven-project/selected-rest-implementation* where selected-rest-implementation is the selected configuration of the RESTify concern, i.e., *spring-boot*, *eclipse-jersey*, *jboss-resteasy* or *apache-cxf*. This directory contains the pom.xml file and the Java source code in a set of subdirectories as specified by the Java packages.

Now, in order to deploy the generated RESTful web service application, users can simply execute a series of Maven commands on the command line. The specific commands vary based on the REST implementation technology. For Spring Boot, the user can execute *mvn clean package spring-boot:run*. For Eclipse Jersey and JBoss RESTEasy, the user can execute *mvn clean package jetty:run*. And for Apache CXF, the user can execute *mvn clean package* and then *java -jar target/artifactId-version* where *artifactId* and *version* are defined in the pom.xml file. By default, the generated RESTful web service application can be accessed at *localhost* with port *8080*.

4

Concern Reuse With COREReuseArtefact

In the vision outlined in chapter 3, when a user wants to create a REST interface for their application, they are going to initiate this process by *reusing* the RESTify concern. Unfortunately, such a reuse was not supported by the CORE metamodel and the TouchCORE tool.

The standard CORE *concern reuse process* is always initiated from within a realization model, i.e., it enables a customized reuse of a model representing a specific configuration of a concern from within some other model. For example, when modelling a design of an application with the Reusable Aspect Models (RAM) language, a modeller can reuse a RAM model implementing a specific variant of the *Observer* design pattern.

Because reuses were always initiated from within realization models, concern reuses are typically stored within a *COREExternalArtefact* in the CORE metamodel as shown in Figure A.1. A *COREExternalArtefact* encapsulates a model (i.e., instance of a CORE modelling language metamodel).

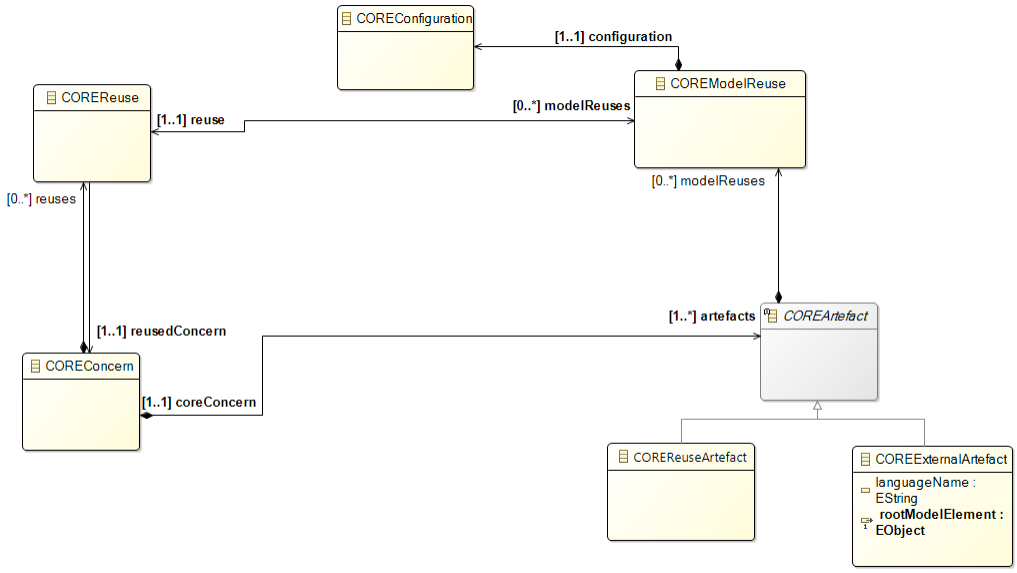
In this chapter, we update the CORE metamodel to support concerns reusing concerns directly. To this aim, we introduce a *COREReuseArtefact*, to enable the concern reuse process without a realization model in section 4.1. In section 4.2, we explain how we added support for direct concern reuse with a *COREReuseArtefact* to TouchCORE.

4.1 Extension of COREReuseArtefact in CORE Metamodel

In order to augment the existing concern reuse lifecycle without an instance of an accepted metamodel, namely a *COREExternalArtefact*, we extend the CORE metamodel with the *COREReuseArtefact* metaclass. We illustrate a snippet of the CORE metamodel that contains only the model elements relevant for the concern reuse procedure along with our newly introduced *COREReuseArtefact* in Figure 4.1. Within the snippet, we only include the model elements and dependencies that are relevant for our new concern reuse pipeline. In particular, we omitt the metaclasses that illustrate the concept of an *extended reuse*, in which when we reuse a concern that already contains

4.1 Extension of COREReuseArtefact in CORE Metamodel

Figure 4.1: CORE Metamodel Extension with COREReuseArtefact



reused concerns recursively, all of the reused concerns are combined in the form of an *extended reuse concern*. Details regarding the *extended reuse* can be seen in the complete CORE metamodel in Figure A.1. The *extended reuse* functionality is not required in our new concern reuse pipeline because when we reuse a concern with the *COREReuseArtefact*, the reused concern does not contain any additional reused concerns.

As illustrated in Figure 4.1, the *COREReuseArtefact* inherits from the *COREArtefact* model element, similar to the *COREExternalArtefact*. The purpose of a *COREReuseArtefact* is to act as an artefact to simply store reused concerns without having an associated *external model*. Now, the concern reuse with *COREReuseArtefact* pipeline starts by encapsulating the reused concern within a *COREConcern* model element. Next, a *COREReuse* is realized and associated with the reused concern. A reuse of a concern requires a specific configuration of the *variation interface* of the reused concern. A *COREConfiguration* captures the specific features selected in the variation interface. Finally, the configuration and reuse of the reused concern is captured within a *COREModelReuse* and stored within an implementation of the *COREArtefact* model element. The *COREReuseArtefact* contains all of the reused concerns as a list of *COREModelReuses* and is stored within a list of artefacts in the original concern.

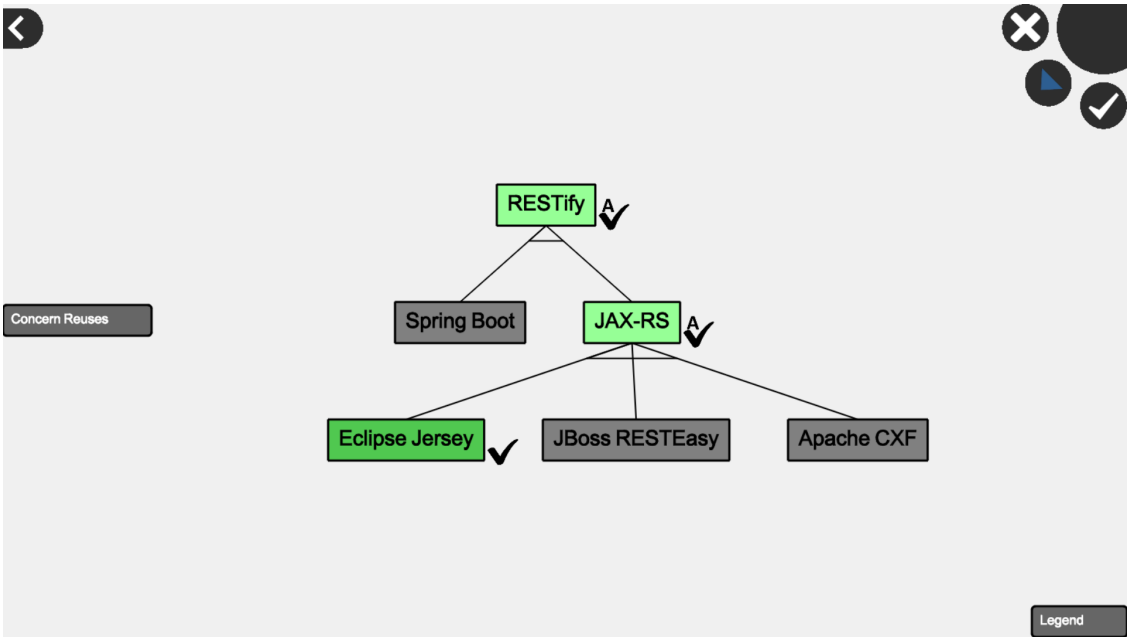
Figure 4.2: Concern Reuse with COREReuseArtefact Button

4.2 Support for Concern Reuse with COREReuseArtefact in TouchCORE

Previously in TouchCORE, a concern reuse is realized through the RAM modelling perspective and can be visualized in both the RAM modelling perspective and feature modelling scene. This process involves a *COREExternalArtefact* which contains the RAM model. We now provide support for concern reuse with *COREReuseArtefact* by integrating a new concern reuse pipeline directly in the feature modelling scene. We illustrate our new concern reuse pipeline below with an example of the step-by-step process of the BookStore concern reusing the RESTify concern with a *COREReuseArtefact*. From the feature modelling scene, we add a new button (button with a "+" sign) next to the Concern Reuses panel as visualized with a red circle around it in the middle-left section in Figure 4.2. Upon clicking on the concern reuse button, the user will be prompted a file directory browser to select an appropriate concern to reuse, which is the RESTify concern in this example.

After the selection of a concern to be reused, the feature selection scene is presented to the user to choose an appropriate configuration for the reused concern. We provide an illustration of selecting the Eclipse Jersey configuration of the RESTify concern within the BookStore concern in Figure 4.3. After clicking on the check mark button in the feature selection scene, our new controller operation in the *ReuseController* will execute a collection of EMF commands to add the

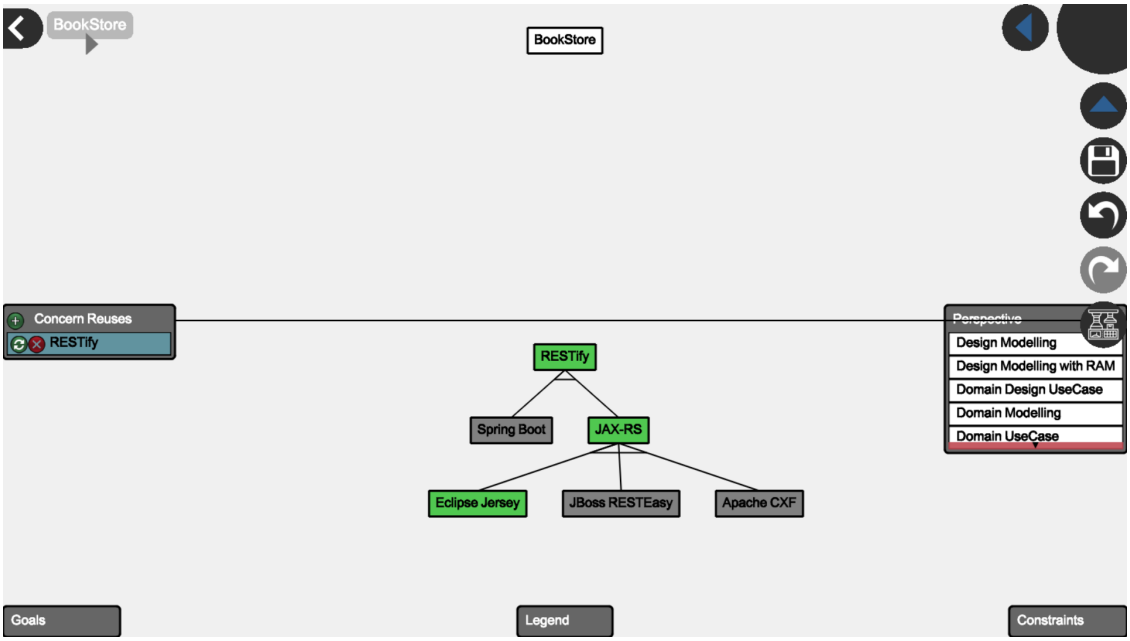
Figure 4.3: Selection of a Configuration for RESTify Concern



reused concern in the *COREReuseArtefact*. In particular, a *COREReuse* model element is created and associated with the RESTify concern. A *COREModelReuse* is instantiated and is associated with the created *COREReuse*, and the given *COREConfiguration* of the feature selection. Next, we check for the presence of the *COREReuseArtefact*, in which a new instance is created if it does not already exist in the BookStore concern's list of artefacts. The *COREModelReuse* is then appended to the *COREReuseArtefact's* list of model reuses. In the case that the *COREReuseArtefact* did not exist previously, we also append it to the BookStore concern's list of artefacts.

After the execution of the EMF commands, the user is navigated back to the feature modelling scene in which the user can now visualize the newly added reused concern, RESTify. When clicking on the RESTify element in the Concern Reuses panel, users can visualize the current Eclipse Jersey configuration of the reused concern in Figure 4.4. For concerns that are reused within the *COREReuseArtefact*, we provide two additional functionalities as visualized by the two buttons on the left of the RESTify element in the Concern Reuses panel, namely *reconfiguration* and *deletion*. With the green button with cycle arrows, users can reopen the feature selection scene to change the configuration of the reused concern if desired. In the context of RESTify, the user can change the REST implementation technology. The reconfiguration process is executed by EMF commands as well in the *ReuseController*. Since elements in the Concern Reuses panel are represented as instances of *COREReuse*, we can update the configuration by locating its associated *COREMod-*

Figure 4.4: Reuse of RESTify Concern in BookStore Concern



elElements and setting the *COREConfiguration* to the new selection. With the red button with an X symbol, users can remove a reused concern from the *COREReuseArtefact*. The implementation of the deletion of reused concerns is also done by means of EMF commands in the *ReuseController*. With the selected *COREReuse* to be deleted, we locate the associated *COREModelElements* and delete them from the *COREReuseArtefact*. If there are no more *COREModelReuses* within the *COREReuseArtefact*, the *COREReuseArtefact* is also deleted from the BookStore concern's list of artefacts.

5 ResTL

Resource Tree Language (ResTL) is a modelling language designed to uniquely describes REST resource layouts. The ResTL model can help users visualize and design a well-structured REST interface for their software applications. In this chapter, we elaborate on technical details in the ResTL metamodel, and the various components that we developed as part of this thesis to support the creation and modification of a RestTL model in TouchCORE.

5.1 ResTL Metamodel

The ResTL *metamodel* [Sch20b] is illustrated in Figure 5.1 and is realized using the Eclipse Modeling Framework (EMF). A ResTL model is contained in an instance of the *RestIF* metaclass. The *RestIF* model element always contains a root instance of *PathFragment* and a list of *Resources*. A *PathFragment* describes a section of an endpoint URI as seperated by the forward slash character. A *DynamicFragment* inherits the *PathFragment* model element and represents a URI matching pattern section in the endpoint URI as expressed by the curly brackets. Similarly, a *StaticFragment* also inherits from the *PathFragment* model element and specifies a static section in the endpoint URI. A *PathFragment* reflexively contains children to form a tree data structure. A complete URI is composed of a *PathFragment* and all of its ancestors. For example, consider a ResTL model with a single endpoint URI */bookstore/isbns/{isbn}*, the *ResIF* model element contains a root node that is an instance of *StaticFragment* with an *internalname* of */bookstore*, the root node has a single child that is an instance of *StaticFragment* with an internalname of */isbns*, and finally the */isbns* node has a single child that is an instance of *DynamicFragment* with a *placeholder* of */{isbn}*.

Next, we introduce the *Resource* model element, which contains all of the HTTP request methods associated to a specific endpoint URI. In the ResTL metamodel in Figure 5.1, a *RestIF* model element contains a list of *Resources*, in which each is associated to zero or one *PathFragment* model element. A *Resource* contains a list of one to four *AccessMethod* model elements in which each *AccessMethod* has a *MethodType* attribute that represents each of the HTTP request methods,

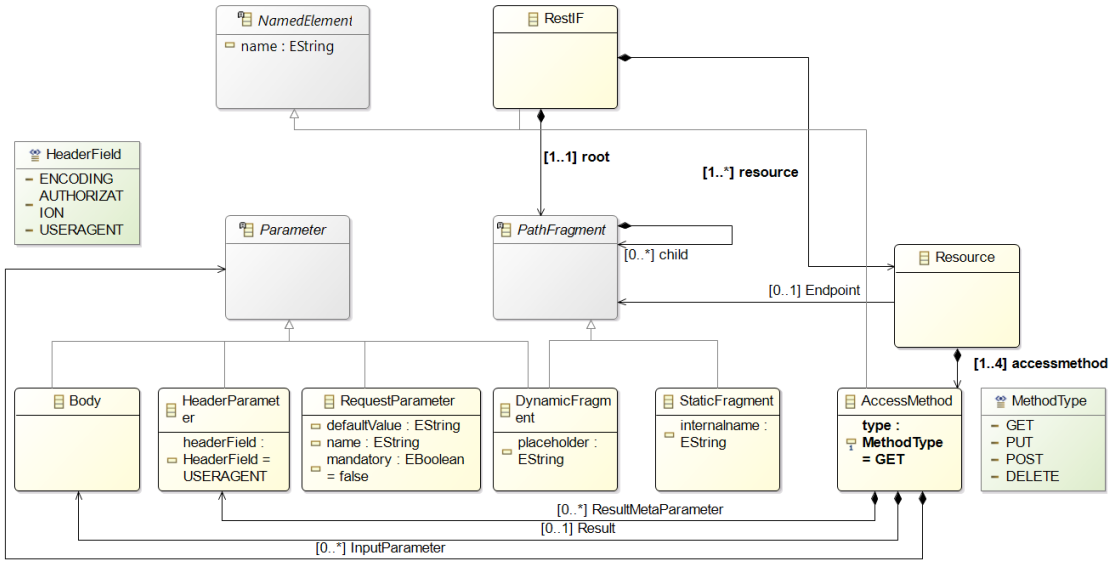
namely *GET*, *PUT*, *POST*, and *DELETE*. Now, to expand upon our previous example, consider a ResTL model with two REST endpoints enabled on the same URI of */bookstore/isbns/{isbn}* : the HTTP request methods of *GET*, and *PUT*. To support the HTTP request methods, we add a *Resource* to the *RestIF* model element, and associate it with the */{isbn} DyanmicFragment* node. This *Resource* contains two *AccessMethod* instances with the *GET* and *PUT MethodTypes*.

The *Parameter* model element groups different kinds of HTTP parameters under an abstract type [Inc22]. A HTTP parameter consists of a type, name and value and can appear in the header or body of a HTTP request. The subclasses of *Parameter* are *Body*, *HeaderParameter*, *RequestParameter* and *DynamicFragment*. A *Body* signifies that the HTTP parameters are provided within a HTTP Request Body. A *HeaderParameter* defines HTTP Header parameters and consists of name/value pairs that appear in the HTTP header. The *HeaderParameter* contains a *HeaderField* attribute which illustrates the HTTP header type and currently supports *ENCODING*, *AUTHORIZATION*, and *USERAGENT*. Consider the previous example of the following endpoint: *GET /bookstore/isbns/{isbn}*, we can integrate HTTP Header parameters such as Content-Encoding to list any encodings that have been applied to the message payload. We can model this HTTP Header parameter with an instance of *HeaderParameter* with the *HeaderField* attribute set as *ENCODING* and associate it with the corresponding *AccessMethod*. Similarly, a *RequestParameter* specifies a HTTP Request parameter that can be used to match fields in the HTTP header. The *RequestParameter* contains *defaultValue*, *name* and *mandatory* attributes where *name* and *defaultValue* illustrate the name/value pair respectively and *mandatory* determines whether the HTTP Request parameter is mandatory. The *DynamicFragment* model element inherits from *Parameter* and *PathFragment* as it is a HTTP parameter that defines a name with the *placeholder* attribute and contains a value in the URI segment implementation. The *AccessMethod* model element contains *Parameters*, *HeaderParameters* and *RequestParameters* as lists linked to a particular REST endpoint.

5.2 Support for ResTL Models in TouchCORE

TouchCORE has a well-defined process that needs to be followed to integrate new modelling languages into the tool. To provide support for interpreting and visualizing ResTL models in TouchCORE, we need to define a new *CORELanguage* and *COREPerspective*. As a reminder, realization models expressed in some modelling language in CORE are stored in their own file and integrated into a concern using an instance of *COREExternalArtefact*. The *COREExternalArtefact* instance has an attribute *rootModelElement* that points to the root model element of the realization model in the model file. As can be seen in Figure A.1, *COREExternalArtefacts* have an attribute *languageName* that must refer to an instance of *CORELanguage* that points to the *.ecore* file containing the metamodel of the modelling language. Hence, for the ResTL metamodel, we define a new *CORE-*

Figure 5.1: ResTL Metamodel



Language named *Resource Tree Language* to interpret ResTL models and link it to the ResTL metamodel shown in Figure 5.1.

Again, as a reminder, *COREPerspectives* are used in CORE to encapsulate one or several *languages* intended to be used for a specific modelling purpose [SLL+21]. To visualize the modelling of a ResTL model, we define a new perspective named *Resource Tree Interface Perspective* to incorporate the *Resource Tree Language* for the creation and visualization of ResTL models.

To build the GUI editor, we develop a set of *scenes*, *views*, *handlers*, and *controllers* that support the creation, modification, saving and loading functionalities of ResTL models in TouchCORE. The graphical user interface of TouchCORE is realized with Multi-Touch for Java (MT4J) [LRZ10], in which we implement abstract GUI elements from MT4J to provide support for ResTL models. First, we develop a *scene* and *scene handler* named *DisplayRestTreeScene* and *DisplayRestTreeSceneHandler* respectively. In MT4J, a *scene* is the highest-level component that contains all of the *view*s and other graphical elements within the application display. *DisplayRestTreeScene* constructs all of the required views for the *Rest Interface Perspective*, adds an EMF listener for updates from the represented ResTL model, and initializes an EMF command stack to support the undo and redo operations with the ResTL model. *DisplayRestTreeSceneHandler* handles the events related to the *DisplayRestTreeScene* including switching to other scenes, exiting the application and the basic undo, redo, and save operations using the EMF command stack.

In TouchCORE, *views* contain the graphical elements used to represent special model ele-

ments. We implement a series of views - *RestTreeView*, *PathFragmentView*, and *ResourceView*, their respective view handlers (*ResourceView* does not have a handler) - *RestTreeViewHandler* and *PathFragmentViewHandler*, to illustrate the represented model elements, namely *RestIF*, *PathFragment*, and *Resource/AccessMethod* respectively. The *RestTreeView* is the highest-level view, which contains and formats the other views. *RestTreeView* incorporates an auto-format system in which it will always ensure that the visualized tree has a balanced structure vertically and horizontally between *PathFragmentViews*. Additionally, *RestTreeView* registers its own MT4J gesture processors to support the following gestures for the visualization, addition, modification, and deletion of model elements - single tap, double tap, tap and hold, pinch, wheel events, right click drag and *Unistroke* gestures. A *Unistroke* gesture visualizes a yellow line as the user holds left click on the mouse and drags it through the application. The gesture events are then handled by the *RestTreeViewHandler*, to delegate the appropriate operation to the *controller*. The *RestTreeView* has an EMF listener for changes to the ResTL model and handles each of the operations to visually maintain consistency with the underlying model. The *PathFragmentView* visually represents a *PathFragment* model element as the upper half rectangle of a tree node with a text field. The text field encloses its internal text with curly brackets when the represented node is a *DyanmicFragment* and does not enclose its text with curly brackets when the represented node is a *StaticFragment*. Events related to a *PathFragment* are notified to the *PathFragmentView*, who then redirects the events to the *RestTreeView* to be properly handled. The *PathFragmentViewHandler* provides a menu when users tap and hold on a *PathFragmentView*, and delegate the appropriate behaviour to the *controller* from the clicked button on the menu.

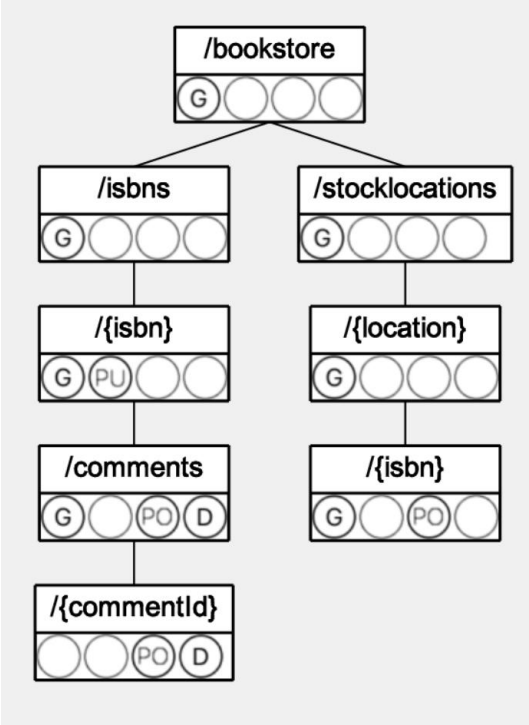
A *ResourceView* visually represents a *Resource* and its associated *AccessMethod* model elements as the lower half rectangle of a tree node with four buttons. Each of the four buttons represent each of the *MethodType* model elements encapsulated by an *AccessMethod*, namely *GET*, *PUT*, *POST*, and *DELETE* with a *G*, *PU*, *PO*, and *D* icon respectively. The buttons can be pressed to toggle the presence of the *AccessMethod* and call the controller to edit the ResTL model accordingly. Hierarchically, a *PathFragmentView* contains a *ResourceView* to achieve the functionality of adding and removing *Resource* and *AccessMethod* model elements that are associated to a specific *PathFragment*.

We implement the *language actions* for ResTL with a single controller class, namely *RestTreeController*, which contains operations for the creation, modification, and deletion of current ResTL model elements through EMF commands. These controller operations are called through the supported gestures and helper menus in the *RestTreeView* and its *handler*s. Currently, we support various operations regarding a *PathFragment*, *Resource*, or *AccessMethod* model element. We support the creation of child *StaticFragment* and *DynamicFragment* nodes by a *UnistrokeGesture*

that occurs from a parent *PathFragment*. The *placeholder* or *internalname* attribute of a *DynamicFragment* or *StaticFragment* respectively can be modified by a simple double tap gesture on the text field of a specific *PathFragment*. The ordering of children *PathFragments* can be modified by a right click drag gesture on a specific *PathFragment* to the desired position in the children list. Similarly, the direct parent of a *PathFragment* can be modified by a right click drag gesture on a specific *PathFragment* to another *PathFragment* that will become its new parent. We support convenient switching between a *PathFragment's* implementation type between *DyanmicFragment* and *StaticFragment* by a button on the pop-up menu from a specific *PathFragment*. A *PathFragment* can also be deleted by a button on the pop-up menu, in which the deleted *PathFragment's* children are re-appended to the parent of the deleted *PathFragment*. When a *PathFragment* is deleted, all of its associated model elements such as *Resource* and *AccessMethod* are deleted as well to maintain consistency in the model. A *Resource* is added automatically when a *MethodType* button is pressed, to contain the *AccessMethod* model elements. The added Resource is then associated with the *PathFragment* that contains the *MethodType* button in its view. In addition, when a *MethodType* button is pressed, it will add or remove an *AccessMethod* within the associated *Resource* depending on whether it was present already. Each of the operations elaborated above is stored as EMF commands within the EMF command stack to support the undo and redo operations. In particular, when the EMF command stack is nonempty, the user can save the model.

A visualization of the BookStore REST Resource Tree can be seen in Figure 5.2. Since there is an extensive amount of REST endpoints illustrated, we elaborate on their behaviour by combining multiple endpoints with URIs. The */bookstore/isbns* and */bookstore/isbns/{isbn}* URIs provide endpoints for retrieving and modifying information regarding books in the BookStore application. The */bookstore/isbns/{isbn}/comments* and */bookstore/isbns/{isbn}/comments/{commentId}* URIs provide endpoints for getting, updating, appending and deleting comment information that correspond to books in the BookStore application. Finally, the */bookstore/stocklocations*, */bookstore/-stocklocations/{location}* and */bookstore/stocklocations/{location}/{isbn}* URIs provide endpoints for getting and adding agency information and their stock of books in the BookStore application. The BookStore ResTL model provides an endpoint for each of the controller operations that already exist in the BookStore application, to conveniently map them together as elaborated in the next chapter.

Figure 5.2: BookStore REST Resource Tree with ResTL Model



6
Generic Split View

In order to apply the RESTify code generation pipeline, we must establish mappings between the RAM model and the ResTL model. These mappings serve two functions - linking business logic of controller methods to specific REST endpoints and parameters to URI matching patterns within the URI of the corresponding REST endpoint. The most intuitive method of illustrating mappings is to simply draw lines between the two models, and hence we develop a split view to support these operations in TouchCORE.

The generic split view enables the visualization and modification of two arbitrary models simultaneously. In addition, we implement an optional mapping functionality to allow the creation and removal of inter-model mappings. The mappings can be validated to ensure the correctness and consistency of the mappings to the underlying models. We provide two implementations of the generic split view below, namely the RAM-ResTL Split View and the Domain-Use Case Split View to demonstrate the flexibility of the generic split view.

Now, with the RAM-ResTL Split View, we can view the RAM and ResTL models of the BookStore concern simultaneously and realize the required inter-model mappings for code generation transformations.

6.1 Abstract Generic Split View

The generic split view is implemented with Java abstract and generic classes, as a new *view* and *view handler* in TouchCORE, namely *GenericSplitView* and *GenericSplitViewHandler*. *GenericSplitView* takes the *scene* of the currently displayed model and the *view* of any arbitrary model as input and visualizes them by extracting the top-level view from the scene and displaying both views side-by-side simultaneously. Users can navigate to the *GenericSplitView* from the scene of an arbitrary model with the enter split view button. The enter split view button visualizes additional realization models that are contained within the current *COREScene* model element to be selected as the secondary view to be displayed in the *GenericSplitView*. Therefore, after the navi-

gation process to the *GenericSplitView*, the view is now updated but the scene remains the same.[1] The inner- and outer-level views of the *GenericSplitView* are separated by a split line to clearly distinguish the views from each other. There is a button in the generic split view to toggle between the display configuration of the two views, between a vertical or horizontal layout, the default of which is vertical. The *GenericSplitView* registers its own MT4J gesture processors in an overlay layer to support the following gestures - single tap, double tap, tap and hold, pinch, wheel events, right click drag and *Unistroke* gestures.

The event flow of gestures is illustrated in Figure 6.1 in which the MT4J gesture processor instantiates the gesture event and delegates the event to the *GenericSplitView* overlay, in which it is processed by either the inner-level view, outer-level view, or the *GenericSplitView*. The *GenericSplitViewHandler* receives the gesture events from the overlay and determines whether to handle the event itself or delegate it to either the inner- or outer-level views. The *GenericSplitViewHandler* supports the drag gesture if it is sufficiently close to the split line, in which it will drag the split line to adjust the designated space for the inner- and outer-level views. There is an additional button in the generic split view to enable the delegation of gestures to the inner- and outer-level views. When enabled, gestures captured by the overlay that are not processed by the generic split view will be delegated to the associated inner- or outer-level view. When a gesture modifies one of the underlying views, the individual view handles the EMF notification to maintain the visual consistency with the represented model.

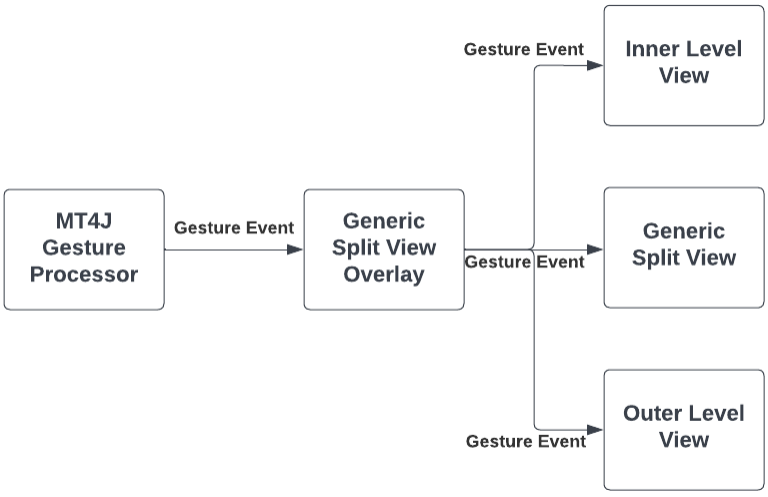
6.2 Abstract Generic Split View with Mappings Functionality

In some cases, we might simply want to display two models side by side in TouchCORE. For example, during software design, it makes sense to display the design class diagram together with a sequence diagram that specifies the behaviour of an operation defined in one of the classes. This functionality is covered by the *GenericSplitView* presented in the previous section.

When applying the RESTify concern, step 4 (see section on page 23) requires the user to specify mappings between the RAM class diagram and the ResTL model. Hence, we extend the functionality of the generic split view by incorporating mapping functionality with *GenericSplitViewWithMappings* and *GenericSplitViewWithMappingsHandler*. Similar to the *GenericSplitView* and *GenericSplitViewHandler*, *GenericSplitViewWithMappings* and *GenericSplitViewWithMappingsHandler* are abstract and generic Java classes and extend the original *view* and *view handler* respectively. Inter-model mappings are realized with a *COREModelElementMapping* model element and consists of a list of typically two arbitrary model elements, represented as *EObjects* from

[1] We chose to not use two scenes as input to the generic split view because the complexity of initializing a secondary scene can decrease the performance of TouchCORE.

Figure 6.1: Generic Split View Event Flow



the EMF plug-in, and a unique id. *GenericSplitViewWithMappings* visualizes the mappings by placing a line directly between the illustrated model elements and listens for notifications regarding the addition and removal of mappings to maintain consistency with the underlying models. Since individual views in the generic split view can be dragged and zoomed as a whole, some individual model elements can be outside of the scope in the displayed view. When one of the mapped model elements is not visible in the underlying view, a short, stippled line is visualized by the other visible mapped model element. When neither of the mapped model elements are visible, the mapping is not visualized.

GenericSplitViewWithMappingsHandler extends *GenericSplitViewHandler* to capture the creation of inter-model mappings with a *Unistroke* gesture from a model element from the inner-level view to a model element from the outer-level view. *GenericSplitViewWithMappingsHandler* contains an instance of the interface, *COREModelElementMappingsValidator* provides several functionalities: 1) validating new mappings given the current mappings, 2) maintaining the correctness of the current mappings, and 3) verifying whether the current mappings are valid for transformation. In particular, functionality 2) is called in several situations, including before the navigation to the split view, when either one of the underlying models are modified, and when mappings are removed to verify if there are any additional mappings to remove to maintain consistency with the underlying models and current mappings. The business logic of the *COREModelElementMappingsValidator* is to be defined by implementations of the generic split view with mappings module

since the validation of inter-model mappings depends on the semantics of the two models that are being displayed as well as the semantics of the perspective that connects them, if any. If the mapping contains a valid model element from both views, and is validated by the *COREModelElementMappingsValidator*, the mapping is then constructed with an EMF command using the *PerspectiveController*. Similar to the creation of inter-model mappings, mappings can also be removed by a *Unistroke* gesture, which checks for intersections with any of the current mapping lines. The removal of mappings is also executed by an EMF command with the *PerspectiveController*. After the removal of mappings, *COREModelElementMappingsValidator*is called to ensure the validity of the remaining mappings, and identify any additional inconsistent mappings, in which they are removed again with EMF commands. With multi-language perspectives, transformations can be designed for the purposes of code generation or model combination. Thus, before the execution of such transformations, the mappings are validated with the *COREModelElementMappingsValidator* to ensure that the underlying models and mappings are ready.

6.3 RAM-ResTL Split View with Mappings Functionality

In order to create mappings between the RAM and ResTL model elements from the BookStore concern, we introduce the RAM-ResTL perspective [Sch21b] that can interpret both the RAM and ResTL modelling languages. Now, we inherit and implement the generic split view with mappings functionality with *RamResTLSplitViewWithMappings* and *RamResTLSplitViewWithMappingsHandler*. In addition, we implement *COREModelElementMappingValidator* with *RamResTLMappingsValidator* to verify the validity of RAM-ResTL mappings. A visualization of the RAM-ResTL split view with mappings for the BookStore concern is shown in Figure 6.2. In the RAM-ResTL split view, REST endpoint to controller operation mappings and URI matching pattern to parameter mappings can be visualized as blue and green lines respectively. The REST endpoint to controller operation mapping connects a specific REST endpoint to the business logic of a controller operation in the RAM model representation of the original application. It therefore always must connect an *AccessMethod* model element in the ResTL model to an *Operation* model element in the RAM model. Similarly, URI matching pattern to parameter mappings connect a *DynamicFragment* model element in the ResTL model with a *Parameter* in the RAM model. The URI matching pattern to parameter mapping enables the application to capture the dynamic information stored within a specific section of the endpoint URI and use its value as a parameter of the method invocation in the controller operation.

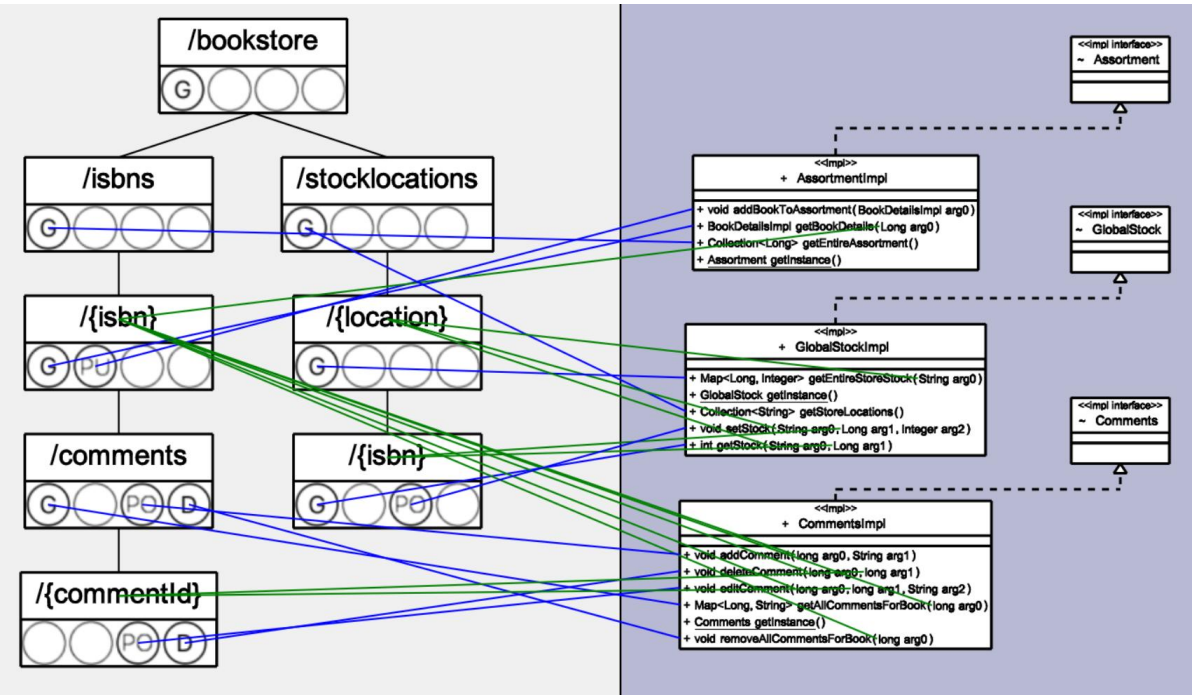
The *RamResTLMappingsValidator* provides validation for new mappings, current mappings, and whether the mappings are valid for transformations. For a new RAM-ResTL mapping, we first classify that the mapping is either an *AccessMethod* to *Operation* mapping or a *DynamicFragment*

to *Paramter* mapping. In the case that the new RAM-ResTL mapping is an *AccessMethod* to *Operation* mapping, we ensure that the mapped *AccessMethod* or *Operation* is not already present in a different mapping. Otherwise, in the case of a new *DynamicFragment* to *Parameter* mapping, we first ensure that the *Parameter* is of "primitive" type since the dynamic URI section can only capture values that can be determined by a sequence of characters. The accepted types for a *Parameter* include *String*, *int*, *Integer*, *long*, *Long*, *boolean*, *Boolean*, *float*, *Float*, *double*, *Double*, *char* and *Character* Java data types. Next, we validate that the *Parameter* does not already exist in a *DynamicFragment* to *Parameter* mapping. Additionally, we verify that the *Operation* of the newly mapped *Parameter* is already mapped in an *Operation* to *AccessMethod* mapping. This is to ensure that the *Operation* of the *Parameter* is associated with a REST endpoint which should contain the URI matching pattern. Finally, we verify that the newly mapped *DynamicFragment* is present on the URI path of the associated *Operation*. We validate this by recursively iterating through the newly mapped *DynamicFragment* and its children and verifying that the *PathFragment* of the corresponding *AccessMethod* exists as a child. This is to ensure that the URI matching pattern is present in the REST endpoint URI. For example, consider that a mapping exists from the */{commentId} DELETE AccessMethod* to the *void deleteComment(long arg0, long arg1) Operation* in the *CommentsImpl* controller class. A new mapping from */{location} DynamicFragment* to the *long arg0 Parameter* of the above *Operation* will not be validated since */{location}* is not a part of the */bookstore/isbns/{isbn}/comments/{commentId}* URI.

RamResTLMappingsValidator also provides validation for all the current RAM-ResTL mappings to identify inconsistent and incorrect mappings to be removed before the navigation to the split view, when either of the underlying models are modified, and when a mapping is removed from an *Unistroke* gesture. *RamResTLMappingsValidator* first ensures that the model elements are still present in the underlying views. Next, it validates the *DynamicFragment* to *Parameter* mappings by verifying that the corresponding *Operation* exists in an *AccessMethod* to *Operation* mapping, and that the *DynamicFragment* exists in the associated REST endpoint URI, similar to the verification process for a new RAM-ResTL mapping articulated above. Since *DynamicFragment* to *Parameter* mappings have a dependency to *AccessMethod* to *Operation* mappings, we need to constantly verify that the *AccessMethod* to *Operation* mappings exist and are valid.

Additionally, *RamResTLMappingsValidator* validates the RAM-ResTL mappings before the execution of RESTify transformations to ensure the validity and consistency of the mappings with the underlying models. This validation procedure incorporates all of the verification steps in the previous mappings validation, and additionally checks for the amount of unmapped *Parameters* in mapped *Operations*. In a RESTful web service, there can only exist at most one parameter in a controller operation with the appropriate HTTP Request Body annotation. During the RES-

Figure 6.2: RAM-ResTL Split View with Mappings for the BookStore Application

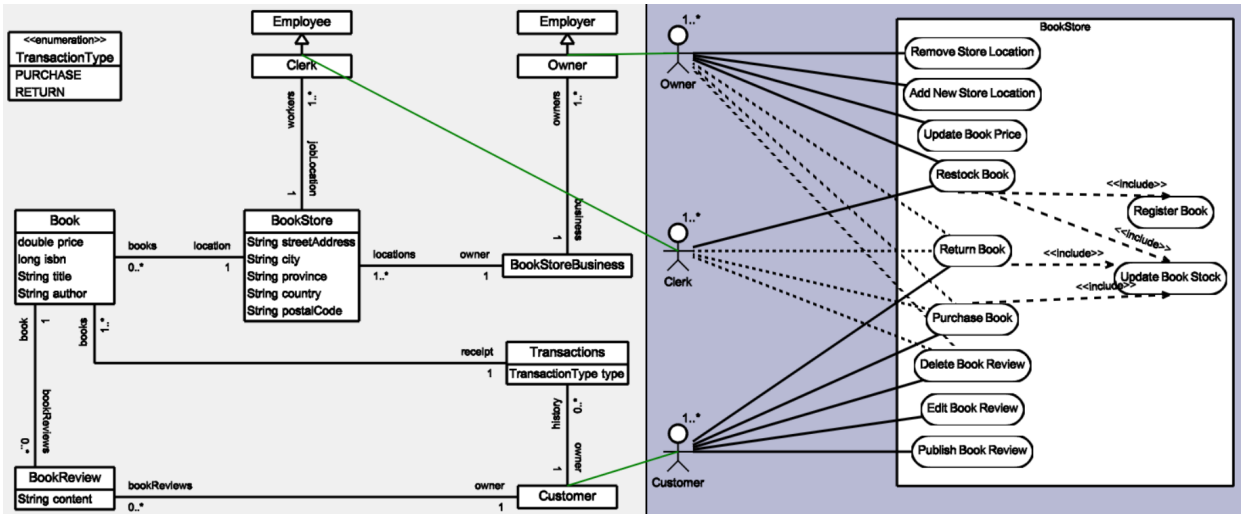


Tify transformation pipeline, we place HTTP Request Body annotations for the *first* unmapped *Parameter* in all mapped *Operations*. Therefore, it is incorrect to have more than one unmapped *Parameters* in mapped *Operations*, in which case we prevent the execution of the RESTify transformations.

6.4 Domain-Use Case Split View with Mappings Functionality

In this section, we present another implementation of the generic split view with mappings functionality to demonstrate the generic nature of the generic split view. We introduce a Domain-Use Case perspective [Ali21] that can interpret both the *Class Diagram* and *Use Case* modelling languages. In the Domain-Use case split view, implemented in the classes *DomainUseCaseSplitViewWithMappings* and *DomainUseCaseSplitViewWithMappingsHandler*, a *Class* model element from the Class Diagram model can be mapped to an *Actor* model element from the Use

Figure 6.3: Domain-Use Case Split View with Mappings for the BookStore Application



Case model and visualized as a green line. This Domain-Use Case mapping can serve as a consistency link between the two model elements. Additionally, we provide an empty implementation of *COREModelElementMappingsValidator* as *DomainUseCaseMappingsValidator* to avoid runtime errors.

Figure 6.3 illustrates the BookStore application in the Domain-Use Case perspective. The Domain model illustrates the relevant classes, attributes, relationships, and enumerations for a BookStore application. On the other side, the Use Case model highlights the potential actors who interact with the BookStore system and the various operations they can perform. Domain-Use Case mappings illustrate the consistency between the *Class* and *Actor* model elements. For example, in Figure 6.3, the *Customer Class* is linked to the corresponding *Customer Actor*.

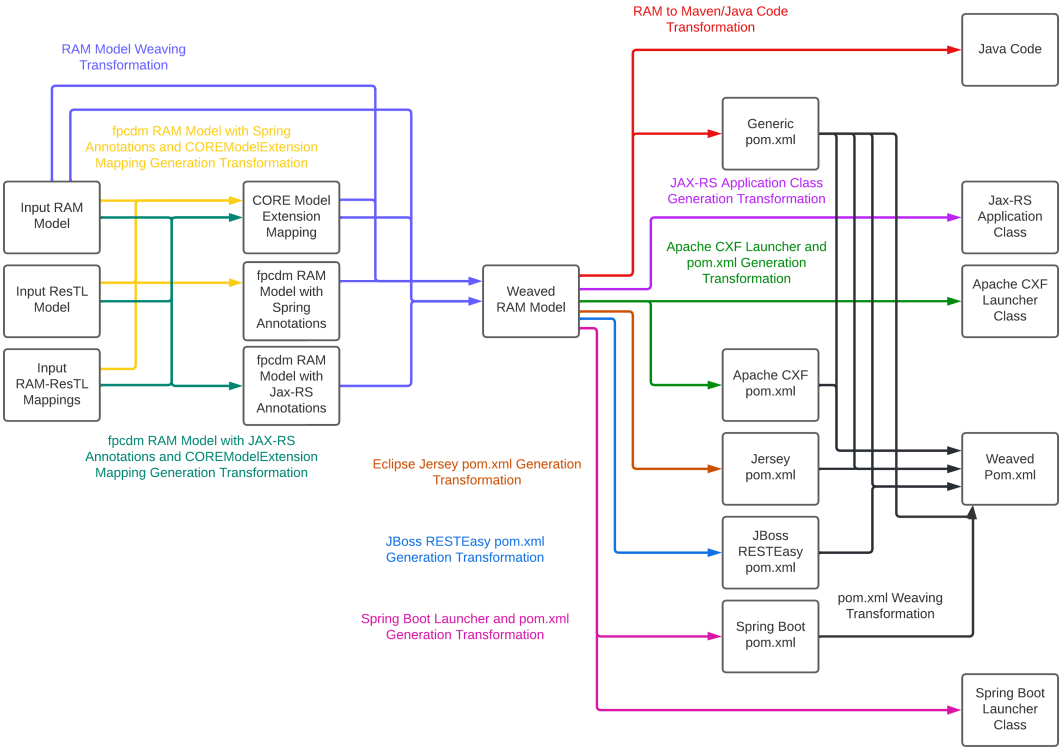
7
RESTify Transformation Pipeline

After the reuse of the RESTify concern, construction of the RAM and ResTL models in the RAM-ResTL perspective [Sch21b] and establishment of the RAM-ResTL mappings between the two models, the concern is now ready to execute a set of model-to-model, model-to-code and code-to-code transformations in order to generate a functional RESTful web service. The model-to-code transformations are implemented with the Acceleo Eclipse plugin to provide code generation functionality from EMF models. The inputs to the RESTify transformation pipeline are the RAM and ResTL models and the RAM-ResTL mappings between the models, whereas the output of the transformations include the Java source code and Maven configuration file, i.e., the `pom.xml` file.

Before the start of this thesis, there was already a *monolithic* model transformation that would take the RAM and ResTL models and the RAM-ResTL mappings between the models as input and generate a Java REST implementation based on Spring Boot. In this thesis, in order to support additional REST frameworks, we have decomposed the original monolithic transformation into several individual transformations that are executed by a model transformation pipeline. Furthermore, we added support for generating implementations based on three additional REST technologies, namely Eclipse Jersey, JBoss RESTEasy and Apache CXF. Thanks to the modularity provided by the new transformation pipeline, whenever the implementations exhibit commonalities among them, the individual transformations generating the common artefacts can now be reused effectively.

Figure 7.1 show the complete flowchart of the new transformation pipeline. Each arrow represents a model-to-model, model-to-code or code-to-code transformation executed during the RESTify transformation process. All transformations, except the blue one entitled RAM Model Weaving Transformation, have been implemented and tested as part of this thesis. The exact transformations to be executed when the RESTify user requests the code to be generated depend on the selected features, i.e., REST implementation technology, in the reused RESTify concern as illustrated in Table 7.1. We elaborate on the implementation details of each of the transformations in

Figure 7.1: Flowchart of the New RESTify Transformation Pipeline



the sections below.

7.1 fpcdm RAM Model with Spring Annotations and CORE-ModelExtension Mapping Generation Transformation

The Functionality Poor Class Diagram Model (fpcdm) RAM Model with Spring Annotations and COREModelExtension Mappings Generation Transformation is a model-to-model transformation that takes as inputs the RAM and ResTL models and the RAM-ResTL mappings and produces an intermediary fpcdm RAM model with Spring annotations [Sch21a]. This transformation is illustrated in yellow in Figure 7.1 and is executed for Spring Boot implementations. First, we create an empty RAM model named fpcdm and clone the *classes*, *operations*, *signature* and Maven artifact *dependencies* of the input RAM model. Next, we create empty controller *classes* and *operations* for each *ImplementationClass* and their *operations* from the input RAM Model that is referenced by a RAM-ResTL mapping in the provided mappings. Additionally, we create a

7.1 fpcdm RAM Model with Spring Annotations and COREModelExtension Mapping Generation Transformation

Table 7.1: Executed RESTify Transformations for Selected REST Technology

RESTify Transformation	Spring Boot	Eclipse Jersey	JBoss RESTEasy	Apache CXF
fpcdm RAM Model with Spring Annotations and COREModelExtension Mappings Generation Transformation	Y	N	N	N
fpcdm RAM Model with JAX-RS Annotations and COREModelExtension Mappings Generation Transformation	N	Y	Y	Y
RAM Model Weaving Transformation	Y	Y	Y	Y
RAM to Maven/Java Code Transformation	Y	Y	Y	Y
JAX-RS Application Class Generation Transformation	N	Y	Y	Y
Apache CXF Launcher and pom.xml Generation Transformation	N	N	N	Y
Eclipse Jersey pom.xml Generation Transformation	N	Y	N	N
JBoss RESTEasy pom.xml Generation Transformation	N	N	Y	N
Spring Boot Launcher and pom.xml Generation Transformation	Y	N	N	N
pom.xml Weaving Transformation	Y	Y	Y	Y

COREModelExtension to provide a mapping from the generated fpcdm RAM model to the input RAM model. The *COREModelExtension* mapping is required as an input for the RAM Model Weaving Transformation that is executed next in the pipeline.

After the creation of the fpcdm RAM model, initialization of its model elements, and definition of mappings between the fpcdm and input RAM models with a *COREModelExtension*, we proceed to the decoration step with Spring-specific *annotations*. For *annotations*, we also specify the *import* identifier for the RAM to Maven/Java Transformation to generate the required import statements. We begin by decorating the controller classes with @*RestController* and @*CrossOrigin* annotations. The @*CrossOrigin* annotation enables cross origin requests for a RESTful web service. Next, we place Spring endpoint *annotations* on the mapped *operations* from the input RAM-ResTL mappings. To elaborate, we place a @*GetMapping*, @*PutMapping*, @*PostMapping* or @*DeleteMapping* annotation on the controller *operation* based on the *MethodType* of the mapped *AccessMethod* model element and specify the URI path inside the annotation as the complete path of the associated *PathFragment*. Now, we extend the existing decoration procedure by implementing *parameter* annotations, namely @*PathVariable* and @*RequestBody*. We decorate a @*PathVariable* annotation on each mapped *parameter* in the input RAM-ResTL mappings, where the value of the annotation is the *placeholder* of the *DynamicFragment* of the mapping. For each *parameter* in controller *operations*, if there is an unannotated *parameter*, we decorate them with the @*RequestBody* annotation.

After the decoration of annotations in the fpcdm RAM model, we define the business logic in the controller *operations* with the *message* model elements. The first *message* retrieves the singleton instance of the underlying *class* from the input RAM model and stores it as a variable named *instance*. The second *message* delegates the business logic to the same operation in the *instance* variable. If the *return type* of the *operation* is not *void*, we store the result of the delegating *operation* as a variable and return it in the third *message*. Otherwise, we do not store the return of the delegated *operation* and there is no third *message*.

7.2 fpcdm RAM Model with JAX-RS Annotations and COREModelExtension Mapping Generation Transformation

The fpcdm RAM Model with JAX-RS Annotations and *COREModelExtension* Mapping Generation Transformation is a model-to-model transformation that inputs the RAM and ResTL models and the RAM-ResTL mappings and outputs an intermediary fpcdm RAM model with JAX-RS annotations. This transformation is illustrated in dark green in Figure 7.1 and is executed for all JAX-RS implementations. Similarly with the previous fpcdm RAM Model generation transformation, we create an empty fpcdm RAM model and clone the *classes*, *operations*, *signature* and

Maven artifact *dependencies* of the input RAM model. We create additional controller *classes* and *operations* for each *ImplementationClass* and their *operations* that are contained by a RAM-ResTL mapping from the input mappings. We establish a *COREModelExtension* to define a mapping from the newly created fpcdm RAM model and the input RAM model so that it can be processed later by the RAM model weaver.

The decoration of *annotations* step varies from the previous transformation in which we now incorporate JAX-RS *annotations* as opposed to Spring. We begin by decorating the controller classes with @*Path* annotations. Next, we create the appropriate JAX-RS endpoint *annotations* based on the input RAM-ResTL mappings. In particular, we place a @*GET*, @*PUT*, @*POST* or @*DELETE* annotation on the controller *operation* based on the *MethodType* of the mapped *AccessMethod*. Additionally, we decorate the controller *operation* with a @*Path* annotation with the complete path of the associated *PathFragment* of the mapped *AccessMethod*. We define a @*PathParam* annotation for each mapped *parameter* in the input RAM-ResTL mappings, where the value is the *placeholder* of the *DynamicFragment*. Now, for each controller *operation*, if there exists an unannotated *parameter*, we decorate the *operation* with a @*Consumes* annotation. We also verify the *return type* of each controller *operation*, if it is not *void*, we place a @*Produces* annotation for JSON payload encoding.

After the annotation decoration procedure, we link the controller *operations* to business logic with *message* model elements. Similar to the previous fpcdm generation transformation, the first *message* retrieves the singleton instance of the associated *class* in the input RAM model and store it as an *instance* variable. Next, the second *message* delegates the business logic to the identical operation from the *instance* variable. Now, we store the result of the delegated operation with a variable and return it in the third *message* if the *return type* of the controller *operation* is not *void*.

7.3 RAM Model Weaving Transformation

The RAM Model Weaving Transformation is a model-to-model transformation that takes as inputs the original RAM model, the intermediary fpcdm RAM model and the *COREModelExtension* mapping and outputs a woven RAM model that combines the model elements and properties of the two input models. This transformation is illustrated in dark blue in Figure 7.1 and is executed for all supported REST implementations. We name the original input RAM model as the *functionality rich class diagram model* (frcdm) as it represents the original application with *Implementation-Class* model elements and contains the business logic as specified by the original software system. The intermediary fpcdm RAM model contains empty clones for the *classes* and *operations* from the frcdm RAM model and additionally provides annotated controller classes with business logic defined by *message* model elements. Now, we input the fpcdm and frcdm RAM models to

the *COREWeaver* to produce a woven RAM model used for subsequent code generation transformations. The *COREWeaver* is an existing transformation for all CORE-related composition algorithms and is reused for the RAM Model Weaving Transformation.

7.4 RAM to Maven/Java Code Transformation

The RAM to Maven/Java Code Transformation is a model-to-code transformation that takes as input a RAM model (in our case the woven RAM model produced by the weaver) and outputs the corresponding Java source code and a `pom.xml` file for use with Maven. The transformation is illustrated in red in Figure 7.1 and is executed for all supported REST technologies. This generation transformation consists of two separate model-to-code transformations, but they are combined as a singular transformation since the Maven configuration and Java code transformations are used in all REST implementations. For the RAM to Maven Transformation, the output file name is *GenericPom.xml* to avoid being executed by Maven by accident. This also emphasizes that the generated *GenericPom.xml* is only a partial Maven specification. It consists of the *modelVersion*, *groupId*, *artifactId*, *version*, *packaging*, *properties*, and *dependencies* Maven tags. We specify the *modelVersion* as 4.0.0 to be compatible with Maven 3. For the *groupId*, *artifactId* and *version*, we extract the information dynamically from the *signature* of the *structureView* of the input RAM model. The *groupId*, *artifactId* and *version* are required fields to define the Maven coordinates of the underlying Maven project. For *packaging*, we use the default configuration with *jar*. The *packaging* format can in a later step of the transformation pipeline be overridden by the technology-specific `pom.xml` during the pom.xml Weaving Transformation if the selected REST technology requires a specific format. For *properties*, we specify the *project build encoding* to be UTF-8 and set the *source* and *target* Java version of the *Maven compiler* as 1.8. Now for the *dependencies* section, we dynamically append the *dependencies* contained in the *structuralView* of the input RAM model. In particular, there is a *dependency* to the original application as defined in the input RAM model. For the interested reader, the implementation of the RAM to Maven Transformation is presented in detail in the Appendix, in Figure B.1 as an Acceleo module.

The RAM to Java Code Transformation generates Java source code given the enumeration of the *classes* and *types* in the *structureView* of the input RAM model using the original RAM-to-Java generator provided by TouchCORE since 2012. It is important to note that the Java generator does not generate *ImplementationClass* model elements. For the *class* model elements in RAM models, we generate the *package* identifier, *import* statements, *class* definition, *attributes* and *operations* in a Java class file. Additionally, we decorate the *class*, *operations*, and operation *parameters* with the appropriate REST annotations from the model. For *operations*, we generate business logic with the enumerated *messages* in an *operation* model element. Now, for *type* model elements, we

generate the *package* identifier, *enumeration* type and individual *literal* values.

7.5 JAX-RS Application Class Generation Transformation

The JAX-RS Application class generation transformation is a model-to-code transformation that takes as an input the woven RAM model and outputs an implementation of the JAX-RS Application class. The transformation is illustrated in purple in Figure 7.1 and is executed for all JAX-RS implementations, including Eclipse Jersey, JBoss RESTEasy and Apache CXF. Specifically, we generate the *ApplicationConfig* class that extends *Application* and is decorated with the *@ApplicationPath* annotation. A majority of the elements in the Acceleo module are static, except for the package statement and enumeration of controller classes. We obtain the package identifier from the *groupId* and *artifactId* from the signature of the *structureView* of the woven RAM model. Similarly, we identify the controller classes by verifying the *controller* keyword from the list of *classes* in the *structureView* of the woven RAM model. The implementation of the JAX-RS Application Class Generation Transformation is shown in the Appendix, in Figure B.3.

7.6 Apache CXF Launcher and pom.xml Generation Transformation

The Apache CXF Launcher and pom.xml Generation Transformation is a model-to-code transformation that takes as input the woven RAM model and outputs the Apache CXF launcher and Apache CXF-specific `pom.xml` file. The transformation is illustrated in light green in Figure 7.1 and is executed for only Apache CXF implementations. This generation transformation actually consists of two separate model-to-code transformations, but they are combined as a singular transformation since the launcher and `pom.xml` are required for all Apache CXF implementations. In the Apache CXF launcher class, namely *ApacheCXFLauncher*, we define a main method to create the required endpoints as specified by the generated *ApplicationConfig* class from the previous transformation with a *JAXRSServerFactoryBean*, set its address to *http://localhost:8080/* and add a *JacksonJaxbJsonProvider* to provide encoding and decoding from JSON payload formats. Now, to run the Apache CXF application, we delegate the created *JAXRSServerFactoryBean* to the Apache CXF provided *Server* for deployment. The package name of the Apache CXF launcher Acceleo module is dynamic in which it retrieves the *groupId* and *artifactId* from the *signature* of the *structureView* of the input RAM model. The rest of the module is static to invoke the default Apache CXF launcher specifications. The implementation of the Apache CXF Launcher Generation Transformation is illustrated in the Appendix, in Figure B.4.

The Apache CXF pom.xml Generation Transformation outputs a file named *ApacheCXF-Pom.xml*. It includes the *packaging*, *properties*, *dependencies* and *build* Maven components. For

packaging, we specify the default *jar* format to execute the resulting *jar* with a simple Java command. Apache CXF applications generally define a launcher class and deploy the controller endpoints directly on the internal Apache CXF server. Thus, we do not need to package the source code as a *war* file and deploy to a Jetty web server, contrary to what needs to be done for the other JAX-RS implementations, such as Eclipse Jersey and JBoss RESTEasy (see the following subsections). For *properties*, we define the *apache cxf version* as 3.3.0 to maintain consistency with the various Apache CXF *dependencies*. The *dependencies* section includes the Apache CXF Runtime JAX-RS Frontend, Apache CXF Runtime HTTP Transport, Apache CXF Runtime HTTP Jetty Transport and Jackson JAX-RS JSON for handling JSON input and output for JAX-RS implementations using the standard Jackson data binding. The *build* section specifies the required *plugins* during the Maven build lifecycle, which includes the Apache Maven Dependency Plugin and Apache Maven JAR Plugin. The Apache Maven Dependency Plugin utility goals to work with dependencies in which we define the *copy dependencies* goal to be executed during the *packaging* step to ensure the output jar file contains the enumerated *dependencies*. The JAR Plugin provides the capability of building jars, and we specify the complete location of the *main class* to be the generated Apache CXF launcher in its *configuration*. Note that the complete location of the Apache CXF launcher class includes the dynamic package name, which includes the *groupId* and *artifactId* retrieved from the RAM input model as explained above. The implementation of the Apache CXF pom.xml Generation Transformation is illustrated in the Appendix, in Figure B.5 as an Acceleo module.

7.7 Eclipse Jersey pom.xml Generation Transformation

The Eclipse Jersey pom.xml Generation Transformation is a model-to-code transformation that takes as input the woven RAM model and outputs an Eclipse Jersey-specific `pom.xml` file. The transformation is illustrated in orange in Figure 7.1 and is executed only when the Eclipse Jersey feature of the RESTify concern is chosen. The output file name is *EclipseJerseyPom.xml* in order to prevent Maven from using this particular `pom.xml` file during deployment as it is incomplete and contains only Eclipse Jersey-specific configurations. The Eclipse Jersey `pom.xml` contains the *packaging*, *properties*, *dependencies* and *build* Maven tags. We specify the packaging format as *war* because Eclipse Jersey applications are generally packaged in the war format and deployed to a web server. We establish the *jersey version* to be 2.35 in *properties* to maintain consistency and avoid repetition in the *dependency version* elements in the *dependencies* tag. The *dependencies* tag includes the required Maven artifacts for Eclipse Jersey applications. To elaborate, the *dependencies* include the Jersey core server implementation, Jersey core servlet implementation, HK2 InjectionManager implementation and Jersey JSON Jackson entity providers support mod-

ule. Now the *build* section defines the required *plugins* for the build lifecycle including the Apache Maven WAR plugin and the Jetty web server. The Apache Maven WAR plugin is responsible for collecting all artifact dependencies, classes and resources of the web application and packaging them into a web application archive [Kin19]. In particular, we indicate that our generated source code does not include a web.xml by setting *failOnMissingWebXml* to false. Now, the packaged war file can be reliably deployed on the Jetty web server to run the Eclipse Jersey application. The implementation of the Eclipse Jersey pom.xml Generation Transformation is shown in the Appendix, in Figure B.6. The Acceleo module does not contain any dynamic aspects, but by the Acceleo specification, an EMF model must be provided as input.

7.8 JBoss RESTEasy pom.xml Generation Transformation

The JBoss RESTEasy pom.xml Generation Transformation is a model-to-code transformation that takes as input the woven RAM model and outputs a JBoss RESTEasy specific `pom.xml` file. The transformation is illustrated in light blue in Figure 7.1 and is executed only when the JBoss RESTEasy feature of RESTify is chosen. The output file name is *JBossRESTEasyPom.xml* to avoid being executed by Maven as it does not provide a complete Maven definition. The JBoss RESTEasy `pom.xml` contains the *packaging*, *properties*, *dependencies* and *build* specifications. We declare the *packaging* format as *war* since similar to Eclipse Jersey applications, JBoss RESTEasy applications are also typically packaged as a *war* file and deployed to a web server. For *properties*, we specify the *resteasy version* as 3.15.Final to include up-to-date and stable releases for the dependency artififacts. For *dependencies*, we include the RESTEasy servlet container initializer and RESTEasy Jackson provider. Now, the *build* component is identical to the *build* in the Eclipse Jersey `pom.xml` file, with the Apache Maven WAR plugin and the Jetty web server. The implementation of the JBoss RESTEasy pom.xml Generation Transformation is illustrated in the Appendix, in Figure B.7. The implemented Acceleo module is completely static as well and does not contain any dynamic segments.

7.9 Spring Boot Launcher and pom.xml Generation Transformation

The Spring Boot Launcher and pom.xml Generation Transformation is a model-to-code transformation that takes as input the woven RAM model and outputs the Spring Boot launcher class and Spring Boot-specific `pom.xml` file. The transformation is illustrated in pink in Figure 7.1 and is executed when the Spring Boot feature of RESTify is chosen. This generation transformation consists of two separate model-to-code-transformations; however, they are combined as a singular transformation since the launcher and `pom.xml` are required for all Spring Boot implementations.

Before the start of this thesis, the Spring Boot launcher was generated as a mandatory *class* in the intermediary fpcdm RAM model and its code then generated by the monolithic transformation. With the new transformation pipeline introduced by this thesis, we generate the launcher code directly using an Acceleo module. The generated Spring Boot launcher class is decorated with the *@SpringBootApplication* annotation and contains a main method to execute the launcher class as a Spring application. Additionally, the launcher class includes a package identifier which includes the *groupId* and *artifactId* from the *signature* of the *structureView* of the RAM input model. The implementation details of the Spring Boot Launcher Generation Transformation are illustrated in the Appendix, in Figure B.8.

Before the start of this thesis, the monolithic model-to-code transformation for Spring generated a `pom.xml` file with everything included. Now, with our new transformation pipeline, we have modularized the Spring Boot pom.xml Generation Transformation to output a *StringBoot-Pom.xml* file which only contains the Spring Boot-specific sections, including *parent*, *dependencies* and *build*. We declare the *parent* as the Spring Boot Starter Parent artifact which is a parent POM for providing dependency and plugin management for Maven applications. The *parent* tag enables inheritance to the parent POM contents. For *dependencies*, we include the Spring Boot Starter Web Maven artifact, which is a starter for building web applications, including RESTful with Spring MVC and uses Tomcat web server as the default embedded container. For the *build* section, we include the Spring Boot Maven Plugin artifact as a *plugin* and explicitly specify the full path to the Spring Boot launcher class in the *configuration*. The Spring Boot launcher class path includes the dynamic *groupId* and *artifactId* retrieved from the RAM input model. The implementation of the Spring Boot `pom.xml` is shown in the Appendix, in Figure B.9.

7.10 pom.xml Weaving Transformation

The pom.xml Weaving Transformation is a code-to-code transformation that takes as input the generated generic and technology-specific `pom.xml` files from the previous steps and outputs a woven `pom.xml` file to be used for Maven build configurations. The technology-specific `pom.xml` files include Spring Boot, Eclipse Jersey, JBoss RESTEasy and Apache CXF. The generated output has the key file name of `pom.xml` so that it can be executed during the Maven build lifecycle. Chapter 8 elaborates on the implementation and transformation details of the `pom.xml` weaving algorithm.

7.11 Generated Java/Maven Source Code

The complete generated Java/Maven source code for the supported REST frameworks: Spring Boot, Eclipse Jersey, JBoss RESTEasy, and Apache CXF for the BookStore application can be

seen in the Appendix. It is important to note that in order to compile the generated source code, the user must install the BookStore application in their local Maven repository with the following command, *mvn clean install*.

For Spring Boot, the woven `pom.xml` file is seen in Figure C.3. The Java source code annotated with Spring annotations for the REST controllers include *AssortmentImplController.java*, *CommentsImplController.java*, and *GlobalStockImplController.java* and they can be seen in Figures C.10, C.11, and C.12 respectively. Additionally, the Spring Boot launcher class named *SpringBootLauncher.java* can be seen in Figure C.16.

For the remaining JAX-RS implementations, the Java source code annotated with JAX-RS annotations for the REST controllers include again *AssortmentImplController.java*, *CommentsImplController.java*, and *GlobalStockImplController.java* and they can be seen in Figures C.13, C.14, and C.15 respectively. JAX-RS defines an application configuration class, and it can be seen in Figure C.17 named *ApplicationConfig.java*.

For Eclipse Jersey, the woven `pom.xml` file is seen in Figure C.5. There is no additional Java source code from the generated source code for JAX-RS implementations as specified above.

For JBoss RESTEasy, the woven `pom.xml` file is seen in Figure C.7. Similarly, the generated Java source code for JBoss RESTEasy does not contain any additional files from the generated source code for all JAX-RS implementations as specified above.

For Apache CXF, the woven `pom.xml` file is seen in Figure C.9. Apache CXF requires a launcher class, and it can be seen in Figure C.18 named *ApacheCXFLauncher.java*.

8

Weaving pom.xml Files

During the RESTify transformation pipeline, two Maven project configuration files, i.e., `pom.xml` files, are generated. POM stands for *Project Open Model* and is an XML representation of a Maven project held in a file named `pom.xml` [Red19]. In particular, we generate a generic and a technology-specific `pom.xml` file. The generic `pom.xml` file contains the POM model version, Maven coordinates with group ID, artifact ID and version, project packaging, Maven properties and a dependency to the original application. The technology-specific `pom.xml` file includes the project packaging, technology specific dependencies and build configurations. The project packaging is duplicated in the technology-specific `pom.xml` file since some implementation technologies require specific packaging formats for deployment. Each individual file is partial, i.e., it is missing some important information so that Maven can successfully compile and generate an executable.

This chapter describes a pom.xml weaving algorithm that takes as input two `pom.xml` files and produces as an output a woven `pom.xml` file that contains the union of the information provided in the two input files. For those elements where it is impossible to create a union, the element from the second file will take precedence. In summary, the weaving algorithm first sorts each XML element of the input `pom.xml` files and creates a merged XML element list in which the XML elements that exist only in one file are simply included, and the XML elements that exist in both of the input `pom.xml` files are woven together to be integrated as a singular XML element. The merged XML element list is then output as a new `pom.xml` file.

The weaving algorithm is based on the pom.xml schema file which defines the comprehensive structure and rules for `pom.xml` files to follow. Validation with the schema file is incorporated to verify the correctness of each XML element, sort the input `pom.xml` files for the purposes of weaving the XML elements, and merging XML elements at the correct level based on the schema guidelines.

8.1 pom.xml Schema File

The pom.xml schema file is located at `https://maven.apache.org/xsd/maven-4.0.0.xsd` and illustrates the configuration and specification for each XML element in `pom.xml` files. XML Schema Definition (XSD) files provide a standard method of validation of whether a given XML document conforms to it. Similar to a XML file, the schema file is also represented as a tree data structure. The first child of the root node from the schema file has an attribute named *project* and defines the root node of the `pom.xml` file. The *project* node has a *type* attribute of *Model* and is a link to the second child of the root node in the schema file that has a name attribute of *Model*. Now, the children of the *Model* node specify the accepted child nodes of the root node in `pom.xml` files. Each child of the *Model* node may define a *type* attribute, which can be classified into three categories - *primitive type*, *link to other complex type* and *none*. We can identify the primitive nature of the *type* attribute by validating if it starts with *xs:*. The most common *primitive type* in the schema file is *xs:string*. The *primitive type* attributes specify the data type of the accepted children of the node, in which they must be represented as text of the appropriate data format and not contain any additional children. When the *type* attribute exists and does not start with *xs:* it is a *link to other complex type*. Similar to how we navigated to the *Model* node from the *project* node, i.e., using the name of the type attribute, we can navigate from a *link to other complex type* node to the associated *complex type* node by directly examining the children of the root node of the schema document to find the child that has the same *name* as the *type* attribute of the *link to the other complex type* node. Now, the children of the linked complex type node provide in turn the specification of that XML element in the `pom.xml` file, and can further be navigated with the *type* attributes. When the *type* attribute does not exist in the child node, it belongs to the *none* category of *type* attributes. Nodes with *type* attributes belonging to the *none* category can only have one possible child and define it in the schema file as a child complex type node. The complex type node can define potential children, and their information can be found by navigating recursively based on its *type* attribute categories as elaborated above.

There are two additional attributes in the schema file that are used in the pom.xml weaving algorithm, namely *maxOccurs* and *processContents*. These attributes are used to find the appropriate level within the XML tree to merge nodes belonging to the same XML element. The *maxOccurs* attribute determines the multiplicity of a XML element in a `pom.xml` file, i.e., how many times that XML element can occur in the file. The default value for *maxOccurs* is one, but it can also be *unbounded*. For the purpose of merging two XML elements, we find the corresponding schema node in which the *maxOccurs* attribute is *unbounded* to combine the nodes at this particular level. In the schema file, similarly, the *processContents* attribute determines the validation process for a particular XML element in the input `pom.xml` file. In particular, when the value of the *process-*

Contents attribute is *skip*, the contents of this particular node and its children are arbitrary. Thus, schema nodes that contain the *processContents* attribute of *skip* are also an appropriate level to combine nodes.

Initially, the implementation of the pom.xml weaving algorithm retrieves the pom.xml schema file online from the Apache server. However, in order to maintain the offline modelling features in TouchCORE, the schema file, namely *maven-4.0.0.xsd*, is now downloaded and integrated as a resource in order to support the offline functionalities.

8.2 pom.xml Weaving Algorithm

The main pom.xml weaving algorithm is illustrated in Algorithm 8.1. With the Java built-in Document Object Model (DOM) Application Programming Interface (API), we can parse XML files as a tree data structure which represents XML elements as tree nodes. Similarly, the DOM API can be used to export a *Document* object to a file on the file system. For the main pom.xml weaving algorithm, we extract the direct children of the root project nodes of the input `pom.xml` files as a list and sort them with a variation of the classical merge sort algorithm as illustrated in Algorithm 8.2. Next, the two sorted lists are woven together into one merged list during the *merge* step of the algorithm by directly appending nodes that exist only in one of the lists, and by creating a merged node for elements that occur in both lists. We construct an empty *Document* object with the function *createEmptyDocument* and create the default root *project* node with *createRootMavenNode*. The root *project* node defines the fundamental properties including *xmlns*, *xmlns:xsi* and *xsi:schemaLocation*. We define *xmlns* as *https://maven-apache.org/POM/4.0.0* to elaborate the default namespace declaration. Similarly, we initialize *xmlns:xsi* as *http://www.w3.org/2001/XMLSchema-instance* to obtain the XML Schema Instance namespace and specify *xsi:schemaLocation* as *http://maven.apache.o* *http://maven.apache.org/xsd/maven-4.0.0.xsd* to provide a link from the namespace to the location of the associated XML schema. Finally, each node in the woven list is appended to the newly created *project* node and the new *Document* object is written to the disk as the output `pom.xml` file.

Algorithm 8.2 presents the main XML weaving algorithm. It is based on the merge sort algorithm, i.e., it sorts and merges the lists of the direct children of the root *project* node. The *merge* function retrieves numerical priority values from the pom.xml schema file with *getPriorityValueFromPomXmlSchema* to compare element nodes. The function *getPriorityValueFromPomXmlSchema* parses the schema file to locate the children of the *Model* complex type node, which specifies the accepted children of the *project* root node in `pom.xml` files. The priority value is assigned by the indices of each child of the *Model* complex type node that determines the ordering of the XML elements with the XML schema. When the priority values are equal, we combine the two element

8.2 pom.xml Weaving Algorithm

Algorithm 8.1 pom.xml Weaving - Main Algorithm

```
1:  function WEAVETWOXMLFILES(fileLocationOne, fileLocationTwo, wovenFileLocation)
2:      documentOne ← PARSEXMLFROMDISKTODOCUMENT(fileLocationOne)
3:      documentTwo ← PARSEXMLFROMDISKTODOCUMENT(fileLocationTwo)
4:      rootNodeOne ← documentOne.documentElement
5:      rootNodeTwo ← documentTwo.documentElement
6:      elementNodesOne ← empty array
7:      elementNodesTwo ← empty array
8:      for childNode in rootNodeOne.children do
9:          if childNode is ElementNode then
10:             add childNode to elementNodesOne
11:         end if
12:     end for
13:     for childNode in rootNodeTwo.children do
14:         if childNode is ElementNode then
15:             add childNode to elementNodesTwo
16:         end if
17:     end for
18:     sortedElementNodesOne ← MERGESORT(elementNodesOne)
19:     sortedElementNodesTwo ← MERGESORT(elementNodesTwo)
20:     sortedAndMergedElementNodes ← MERGE(sortedElementNodesOne, sortedElementNodesTwo)
21:     wovenDocument ← CREATEEMPTYDOCUMENT
22:     wovenDocumentRootNode ← CREATEROOTMAVENNODE(wovenDocument)
23:     for elementNode in sortedAndMergedElementNodes do
24:         append child elementNode to wovenDocumentRootNode
25:     end for
26:     append child wovenDocumentRootNode to wovenDocument
27:     WRITEDOCUMENTTODISKASXML(wovenDocument, wovenFileLocation)
28: end function
```

nodes into a singular node and use the resulting node in the sorting algorithm in the *merge* function. We implemented the merge sort algorithm in order to reuse the *merge* function after both input `pom.xml` documents are sorted. This can be illustrated in Algorithm 8.1 in line 20.

Algorithm 8.3 provides the pseudocode for merging two XML elements with equal priority values. It takes two element nodes as input. We first validate the first element node with the schema document with *isNodePrimitiveTypeAndMaxOccursIsOne*. This function navigates to the node definition of the input element node in the schema document and verifies the *type* and *maxOccurs* attributes of the schema node definition. When the *type* attribute is of *primtitive type*, i.e. it starts with *xs:*, and has a *maxOccurs* attribute of one, the schema node definition specifies that the result can only contain one *primitive type* child, which itself can only contain text and no recursive children. In this particular case, we keep the element node that was passed as the second parameter (and hence discard the first element node) since the second parameter will always be from the more specific `pom.xml` file. In our case this will be the technology-specific `pom.xml` file, hence we prioritize its specifications over the generic `pom.xml`. In the case when the schema node definition does not have a *primitive type type* attribute or *maxOccurs* of one, we determine the appropriate tree level to merge the two input nodes with the function *findCommonMergeNodes*. When *commonMergeNodes* exist, we can combine the input nodes by directly appending the children of *commonMergeNodeTwo* to *commonMergeNodeOne* and return the merged node. On the other hand, when *commonMergeNodes* does not exist, we simply return the second technology-specific node as the default behaviour.

Algorithm 8.4 illustrates a recursive algorithm to find the appropriate tree level to merge two XML element nodes based on the pom.xml schema file. We call the function *validateMergeNodesWithSchema* to verify if our current nodes can be used for the purposes of merging. In particular, *validateMergeNodesWithSchema* has three possible responses including *ContinueRecursing*, *FoundMergeNodes* and *MergeNodesDNE* (Merge Nodes Do Not Exist). In the case of *ContinueRecursing*, we ensure that both of the input nodes only have a singular element node as a child with equal names, and recursively call the *findCommonMergeNodes* function with the child nodes. The structure of XML elements in the `pom.xml` files is defined in which the top-level nodes only contain a singular element node as a child, whereas one of the lower-level nodes includes an *unbounded* amount of children. Correspondingly, we merge the input element nodes at the unbounded level. When the response from the schema validator is *FoundMergeNodes*, we simply return the current element node parameters as they are at the correct level for merging. Otherwise, when the response is *MergeNodesDNE*, we cannot perform the merge operation and return *null*.

Algorithm 8.5 provides validation of the current iterated element nodes for merging with the pom.xml schema file. In the schema document, we initialize the starting point for schema parsing

Algorithm 8.2 pom.xml Weaving - Merge Sort Algorithm

```
1:  function MERGESORT(elementNodes)
2:      elementNodesOne ← first half partition of elementNodes
3:      elementNodesTwo ← second half partition of elementNodes
4:      elementNodesOne ← MERGESORT(elementNodesOne)
5:      elementNodesTwo ← MERGESORT(elementNodesTwo)
6:      return MERGE(elementNodesOne, elementNodesTwo)
7:  end function

8:  function MERGE(elementNodesOne, elementNodesTwo)
9:      indexArrayOne, indexArrayTwo ← 0
10:     mergedElementNodes ← empty array
11:     while indexArrayOne < elementNodesOne.size and indexArrayTwo < elementNodesTwo.size do
12:         elementNodeOne ← elementNodes[indexArrayOne]
13:         elementNodeTwo ← elementNodes[indexArrayTwo]
14:         priorityOne ← GETPRIORITYVALUEFROMPOMXMLSCHEMA(elementNodeOne)
15:         priorityTwo ← GETPRIORITYVALUEFROMPOMXMLSCHEMA(elementNodeTwo)
16:         if priorityOne < priorityTwo then
17:             add elementNodeOne to mergedElementNodes
18:             indexArrayOne ← indexArrayOne + 1
19:         else if priorityOne > priorityTwo then
20:             add elementNodeTwo to mergedElementNodes
21:             indexArrayTwo ← indexArrayTwo + 1
22:         else
23:             mergedElementNode ← MERGEELEMENTNODES(elementNodeOne, elementNodeTwo)
24:             add mergedElementNode to mergedElementNodes
25:             indexArrayOne ← indexArrayOne + 1
26:             indexArrayTwo ← indexArrayTwo + 1
27:         end if
28:     end while
29:     while indexArrayOne < elementNodesOne.size do
30:         add elementNodeOne[indexArrayOne] to mergedElementNodes
31:         indexArrayOne ← indexArrayOne + 1
32:     end while
33:     while indexArrayTwo < elementNodesTwo.size do
34:         add elementNodeTwo[indexArrayTwo] to mergedElementNodes
35:         indexArrayTwo ← indexArrayTwo + 1
36:     end while
37:     return mergedElementNodes
38: end function
```

8.2 pom.xml Weaving Algorithm

Algorithm 8.3 pom.xml Weaving - Merge Element Nodes Algorithm

```
1:  function MERGEELEMENTNODES(elementNodeOne, elementNodeTwo)
2:      if ISNODEPRIMITIVETYPEANDMAXOCCURSISONE(elementNodeOne) then
3:          return elementNodeTwo
4:      else
5:          nodeNamesList ← empty list
6:          commonMergeNodes ← FINDCOMMONMERGENODES(elementNodeOne, elementNodeTwo, nodeNamesList)
7:          if commonMergeNodes is not null then
8:              commonMergeNodeOne ← commonMergeNode[0]
9:              commonMergeNodeTwo ← commonMergeNode[1]
10:             mergedNode ← elementNodeOne
11:             for node in elementNodeTwo.children do
12:                 if node is ElementNode then
13:                     if commonMergeNodeOne not contains node then
14:                         append child node to commonMergeNodeOne
15:                     end if
16:                 end if
17:             end for
18:             return mergedNode
19:         else
20:             return elementNodeTwo
21:         end if
22:     end if
23: end function
```

Algorithm 8.4 pom.xml Weaving - Find Common Merge Node Algorithm

```
1:  function FINDCOMMONMERGENODES(elementNodeOne, elementNodeTwo, nodeNamesList)
2:      add elementNodeOne.nodeName to nodeNamesList
3:      mergeNodesResponse ← VALIDATEMERGENODESWITHSCHEMA(elementNodeOne,
    elementNodeTwo, nodeNamesList)
4:      if mergeNodesResponse is ContinueRecursing then
5:          if NUMOFELEMENTNODECHILDREN(elementNodeOne) is 1 and NUMOFELEMENTNODECHIL-
    DREN(elementNodeTwo) is 1 and elementNodeOne.nodeName is elementNodeTwo.nodeName then
6:              newElementNodeOne ← GETFIRSTELEMENTNODECHILD(elementNodeOne)
7:              newElementNodeTwo ← GETFIRSTELEMENTNODECHILD(elementNodeTwo)
8:              return FINDCOMMONMERGENODES(newElementNodeOne, newElementNodeTwo,
    nodeNamesList)
9:          end if
10:     else if mergeNodesResponse is FoundMergeNodes then
11:         nodeList ← new array with elementNodeOne, elementNodeTwo
12:         return nodeList
13:     end if
14:     return null
15: end function
```

at the *Model* complex type node with the function *getSchemaAcceptedNodesChild*. The following for loop iterates through the *nodeNamesList* to streamline the navigation to the schema node definition for the input element nodes. The navigation process is performed by first initializing *currentSchemaNode* to the associated child of the *Model* complex type node as specified by the first name in *nodeNamesList* and and then navigating by finding the *link to other complex type* nodes with the *type* attribute. Now, we find the *typeAttributeCategory* of the *currentSchemaNode* to verify if the schema node definition validates merging at this particular level. When the *typeAttributeCategory* is *None*, the inner schema node provides the definition of the corresponding XML element in `pom.xml` implementations. We can find the *commonChildNodeName* with the function *findCommonChildNodeName,* which attempts to identify a unified name among all of the children of the two input nodes. If the *commonChildNodeName* exists, we can verify if the *maxOccurs* attribute of the inner node definition is *unbounded*, in which case, the input nodes are at the correct level to merge and return *FoundMergeNodes*. On the other hand, if the *commonChildNodeName* does not exist, the input nodes are at the correct level to merge, and we return *FoundMergeNodes* if the *processContents* attribute of the only inner node definition is *skip* and the *maxOccurs* attribute is *unbounded*. When the *processContents* attribute is *skip*, there are no rules in the schema node definition for its children in `pom.xml` implementations, in which they can be any arbitrary element node. Thus, we can merge the input nodes when the *processContents* attribute is *skip* and the *maxOccurs* attribute is *unbounded*. When the *typeAttributeCategory* is *LinkToOtherComplexType*, we return *ContinueRecursing* if the *maxOccurs* attribute of the *currentSchemaNode* is *one*, since the current input nodes are not at the merging level, but their children can be. Finally, the default behaviour for the function *validateMergeNodesWithSchema* returns *MergeNodesDNE*.

8.3 Example with Generated BookStore pom.xml Files

To illustrate our pom.xml weaving algorithm we present in this section an example where we merge the generated BookStore `pom.xml` files when the Eclipse Jersey feature is selected in the RESTify concern. The generic `pom.xml` file produced by the RAM to Maven/Java Code Transformation presented in Section 7.4 is shown in Figure 8.1. The generic `pom.xml` defines the following XML elements as children to the *project* root node: *modelVersion*, *groupId*, *artifactId*, *version*, *packaging*, *properties* and *dependencies*. Similarly, the technology specific Eclipse Jersey `pom.xml` file generated by the Eclipse Jersey pom.xml Generation Transformation presented in Section 7.7 is shown in Figure 8.2 and defines the following XML elements as children to the *project* root node - *packaging*, *properties*, *dependencies* and *build*. The two `pom.xml` files are already well-structured and in the correct order according to the pom.xml schema. During execution of the pom.xml weaving algorithm, we merge the *packaging*, *properties* and *dependencies* nodes.

Algorithm 8.5 pom.xml Weaving - Validate Common Merge Node with pom.xml Schema Algorithm

```
1: function VALIDATEMERGENODESWITHSCHEMA(elementNodeOne, elementNodeTwo, nodeNamesList)
2:     currentSchemaNode ← null
3:     schemaAcceptedNodesChild ← GETSCHEMAACCEPTEDNODESCHILD
4:     for i ← to 0,nodeNamesList.size do
5:         nodeName ← nodeNamesList[i]
6:         if i is 0 then
7:             currentSchemaNode ← GETCHILDBYNAMEATTRIBUTE(schemaAcceptedNodesChild, nodeName)
8:         else
9:             if currentSchemaNode ← is not null then
10:                typeAttribute ← GETTYPEATTRIBUTE(currentSchemaNode)
11:                typeAttributeCategory ← GETTYPEATTRIBUTECATEGORY(typeAttribute)
12:                if typeAttributeCategory is LinkToOtherComplexType then
13:                    currentSchemaNode ← GETCOMPLEXTYPENODEBYATTRIBUTENAME(typeAttribute)
14:                    if currentSchemaNode is not null then
15:                        currentSchemaNode ← GETCHILDBYNAMEATTRIBUTE(schemaAcceptedNodesChild, nodeName)
16:                    end if
17:                end if
18:            end if
19:        end if
20:    end for
21:    if currentSchemaNode is not null then
22:        typeAttribute ← GETTYPEATTRIBUTE(currentSchemaNode)
23:        typeAttributeCategory ← GETTYPEATTRIBUTECATEGORY(typeAttribute)
24:        if typeAttributeCategory is None then
25:            commonChildNodeName ← FINDCOMMONCHILDNODENAME(elementNodeOne, elementNodeTwo)
26:            if commonChildNodeName is not empty then
27:                innerNodeDefinition ← GETINNERNODEDEFINITIONOF(currentSchemaNode,
   commonChildNodeName)
28:                if innerNodeDefinition is not null then
29:                    if GETMAXOCCURSATTRIBUTE(innerNodeDefinition) is "unbounded" then
30:                        return FoundMergeNodes
31:                    end if
32:                end if
33:            else
34:                innerNodeDefinition ← GETONLYINNERNODEDEFINITIONOF(currentSchemaNode)
35:                if innerNodeDefinition is not null then
36:                    if GETPROCESSCONTENTSATTRIBUTE(innerNodeDefinition) is "skip" and
   GETMAXOCCURSATTRIBUTE(innerNodeDefinition) is "unbounded" then
37:                        return FoundMergeNodes
38:                    end if
39:                end if
40:            end if
41:        else if typeAttributeCategory is LinkToOtherComplexType then
42:            if GETMAXOCCURSATTRIBUTE(currentSchemaNode) is 1 then
43:                return ContinueRecursing
44:            end if
45:        end if
46:    end if
47:    return MergeNodesDNE
48: end function
```

8.3 Example with Generated BookStore pom.xml Files

The Eclipse Jersey `pom.xml` file contains more specific information than the generic `pom.xml` file. Because we want the algorithm to favour information from Eclipse Jersey over the generic information, we pass the generic `pom.xml` as the first argument and the Eclipse Jersey `pom.xml` as a second argument to the weaver. The resulting woven `pom.xml` file is visualized in Figure 8.3.

The *packaging* schema definition has a *type* attribute of *xs:string* and a *maxOccurs* attribute of *one*. Since there can only exist a singular child of string type for *packaging*, the algorithm chooses the one from the second file, in our case the Eclipse Jersey `pom.xml`, which has the value *war*. Therefore, the *packaging* node from the Eclipse Jersey `pom.xml` file is prioritized and used in the woven `pom.xml` output.

The *properties* schema definition does not define a type attribute, thus it has a *type* attribute category of *none*. When the *type* atrribute category is *none*, the schema node provides its inner node definition as a child. We can validate that the inner node definition has a *processContents* attribute of *skip* and a *maxOccurs* attribute of *unbounded*. Thus, we can merge at the *properties* node level in which the *properties* node in the woven `pom.xml` file contains *properties* nodes from both the generic and Eclipse Jersey `pom.xml` files.

The *dependencies* schema definition does not define a *type* attribute and has a *type* attribute category of *none*. In the case of the *dependencies* schema node, it contains an inner node definition as a child with the *name* attribute *dependency*. Now, the *maxOccurs* attribute of the *dependency* schema node is *unbounded*, thus we can merge at the *dependencies* node level. Therefore, the *dependencies* node from the woven `pom.xml` output file contains dependency nodes from both the generic and Eclipse Jersey `pom.xml` files.

The other nodes that only occur in one of the input `pom.xml` files are appended in the appropriate order, as shown in the woven `pom.xml` file in Figure 8.3. The woven `pom.xml` file has the name `pom.xml` so that it can be used for the Maven build lifecycle for the generated RESTful web service application.

The complete generic, technology-specific, and woven `pom.xml` files for all supported REST frameworks: Spring Boot, Eclipse Jersey, JBoss RESTEasy, and Apache CXF can be seen in the Appendix. The generic `pom.xml` is seen in Figure 8.1 and is reused as an input for the remaining weaving transformations. For Spring Boot, the technology-specific `pom.xml` is in Figure C.2, and the woven `pom.xml` is in Figure C.3. We have already illustrated the technology-specific, and woven `pom.xml` files for Eclipse Jersey in the above example. For JBoss RESTEasy, the technology-specific `pom.xml` is in Figure C.6, and the woven `pom.xml` is in Figure C.7. For Apache CXF, the technology-specific `pom.xml` is in Figure C.8, and the woven `pom.xml` is in Figure C.9.

8.3 Example with Generated BookStore pom.xml Files

Figure 8.1: Generic pom.xml for BookStore Concern

```
▼<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>
   <groupId>ca.mcgill.sel.restified</groupId>
   <artifactId>BookStore</artifactId>
   <version>1.0.0</version>
   <packaging>jar</packaging>
  ▼<properties>
     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
     <maven.compiler.source>1.8</maven.compiler.source>
     <maven.compiler.target>1.8</maven.compiler.target>
   </properties>
  ▼<dependencies>
    ▼<dependency>
       <groupId>eu.kartoffelquadrat</groupId>
       <artifactId>bookstoreinternals</artifactId>
       <version>1.2</version>
     </dependency>
   </dependencies>
 </project>
```

8.3 Example with Generated BookStore pom.xml Files

Figure 8.2: Eclipse Jersey pom.xml for BookStore Concern

```
▼<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
   <packaging>war</packaging>
  ▼<properties>
     <jersey.version>2.35</jersey.version>
   </properties>
  ▼<dependencies>
   ▼<dependency>
      <groupId>org.glassfish.jersey.core</groupId>
      <artifactId>jersey-server</artifactId>
      <version>${jersey.version}</version>
    </dependency>
   ▼<dependency>
      <groupId>org.glassfish.jersey.containers</groupId>
      <artifactId>jersey-container-servlet</artifactId>
      <version>${jersey.version}</version>
    </dependency>
   ▼<dependency>
      <groupId>org.glassfish.jersey.inject</groupId>
      <artifactId>jersey-hk2</artifactId>
      <version>${jersey.version}</version>
    </dependency>
   ▼<dependency>
      <groupId>org.glassfish.jersey.media</groupId>
      <artifactId>jersey-media-json-jackson</artifactId>
      <version>${jersey.version}</version>
    </dependency>
   </dependencies>
  ▼<build>
   ▼<plugins>
    ▼<plugin>
       <groupId>org.apache.maven.plugins</groupId>
       <artifactId>maven-war-plugin</artifactId>
       <version>3.3.2</version>
      ▼<configuration>
         <failOnMissingWebXml>false</failOnMissingWebXml>
       </configuration>
     </plugin>
    ▼<plugin>
       <groupId>org.eclipse.jetty</groupId>
       <artifactId>jetty-maven-plugin</artifactId>
       <version>9.4.43.v20210629</version>
     </plugin>
   </plugins>
  </build>
</project>
```

Figure 8.3: Woven pom.xml for BookStore Concern

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>ca.mcgill.sel.restified</groupId>
  <artifactId>BookStore</artifactId>
  <version>1.0.0</version>
  <packaging>war</packaging>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
    <jersey.version>2.35</jersey.version>
  </properties>
  <dependencies>
    <dependency>
      <groupId>eu.kartoffelquadrat</groupId>
      <artifactId>bookstoreinternals</artifactId>
      <version>1.2</version>
    </dependency>
    <dependency>
      <groupId>org.glassfish.jersey.core</groupId>
      <artifactId>jersey-server</artifactId>
      <version>${jersey.version}</version>
    </dependency>
    <dependency>
      <groupId>org.glassfish.jersey.containers</groupId>
      <artifactId>jersey-container-servlet</artifactId>
      <version>${jersey.version}</version>
    </dependency>
    <dependency>
      <groupId>org.glassfish.jersey.inject</groupId>
      <artifactId>jersey-hk2</artifactId>
      <version>${jersey.version}</version>
    </dependency>
    <dependency>
      <groupId>org.glassfish.jersey.media</groupId>
      <artifactId>jersey-media-json-jackson</artifactId>
      <version>${jersey.version}</version>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-war-plugin</artifactId>
        <version>3.3.2</version>
        <configuration>
          <failOnMissingWebXml>false</failOnMissingWebXml>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.eclipse.jetty</groupId>
        <artifactId>jetty-maven-plugin</artifactId>
        <version>9.4.43.v20210629</version>
      </plugin>
    </plugins>
  </build>
</project>
```

9 Related Work

In recent years, there has been an increasing interest for model-driven engineering solutions for RESTful web services. To the best of our knowledge, there has not been a model-driven engineering approach that incorporates CORE to provide a modelling interface and transformation pipeline to output a fully functional and ready-to-deploy RESTful web service based on one of the supported REST implementations, namely Spring Boot, Eclipse Jersey, JBoss RESTEasy, and Apache CXF. However, there has been significant work in model-driven approaches to express REST with modelling languages and generation of RESTful web services with transformation pipelines. In this chapter, we elaborate these areas of research and explain the differences with our approach.

9.1 Expressing REST with Modelling Languages

Schreier [Sch11] presents the first version of a REST metamodel to enable modelling and offer a vocabulary for REST in practice and the basis for model-driven development. Their REST metamodel can be divided into *structural* and *behavioural* parts. Structural modelling describes the possible resource types, their attributes, and relations as well as their interface and representations. *Representations* specify the payload format of the data received and sent by the server. In their proposed structural model, they define the root model element as *ResourceElement* which includes a list of resources, uniquely identified by a complete resource URI with a single model element, the enumerated HTTP request methods, their input and output payload formats, and the associated attributes and parameters. Behavioural modelling offers the possibility to describe the behaviour of the REST interface with deterministic finite-state machines. This offers the possibility to represent the current resource state and to define how a resource reacts to a certain request. A limitation to behavioural modelling is that it does not support modelling representation details, which is planned for future versions.

The work of Schreier [Sch11] introduces *representations* in the structural model to represent the input and output payload formats, in our RESTify approach, we do not model payload formats

in the ResTL metamodel. Instead, we assume that JSON is the desired data format and generate JSON specifications in the resulting RESTful web service. Schreier models resource URIs with a single model element to capture the entire URI, whereas we separate the entire URI into fragments in the ResTL metamodel separated by the forward slash symbol with a tree data structure. The advantage of our approach is that it provides a more intuitive method to visualize and design a REST resource tree. Furthermore, it provides a concrete method to establish links between individual *DynamicFragments* and *Parameters* in the generic split view, as opposed to Schreier's method of directly associating the complete URI to multiple parameters. Finally, we do not model the behaviour of REST resources in the REST metamodel as Schierer's behavioural model with deterministic finite-state machines. Instead, we delegate the behaviour to the UML sequence diagrams defined in the RAM metamodel.

Porres and Rauf [PR11] incorporate UML class and protocol diagrams to model the structural and behavioural interface of RESTful web services respectively. The UML class diagram illustrates a *conceptual resource model* to show different resources and how they can be addressed. The class diagram describes the composition of the RESTful web service domain with *classes*, *attributes*, and *operations* and defines the REST URI resource paths by enumerating the static or dynamic fragments in the *association* (between classes) names. The UML protocol diagram depicts a *behavioural model* to specify how the service should be used by showing the order of method invocations and the conditions on these methods. The protocol diagram specifies the exact HTTP request method for a given REST resource URI and elaborate its behaviour. The UML class and protocol diagrams can be used together to generate a *contract* in the form of preconditions and postconditions for methods of an interface. The generated contracts can be included in a Web App Description Language (WADL) interface specification to enrich its behaviour. The resulting service is RESTful by construction.

The work of Porres and Rauf [PR11] separates the resource URI in UML class diagrams, and HTTP request methods in UML protocol diagrams. A limitation with this approach is that it can confusing to contain REST resource URIs in a structural model (UML class diagram), similarly with HTTP request methods in a behavioural model (UML protocol diagram). This violates the principles of separation of concerns and loose coupling. Our approach differs from their approach by defining our own domain-specific modelling language, namely ResTL, to describe the REST resource interface (including both URI resource paths and HTTP request methods) with a tree data structure. Furthermore, we incorporate the RAM model to describe only the structural aspects of the original input software system and provide a method of connection between the RAM and ResTL models with mappings in the generic split view.

Schroth [Sch13] proposes RESTModelling, which combines REST and Model-Driven Soft-

ware Development (MDSD). The huge benefit from combining these two technologies is that constraints which have to be followed by the developers can now be fulfilled by correct and formal model definitions as well as the predefined model-to-model translations. In case a model is valid for a transformation, we can assume that the outcome is as RESTful as the transformation process alters it. The platform independent and platform specific modelling languages introduced in his thesis are realized with Ecore from Eclipse Eugenia. RESTModelling includes several metamodels, including *domain*, *gen*, *resource*, *deployment*, *JAX-RS* and *HTML doc*. The *domain model* captures the *objects* of the domain of the RESTful web service. The *gen model* is a domain model enhanced with user input which will be transformed into the resource model. Thus, there is no direct transformation between domain and resource model. The *resource model* enumerates a list of named *resources* in which each *resource* contains *methods* to define specific HTTP request method, media type, and return type properties and *parameters* to specify HTTP parameters. The *deployment model* provides a generic basis for the URI structure of the application interface by mapping resources to a single user specific link. The platform specific models including *JAX-RS* and *HTML doc models* are required for transformations of the application abstraction models into a specific desired piece of code. Schroth's approach incorporates these models to execute transformations to generate a complete HTML documentation and several JAX-RS stubs that can be incorporated as a basis for a RESTful web service. Petersohn [Pet14] faces the conceptional issues when modelling and generating a RESTful web service with the existing RESTModelling approach from Schroth [Sch13]. Petersohn extends the platform independent models including *domain*, *resource*, and *deployment* models and the platform specific models including *JAX-RS* and *HTML doc* models by introducing new model elements to provide a more accurate method of modelling them.

The proposed RESTModelling models by Schroth [Sch13] separates different aspects of the REST endpoint in two models, namely resource and deployment models. The *resource* model contains the HTTP request methods, while the *deployment* model maps resources to specific URIs. In our RESTify approach, we provide the ResTL modelling language to express REST resource endpoints and contains model elements for both HTTP request methods and URI fragments. It is important to note that we partition the REST resource URI paths into static and dynamic fragments by the forward slash character, resulting in a tree data structure, as opposed to modelling resource URIs as a single model element in RESTModelling. The benefit of partitioning URI paths is that it provides a clear connection between *DynamicFragments* and *Parameters* in the generic split view. RESTModelling specifies various input and output payload data formats in the *resource* model, where in the RESTify approach, we always assume that the data format is in JSON and generate encoding and decoding functionalities based on that assumption. The generated output of RESTModelling include a complete HTML documentation of the RESTful web service and JAX-

RS stubs that need to be integrated to execute correctly. On the other hand, RESTify generates the complete source code of the RESTful web service that can simply just run. Furthermore , we support the generation of three JAX-RS implementations as well, including Eclipse Jersey, JBoss RESTEasy, and Apache CXF.

Kharisma and Mardiyanto [KM20] introduce a text-based modelling language, namely BeREST, for application development with the REST architectural style. The grammar-based implementation of the BeREST language is based on the Xtext development environment `https://www.eclipse.org/Xtext/`. The BeREST metamodel identifies a resource with complete URI identifiers, HTTP request method types and various payload representation formats. Furthermore, the authors illustrate the code generation functionality for BeREST, namely BeRESTDB, which is essentially BeREST with a specialization for executing CRUD-based operations over a REST interface on a relational database. The target language of the code generator is Sinatra, a domain specific language implemented in Ruby that is used for writing web applications. Essentially, Sinatra provides pre-written methods that developers can include in their applications to turn them into Ruby web applications. The challenge to this approach lies in the code generator in the behavioural modelling aspect, especially the business logic since it is very difficult to generalize the various business logic at a high level of abstraction.

The BeREST metamodel proposed by Kharisma and Mardiyanto [KM20] incorporates complete URI identifiers for resource URIs. In our ResTL metamodel, we use fragments of the URI to separate each segment by the forward slash character and distinguish the static and dynamic fragments. A limitation to the BeREST metamodel is that it does not provide parameters associated with the URI identifiers, thus the dynamic fragments are simply wildcards, and its contents are not captured in the application. In our RESTify approach, we can establish mappings between *DynamicFragments* and *Parameters* in the generic split view. The BeREST metamodel includes the PATCH HTTP request method whereas the ResTL metamodel does not. A PATCH request specifies a set of instructions on how to modify a resource. This is not included in the ResTL metamodel because this behaviour is similar to the more commonly used PUT request method which simply updates the current resource with the request payload. The BeREST metamodel payload representation formats, whereas the RESTify approach assumes that JSON is the most commonly used data format and uses it in the generated RESTful web service. Another limitation with their approach is that their modelling abstract level is too high, thus the code generation is difficult to implement. With RESTify, we elaborate the business logic of the application to the UML sequence diagram in RAM models, where a message corresponds to a single line of code, thus making it easier to implement the Java code generator.

Laitkorpi et al. [LSS09] propose a model-driven process with a series of model transformations

to design RESTful web services. The proposed process is partitioned into five phases, including *analysis*, *behavioural canonicalization*, *structural canonicalization*, *service translation*, and *code generation*. During *analysis*, a functional specification is elaborated between a client and the service with top-level interactions expressed as UML sequence diagrams. The analysis models additionally contain two inter-linked views on the problem domain, namely business state of the service and the high-level classes representing the domain vocabulary. *Behavioural canonicalization* breaks down the functional specification, harvests relevant pieces of state information and their relationships, and selects uniform operation primitives to manipulate the state information, comprising an *information model*. *Structural canonicalization* transforms the information model into a *resource model* by introducing externally accessible entities as a resource abstraction layer based on the state information content harvested in the previous phase. *Service translation* generates the final resource hierarchy and other technical aspects by using the semantics of the target architecture, resulting in a *service specification*. *Code generation* transforms the service specification into various software artifacts for service providers and consumers. The authors currently assume that the service specification is in a machine processable form and did not provide a code generator.

Since Laitkorpi et al. [LSS09] do not provide the metamodel for their proposed resource model to represent REST resource endpoints in their paper, we cannot compare our ResTL metamodel with it. However, their proposed process to design RESTful web services with the five phases is completely different from our RESTify approach. The objective of their method is to design the architecture of a RESTful web service from scratch. On the contrary, the objective of RESTify is to extend an existing software system with MDE and CORE techniques to a functional and ready-to-deploy RESTful web service. In our RESTify approach, the design of the RESTful web service is done by constructing a ResTL model, and inter-model mappings between the RAM and ResTL models.

Kenzi and Yakine [KY21] propose a model-driven framework for the design and development of highly adaptable RESTful services. The core building blocks of this framework is a Unified Modelling Language profile called RESTVSoaML, and its associated tool support RESTVSoaML-Tool. RESTVSoaML aims the modelling of adaptable Restful Web services regardless of standards and implementation platforms. The RESTVSoaML metamodel specify a particular REST endpoint with a single model element which encompasses both the complete path of the resource URI as a String and a specific HTTP request method. Now the REST endpoint model element can be associated with numerous parameters. Furthermore, the metamodel specifies the accepted input and output data formats for payloads. RESTVSoamLTool is an MDD tool that enables the generation of code by using a model transformation language, from high level models defined with RESTV-

SoaML. In particular, it permits the generation of the description of each RESTFUL service and its implementation as WADL descriptions.

The RESTVSoaML metamodel proposed by Kenzi and Yakine [KY21] uses a single model element to represent a REST endpoint. The limitations of this approach are that the same resource URI path can be duplicated unnecessarily for REST endpoints with a different HTTP request method and using the complete resource URI path as an identifier in the model element can make it difficult to match the associated parameters to the dynamic sections of the URI path. In our RESTify approach, we provide a ResTL model to partition resource URI paths by the forward slash symbol into static or dynamic fragments. With individual fragments of resource URI paths, we can distinctly map *DynamicFragments* to *Parameters* in the generic split view. The RESTVSoaML metamodel provides a model element to specify input and output payload formats separately, however, in our ResTL metamodel, we assume the payload format is JSON and generate source code based on JSON encoding and decoding. Their approach generates WADL descriptions with the RESTVSoamLTool, whereas in our RESTify approach, we can generate the source code for a complete and ready-to-deploy RESTful web service.

Deljouyi and Ramsin [DR22] propose a model-driven methodology named MDD4REST, for developing RESTful web services. Modelling levels and model transformation rules are precisely defined in MDD4REST, and Domain-Driven Design (DDD) is applied for producing the domain model. MDD4REST is comprehensive in which it provides a multi-level modelling framework along with a process for applying it. MDD4REST provides an effective method for designing RESTful web services using modelling languages and supports automatic code generation through transformation of models. In addition, MDD4REST has the capability to support modern web architectures and patterns, such as Microservice, Event-Driven, and CQRS. Transformation rules are implemented to generate the models to ensure that the transitions between modelling levels are smooth and trouble-free.

The paper by Deljouyi and Ramsin [DR22] does not illustrate the MDD4REST metamodels, thus we cannot compare it with our proposed ResTL metamodel. However, MDD4REST aims to use Model-Driven Development (MDD) and Domain-Driven Design (DDD) to provide automatic code generation for repetitive and trivial tasks. The generated code is not complete and need to be integrated as snippets during development. Our proposed RESTify approach incorporates MDE and CORE to extend an existing software system to a fully functional and ready-to-deploy RESTful web service.

Tavares and Vale [TV13] propose a model-driven approach for the development of semantic RESTful web services, which provides semantic descriptions for these services rather than the complete source code, by raising the development abstraction level, providing language-independent

metamodels of services and semantic resources, and incorporating model transformations to develop interoperable complex services. The proposed RESTful Semantic Web Service (WSSR) metamodel abstracts platform-specific details and annotation formats by including only the important features that describe RESTful web services. To elaborate, the WSSR metamodel specifies the REST resource URIs with single model elements, describes their possible HTTP request method endpoints, mapped variables, and HTTP response status codes. With mapping and transformation rules, instances of the WSSR metamodel can generate semantic RESTful web services with WADL and OWL files.

Our proposed REST domain-specific modelling language, ResTL, differs from the WSSR metamodel proposed by Tavares and Value [TV13] by partitioning the REST resource URIs by forward slash symbols in a tree data structure. Thus, our approach allows concrete mappings between *DynamicFragments* and *Parameters* with the generic split view as opposed to the WSSR metamodel, which maps a complete resource URI to multiple parameters. The ResTL metamodel does not specify HTTP response status codes as the WSSR metamodel does, however, the generated RESTful web service will return appropriate HTTP response codes based on the correctness of the HTTP request. Finally, the RESTify approach incorporates CORE and MDE to generate the complete source code of a functional and ready-to-deploy RESTful web service, while their approach generates the semantic descriptions of these services.

9.2 Model-Driven Code Generation Transformations of RESTful Web Services

Gonçalves [Gon18] introduces a engineering solution, based on MDE techniques, specifically DSLs, to take as input the structural and behavioural aspects of the business domain, and generate the associated RESTful web services, while being compliant with the most recent version of the OpenAPI specification. To summarize all of the components developed in the context of this work, two DSLs were developed, one focused on the resource definition and another in the OpenAPI specification; a reference implementation was built to support the code generation process, providing a code base from which the templates used in the Xtend were extracted; the generated outcome is a combination between the generated code and a base application, where the main methods are materialized and from which the generated code extends and overrides their implementations, assuring a better maintenance and promoting code reuse along the different layers.

Hernandez-Mendez et al. [HMSM18] presents a model-driven approach for the consumption of RESTful web services in Single-Page Applications (SPAs) by introducing a Query Service metamodel and providing a tool to semi-automatically generate a SPA based on their reference architecture. SPAs support relevant processes in enterprises and are not limited to show static

information to users. The architecture of SPAs has changed from a one-to-one communication between client and server to a client querying information from multiple servers using RESTful APIs in a microservice architecture. The generated web service handles the access to APIs and reduces the complexity of the SPA due to the shift of responsibility away from the client.

Ed-douibi et al. [HCIG+16] proposes EMF-REST, an approach that leverages on MDE techniques to generate RESTful Web API from EMF models, thus promoting model management in distributed environments. The generated RESTful Web API relies on well-known libraries including JAX-RS, Context and Dependency Injection (CDI), and Enterprise Java Bean (EJB) and standards with the aim of facilitating its understanding and maintainability. EMF-REST provides a direct mapping to access data models by means of web services following the REST principles. Furthermore, EMF-REST takes advantage of model and web-specific features such as validation and security respectively.

Fischer [Fis15] addresses the generation of source code for applications compliant to the constraints of REST by incorporating the formal model to fully describe a REST API by the work of Haupt et al. [HKLS14]. The main contribution of the thesis is the development of a platform-specific model to generate REST compliant code using the Dropwizard (8.1.3) framework. A reference application is developed to identify programming patterns that could serve as a template for the resulting Java classes and derive a platform-specific model covering all the aspects that fully describe a Dropwizard application. The platform-specific model and generation of code can be used independently from the existing modelling tool. The existing modelling tool is integrated by developing model transformations that use the *resource* and *deployment* models as input and transforms them into *platform-specific* models. The modelling tool is implemented with the Eclipse Modelling Framework, Eclipse Epsilon, and the Graphical Modelling Framework, the model transformations are realized with the Eclipse Transformation Language, and the generator is written with Xtend and added to the modelling tool as a plugin.

Hussein et al. [HZS20] presents a model-based approach with a framework named REST API Auto-Generation (RAAG) that can automatically build REST APIs. The RAAG framework reuses the following platforms and technologies in a unified and decouple way. The framework integrates these packages as one infrastructure with *plugins* inside the STS Eclipse IDE, including Spring Framework, Hibernate ORM, Jackson Object Mapper, Springsource Tool Suite (STS), Eclipse Data Tools Platform (DTP), Apache Maven, and JBoss Hibernate Tools. RAAG significantly increases work productivity while introducing an easier-to-use approach. Their contributions include reducing development time for backbone services, reducing time to learn or understand the built software, enabling customization without restrictions, avoiding limitations of auto code generators, and utilizing particular frameworks in a loosely coupled manner.

9.2 Model-Driven Code Generation Transformations of RESTful Web Services

Taktak et al. [TBM+20] proposes an approach based on MDE and Model-to-Text (M2T) transformations to automatically generate the source code templates and the semantically annotated descriptions of RESTful services from heterogeneous environment data sources. Their approach introduces a *source* metamodel to model data sources, a *target* metamodel to enable modelling RESTful web services, and *transformation rules* to realize semantic mappings between the source and target metamodels. To elaborate, the *source* metamodel represents the WADL descriptor of the generated RESTful web service and the *target* metamodel complies with Hypermedia-Driven APIs (Hydra) descriptors. Hypermedia-Driven APIs enables clients to dynamically navigate to the appropriate resources by traversing hypermedia links in the API response contents. Furthermore, the authors introduce a *semantic annotation module* to automatically generate and semantically enhance descriptors of the RESTful web services with Hydra annotations. This module follows two processes, including the Hydra template generation process based on M2T transformations and the Semantic Annotation Process (SAP) of the Hydra descriptor, which is based on domain concepts matching. The transformations automatically generate a Hydra template and enhance it with semantic annotations.

Ed-douibi [Ed19] extends the work of Ed-douibi et al. [HCIG+16] by presenting a model-driven approach to facilitate the design, implementation, composition, and consumption of REST APIs. His work mainly targets 1) the OpenAPI specification which has become the preferred format to define REST APIs, and 2) OData which is focused on data-centric REST APIs and has gained momentum because of the emergence of the Open Data initiatives. For OpenAI and OData, he provides a metamodel and a UML profile to give users the flexibility to choose the representation that suits them best. By targeting the Eclipse platform, users can rely on a plethora of model-based tools to perform tasks including *model-to-model transformations* (ATL, Eclipse QVT Operational), *code generation* (Acceleo, EGL), *model validation* (Eclipse OCL, Episilon Validation Language), *model weaving* (EMF Views), and *model comparison* (EMF Compare). Concretely, his work presents EMF-REST, APIDISCOVERER, APITESTER, APIGENERATOR, and APICOMPOSER. EMF-REST enables model management via REST APIs, thus unlocking modelling tasks which currently rely on heavy desktop environments. APIDISCOVERER is an example-driven approach to discover Web API specification, thus helping developers increase the exposure of their APIs without fully writing API specifications and benefit from the OpenAPI tooling infrastructure (e.g., generating documentation, SDKs). APITESTER automates specification-based REST API testing by relying on OpenAPI. APIGENERATOR is a model-driven approach to generate OData REST APIs from conceptual data models. APICOMPOSER proposes a lightweight model-driven approach to compose and orchestrate data-centric REST APIs. Collectively, these contributions constitute an ecosystem of solutions which automate different tasks related to REST APIs devel-

opment and consumption.

In our RESTify approach, we take a collection of EMF models as input, and use a series of model-to-model, model-to-code, and code-to-code transformations to generate a fully functional and ready-to-deploy RESTful web service. In particular, we implement Acceleo modules for the model-to-code transformations to generate the resulting source code and build configuration file. Our approach differs from the work illustrated above by using CORE to encapsulate the technical complexity of REST frameworks as a *concern* named RESTify and enable the *reuse* of the concern as a method of selecting the desired REST framework to use. Our RESTify methodology currently supports the Spring Boot, Eclipse Jersey, JBoss, RESTEasy, and Apache CXF frameworks. A key feature in our approach is that users can at any time change the configuration of the reused RESTify concern to generate a new implementation of the RESTful web service based on a different REST implementation framework.

10 Conclusion & Future Work

10.1 Conclusion

The standard development procedure for extending an existing software system to a RESTful web service involves first designing a REST resource tree architecture for the underlying resource endpoints. Next, an applicable REST framework is selected as a *dependency* to the software system. The existing build module is updated and required additional files are implemented to accommodate for the selected REST implementation. Now, the existing software system is decorated with REST annotations provided by the REST framework dependency.

We introduce the RESTify concern to provide an approach based on MDE and CORE for modelling and automating the task of extending an existing software system with a RESTful web service. Our approach begins by importing the existing software system as *ImplementationClass* model elements in a RAM model. Next the developer *reuses* the RESTify concern and simply selects an appropriate REST implementation technology, completely shielding them from the implementation framework's technical complexity. Now, the developer models a resource tree with the visual ResTL modelling language to define the REST endpoints. They then specify inter-model mappings between the operations in the implementation classes of the RAM model and the REST endpoints in the ResTL model in the RAM-ResTL split view simply by drawing connecting lines between the two models. This is all our approach needs to generate a functional and ready-to-deploy RESTful web service with the RESTify transformation pipeline.

In comparison with the traditional ad hoc software development approach which converts legacy software systems to RESTful web services incrementally by programming, the RESTify approach incorporates modelling techniques to correlate the business logic of the software system to specific REST endpoints progressively and a series of model transformations to automate the conversion all-or-none. The process of extending a function with REST annotations is streamlined with the specification of inter-model mappings in the RAM-ResTL split view. Additionally, the

build configuration and additional required files are automatically generated during the RESTify transformation pipeline. Inexperienced backend web developers can find the RESTify approach to be a significantly more intuitive approach than traditional software development and achieve a functional and ready to deploy RESTful web service without any software or design defects.

Our proposed RESTify approach automates the development lifecycle with model-to-model, model-to-code, and code-to-code transformations, visualizes the REST resource tree architecture to naturally ensure a correct tree structure and enforce REST constraints.

There is significant technical complexity in learning and developing with REST implementation technologies. Even with extensive review of technical documentation of the selected REST framework, it can take substantial trial and error to build a functional RESTful web service without any software defects. By encapsulating the technical details of REST frameworks within the RESTify concern, users do not need any prerequisite technical knowledge about REST frameworks.

During the traditional development lifecycle, it can be challenging to comply the extensive constraints that REST defines. We highlight the URI naming and HTTP request method behaviour conventions. It can be difficult to abide these conventions without any visualization of the resulting REST resource architecture. With the RESTify methodology, we provide the ResTL GUI editor and generic split view for visualizing the REST resource tree architecture and RAM-ResTL mappings for annotation decoration. By strictly adhering to the REST constraints in a web service system, there can be a considerable improvement in the design and reusability of the software system.

Due to the constant evolution of new system and user specifications, software systems may often require migration to different REST frameworks during its lifetime. After the development of a RESTful web service, it can be technologically demanding and troublesome to change the REST framework. REST implementation technologies define their annotations, required additional files, and build configurations differently. The procedure of migrate an existing RESTful web service to a different REST framework can be a steep learning curve and vulnerable to software defects. In the RESTify approach, one of the key features is the capability to generate different implementations of REST by updating the configuration of the reused RESTify concern. A new RESTful web service with the newly selected REST technology can now be generated by simply re-executing the RESTify transformation pipeline.

10.2 Future Work

10.2.1 Extensions to Complete Modelling Approach of RESTify

The RESTify approach currently supports extension transformations by incorporating MDE and CORE techniques to generate a functional and ready-to-deploy RESTful web service based on an

existing software system. However, a prerequisite to the RESTify transformations is a functional software system that already provides controller classes and operations for business logic related to querying and modification of the underlying data.

Instead of starting with an already implemented system, another approach would be to model the entire software system in TouchCORE with RAM. With RAM, a complete software system can be modelled with a UML Class Diagram and UML Sequence Diagrams for each operation. Additionally, we can specify the Maven coordinates with the *groupId*, *artifactId* and *version* identifiers contained within the *ArtifactSignature* model element to integrate the Maven build automation tool to the generated application. Now, the existing RESTify procedure can be reused to generate a RESTful web service from a complete modelling approach. The new RESTification procedure would remain mostly identical to the current process, by modelling a new RAM model representation of a software system, reusing the RESTify concern, designing the REST resource tree with a ResTL model and realizing inter-model mappings between the RAM and ResTL models. In particular, the *fpcdm and COREModelExtension generation transformations* in the RESTify transformation pipeline need to be refactored to interpret *Classes* as opposed to *ImplementationClasses* from the input RAM model. The generated RESTful web service from the modified RESTify transformations will include the modelled software system in the input RAM model, the newly created and annotated controller classes and operations and the Maven build configuration file, namely the `pom.xml`.

By providing both ways to use RESTify, developers will have the choice of either following an MDE approach and modelling the software under development which queries and modifies data or use a more conventional approach and implement the software first and then use RESTify as described in this thesis to generate a RESTful web service.

10.2.2 Round-Trip Engineering with RESTify

Round-Trip Engineering (RTE) is a functionality of software development tools that synchronizes various related software artifacts, including source code, models, configuration files, and even documentation. In our RESTify approach, we provide a model-to-code approach to realize models and their inter-model mappings, in order to generate a functional and ready-to-deploy RESTful web service. However, if the users augment the generated source code subsequently, the models will be inconsistent with the code from then on. As a result, the users cannot rely on RESTify any more to model and generate any new functionalities as the models don't contain the recent source code changes.

We propose that with RTE, we can perform code-to-model transformations to parse a RESTful web service in the form of RAM and ResTL models, along with inter-model mappings between them. The main benefit of RTE is that users have the flexibility to extend the generated RESTful

web service directly in the source code, or in the models during the software development lifecycle and still have the option to incorporate the RESTify methodology to generate new functionalities.

10.2.3 ResTL Weaver

Languages in CORE are required to provide a weaver in order to fully incorporate the concern reuse process. Currently, the ResTL language does not provide a weaver implementation.

With a ResTL weaver, we can incorporate the *variation interface* of CORE to modularize the complete resource tree of a concern with *features*. Consider the BookStore concern, it can have two optional features including *Comments* and *Locations* as illustrated in Figure 10.1. The complete ResTL model for the BookStore concern is highlighted in Figure 3.4. Now, we can modularize the complete ResTL model by separating the relevant resources to the ResTL models of each feature. A possible way of modularizing the ResTL models according to features is visualized in Figure 10.2. The *BookStore* feature contains only the REST endpoints relevant to the storage of books. Since that feature is at the root of the feature model, those endpoints will always be part of the REST interface of the bookstore. The *Comments* feature reuses the */{isbn} DynamicFragment* from the ResTL model of the *BookStore* feature as the root *PathFragment*, and contains the endpoints relevant to comments from readers. The *Locations* feature reuses the */bookstore StaticFragment* from the ResTL model of the *BookStore* feature as the root *PathFragment*, and contains the endpoints relevant to the bookstore locations and their respective stock of books. We envision that the ResTL models of child features must set the *root PathFragment* to a specific *PathFragment* from the ResTL model of one of the parent features. Figure 10.2 illustrates the reuse of *PathFragments* with a small yellow box from an external source, i.e. not from TouchCORE.

Now, with a modularized BookStore concern and its underlying ResTL models, users can *reuse* the BookStore concern as specified by the concern reuse process. After a valid selection of the *variation interface* of the BookStore concern, the envisioned ResTL weaver will create a woven ResTL model based on the selected features. Users can now integrate their customized ResTL model for the BookStore concern with their other modelling tasks.

10.2.4 Augmentation of Cacheable to ResTL

The ResTL metamodel [Sch20b] visualizes a resource tree and currently supports the *uniform interface* constraint of the REST software architectural style by defining the endpoint URIs, HTTP request methods and HTTP parameters with the *PathFragment*, *AccessMethod* and *Parameter* model elements respectively. The current ResTL metamodel encompasses only a partial definition of the entire REST architectural style.

We can incorporate the *cacheable* constraint from REST in the ResTL metamodel with HTTP response headers including *Cache-Control*, *ETag* etc. The *Cache-Control* HTTP response header

Figure 10.1: Feature Model of BookStore Concern

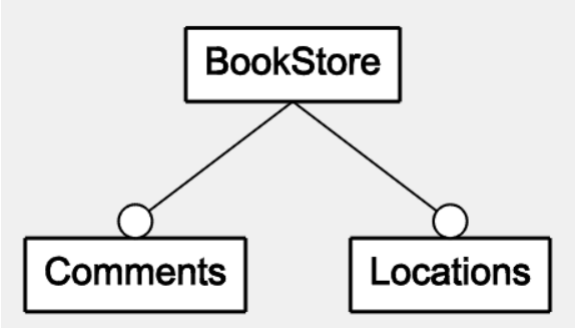
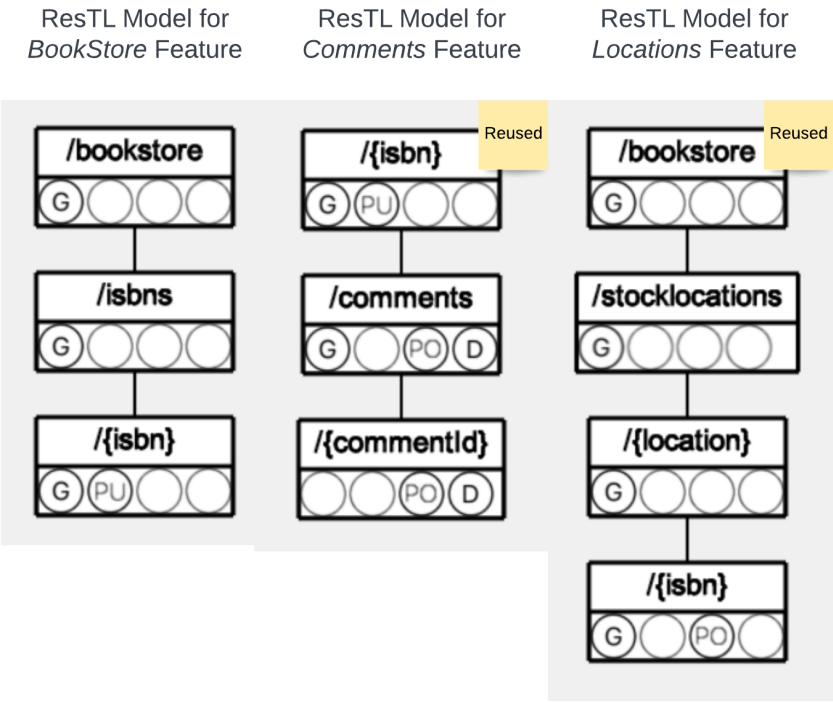


Figure 10.2: ResTL Models of Each Feature in BookStore Concern



specifies whether the response is cacheable, and if so, by whom, and for how long [Gup22a]. The *ETag* HTTP response header is an opaque string token that a server associates with a resource to uniquely identify the state of the resource over its lifetime [Gup22a]. These HTTP response headers can be added to the ResTL metamodel and contained by a REST endpoint, namely an *AccessMethod* model element. Now, the RESTify transformation pipeline can integrate these HTTP response headers with REST annotations, HTTP response builders etc. depending on the selected REST framework.

Currently, the ResTL GUI editor and controller supports operations with the *PathFragment*, *Resource* and *AccessMethod* model elements. Introducing new additions to the ResTL metamodel would also require the corresponding accommodation in the ResTL GUI editor and controller.

10.2.5 New Modelling Language to Express Layered Systems

There are additional REST constraints that are not currently expressed in the ResTL modelling language. In particular, we can introduce a new modelling language to express the *layered system* constraint from the REST architectural style. We do not express *layered systems* in the ResTL metamodel because the architecture of distributed systems is conceptually independent from the REST resource tree architecture.

Similar to the ResTL modelling language, we can express the concept of layered systems in a tree data structure by introducing *servers*. Intermediary servers can be used for the purposes of security, caching, or load balancing. Terminal servers contain the business logic for querying and modifying the database. Multiple terminal servers can enable a *microservices* architecture by only exposing a subset of the complete REST endpoints illustrated in the ResTL model.

Similar to all plug-in languages in CORE, the implementation of the layered system modelling language will need to provide a *metamodel*, *language actions*, *graphical user interface*, and a *weaver*. To extend the RESTify approach, a RAM-LayeredSystem implementation of the generic split view with mappings functionality can realize mappings between controller *classes* and terminal *servers*. Furthermore, the RESTify transformation pipeline can then generate implementations of multiple RESTful web service based on the intermediary and terminal servers that can communicate to each other as specified by the layered system architecture.

A well-designed layered system architecture can improve the scalability of the web application by reducing the network traffic to each terminal server. Furthermore, it can improve the performance of the web application by reducing server response time for clients.

10.2.6 New Modelling Language for RESTful Web Service Security

Currently with the RESTify approach, we generate a functional and ready-to-deploy RESTful web service, however, the generated application does not provide a security layer for the validation

of client requests. Security is indispensable in a web application to prevent HTTP attacks from flooding the servers with relentless requests, stealing classified information etc. However, REST defines the *stateless* constraint which specifies that the server does not retain any of the session information. Consequently, requests to the server can contain the relevant session data to be understood in isolation, without the context information from previous requests in the same session. Therefore, the RESTful Web Service Security modelling language must provide security concepts while complying to the *stateless* constraint of the REST architectural style.

As a plug-in language in CORE, the implementation of the RESTful Web Service Security modelling language will need to provide a *metamodel*, *language actions*, *graphical user interface*, and a *weaver*. The generic split view with mappings functionality can be implemented to realize mappings between a ResTL model and a RESTful Web Service Security model. These inter-model mappings can provide customized levels of security to each resource endpoint. Furthermore, the RESTify transformation pipeline can be augmented to provide the code generation of security functionality.

By incorporating security in a RESTful web service, we can reduce the network traffic by blocking HTTP flood attacks and validating the identity of incoming client requests to ensure that only authorized clients can execute certain operations.

10.2.7 New Implementations of the Generic Split View

The generic split view is an abstract module that can be specialized for displaying, editing, and realizing mappings between two arbitrary models. Currently, we provide a complete implementation for the RAM-ResTL split view with mappings to facilitate the RESTify transformation pipeline. Additionally, we demonstrate the generic nature of the generic split view by providing the Domain-Use Case split view with mappings implementation. Although there is currently no transformation pipeline for Domain-Use Case models, the Domain-Use Case split view with mappings ensures the consistency between the *Actor* and *Class* model elements.

Future work in TouchCORE can provide additional implementations of the generic split view module to realize inter-model mappings and facilitate new transformation pipelines for code generation or model transformations.

Bibliography

[AKM13] Omar Alam, Jörg Kienzle, and Gunter Mussbacher. Concern-oriented software design. In *MoDELS*, volume 8107 of *Lecture Notes in Computer Science*, pages 604–621. Springer, 2013. 1.1, 2.3

[Ali21] Hyacinth Ali. Domain-use case perspective, 2021. 6.4

[AMK22] Hyacinth Ali, Gunter Mussbacher, and Jörg Kienzle. Perspectives to promote modularity, reusability, and consistency in multi-language systems. *Innovations in Systems and Software Engineering*, January 2022. 2.3.3

[Cop22] Daniela Coppola. E-commerce worldwide, February 2022. https://www.statista.com/topics/871/online-shopping/#dossier-chapter6. 1

[Cor] Oracle Corporation. Lesson: Annotations. https://docs.oracle.com/javase/tutorial/java/annotations/. 2.5

[DR22] Amirhossein Deljouyi and Raman Ramsin. Mdd4rest: Model-driven methodology for developing restful web services. In *Proceedings of the 10th International Conference on Model-Driven Engineering and Software Development - Volume 1: MODELSWARD,*, pages 93–104. INSTICC, SciTePress, 2022. 9.1

[Ed19] Hamza Ed-douibi. *Model-driven round-trip engineering of REST APIs*. PhD thesis, Universitat Oberta de Catalunya, May 2019. 9.2

[Edu] IBM Cloud Education. What is java spring boot? https://www.ibm.com/cloud/learn/java-spring-boot. 1, 2.5.1

[Fie00] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000. 1, 2.5

[Fis15] Markus Fischer. Model-driven code generation for rest apis. Master's thesis, University of Stuttgart, 2015. 9.2

[Fou] Apache Software Foundation. Apache cxf: An open-source services framework. https://cxf.apache.org/. 1, 2.5.2.3

[Fou21] Eclipse Foundation. Eclipse jersey, July 2021. https://projects.eclipse.org/projects/ee4j.jersey. 1, 2.5.2.1

[Gon18] Rafael Corveira da Cruz Gonçalves. Restful web services development with a model-driven engineering approach. Master's thesis, Mestrado em Engenharia Informática, 2018. 9.2

[Gui] Christopher Guindon. Jakarta restful web services: The eclipse foundation. https://jakarta.ee/specifications/restful-ws/. 1, 2.5.2

[Gup22a] Lokesh Gupta. Caching rest api response, January 2022. https://restfulapi.net/caching/. 10.2.4

[Gup22b] Lokesh Gupta. What is rest, February 2022. https://restfulapi.net/. 2.5

[Hat] Red Hat. Resteasy. https://resteasy.dev/. 1, 2.5.2.2

[HCIG+16] Ed-Douibi Hamza, Javier Luis Cánovas Izquierdo, Abel Gómez, Massimo Tisi, and Jordi Cabot. Emf-rest: Generation of restful apis from models. In *Symposium on Applied Computing 2016*, Pisa, Italy, 2016. 9.2

[HKLS14] Florian Haupt, Dimka Karastoyanova, Frank Leymann, and Benjamin Schroth. A model-driven approach for rest compliant services. In *2014 IEEE International Conference on Web Services, ICWS, 2014, Anchorage, AK, USA, June 27 - July 2, 2014*, pages 129–136. IEEE Computer Society, 2014. 9.2

[HMSM18] Adrian Hernandez-Mendez, Niklas Scholz, and Florian Matthes. A model-driven approach for generating restful web services in single-page applications. In *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development*, MODELSWARD 2018, pages 480–487, Setubal, PRT, 2018. SCITEPRESS - Science and Technology Publications, Lda. 9.2

[HZS20] Salah Hussein, Samer Zein, and Norsaremah Salleh. Rest api auto generation: A model-based approach. September 2020. 9.2

[Inc22] Broadcom Inc. Http request parameter types, March 2022. https://techdocs.broadcom.com/us/en/ca-enterprise-software/it-operations-management/application-performance-management/10-7/administrating/cem-configuration/transaction-definition/about-transaction-identification/http-request-parameter-types.html. 5.1

BIBLIOGRAPHY

[KÖ6] Thomas Kühne. Matters of (meta-) modeling. *Software and Systems Modeling (SoSyM)*, 5:369–385, December 2006. 2.2

[KAAK09] Jörg Kienzle, Wisam Al Abed, and Jacques Klein. Aspect-oriented multi-view modeling. In *Proceedings of the 8th ACM International Conference on Aspect-Oriented Software Development*, AOSD '09, pages 87–98, New York, NY, USA, 2009. Association for Computing Machinery. 2.3.4

[Kin19] Pete Marvin King. Apache maven war plugin, September 2019. https://maven.apache.org/plugins/maven-war-plugin/. 7.7

[KM20] Agi Putra Kharisma and Mochamad Sukrisno Mardiyanto. Towards text-based domain-specific modeling language for representational state transfer compliant services. In *Proceedings of the 5th International Conference on Sustainable Information Engineering and Technology*, pages 74–78, New York, NY, USA, 2020. Association for Computing Machinery. 9.1

[KY21] Adil Kenzi. and Fadoua Yakine. A model driven framework for the development of adaptable rest services. In *Proceedings of the 17th International Conference on Web Information Systems and Technologies - WEBIST,*, pages 544–552. INSTICC, SciTePress, 2021. 9.1

[LRZ10] Uwe Laufs, Christopher Ruff, and Jan Zibuschka. Mt4j - a cross-platform multi-touch development framework. *CoRR*, abs/1012.0467, 2010. 5.2

[LSS09] Markku Laitkorpi, Petri Selonen, and Tarja Systa. Towards a model-driven process for designing restful web services. *2009 IEEE International Conference on Web Services*, 2009. 9.1

[Pet14] J. Petersohn. *A Multilayered Model for REST Applications*. Universitätsbibliothek der Universität Stuttgart, 2014. 9.1

[PR11] Ivan Porres and Irum Rauf. Modeling behavioral restful web service interfaces in uml. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pages 1598–1605, New York, NY, USA, 2011. Association for Computing Machinery. 9.1

[PZL] Brett Porter, Jason van Zyl, and Olivier Lamy. Welcome to apache maven. https://maven.apache.org/. 1, 2.5

[Red19] Eric Redmond. Pom reference, December 2019. https://maven.apache.org/pom.html. 8

[SAKM16] Matthias Schöttle, Omar Alam, Jörg Kienzle, and Gunter Mussbacher. On the modularization provided by concern-oriented reuse. In *Companion Proceedings of the 15th International Conference on Modularity*, pages 184–189, New York, NY, USA, 2016. Association for Computing Machinery. 2.3, 2.3.1

[Sch06] Douglas C. Schmidt. Guest editor's introduction: Model-driven engineering. 39(2):25–31, February 2006. 2.1

[Sch11] Silvia Schreier. Modeling restful applications. In *Proceedings of the Second International Workshop on RESTful Design*, WS-REST '11, pages 15–21, New York, NY, USA, 2011. Association for Computing Machinery. 9.1

[Sch13] Benjamin Schroth. Entwurf und realisierung von rest- anwendungen nach prinzipien der modellgetriebenen softwareentwicklung, 2013. 9.1

[Sch20a] Maximilian Schiedermeier. Book store internals, 2020. https://github.com/kartoffelquadrat/BookStoreInternals. 1.2, 3

[Sch20b] Maximilian Schiedermeier. Restl metamodel, 2020. 5.1, 10.2.4

[Sch21a] Maximilian Schiedermeier. fpcdm ram model with spring annotations and coremodelextension mapping generation transformation, 2021. 7.1

[Sch21b] Maximilian Schiedermeier. Ram-restl perspective, 2021. 6.3, 7

[SKK21] Maximilian Schiedermeier, Jörg Kienzle, and Bettina Kemme. FIDDLR: streamlining reuse with concern-specific modelling languages. In *SLE*, pages 164–176. ACM, 2021. 2.4

[SLL+21] Maximilian Schiedermeier, Bowen Li, Ryan Languay, Greta Freitag, Qiutan Wu, Jörg Kienzle, Hyacinth Ali, Ian Gauthier, and Gunter Mussbacher. Multi-language support in touchcore. In *MoDELS (Companion)*, pages 625–629. IEEE, 2021. 2.3.2, 2.3.3, 5.2

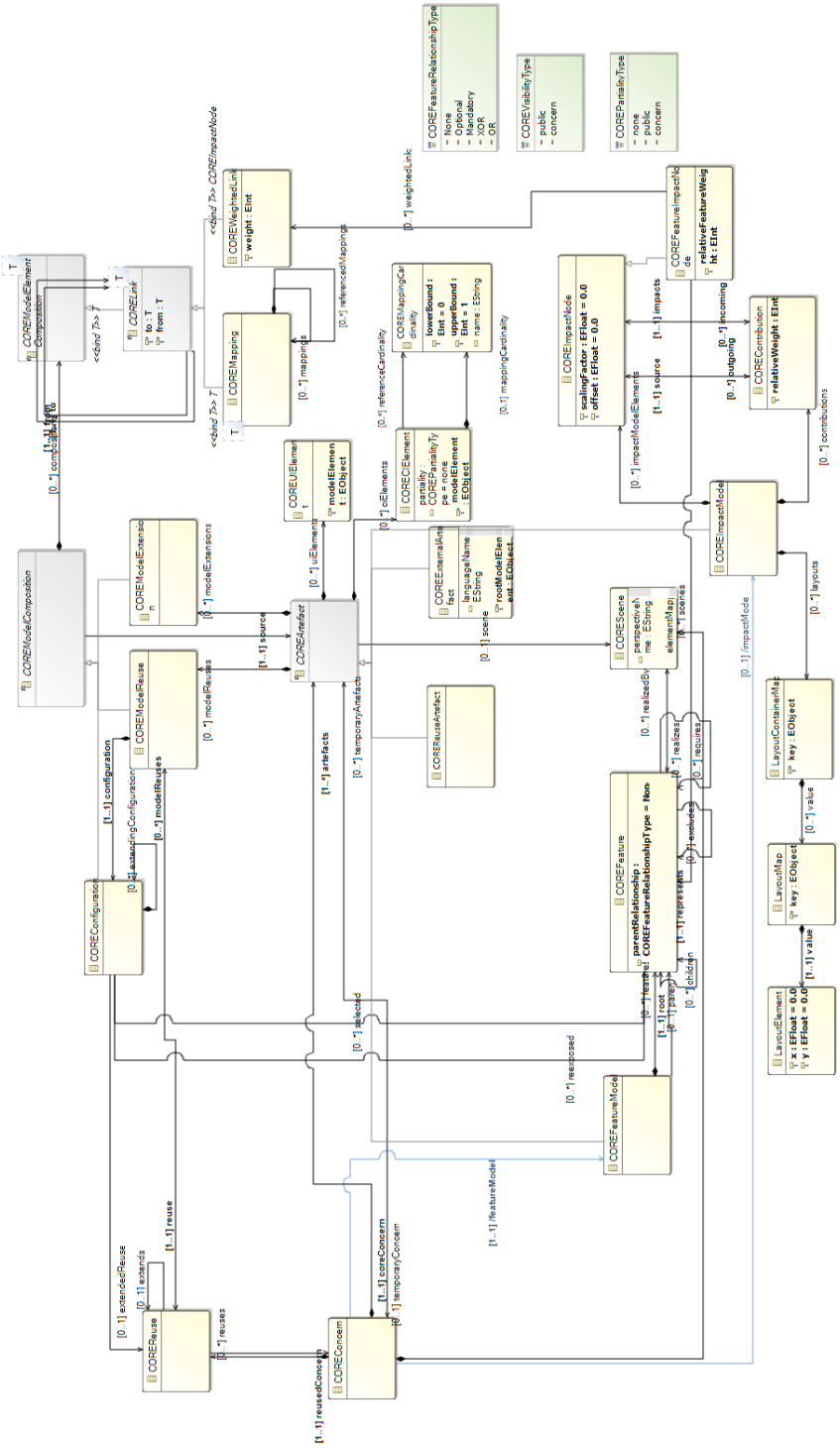
[TBM+20] Hela Taktak, Khouloud Boukadi, Michael Mrissa, Chirine Ghedira Guegan, and Faïez Gargouri. A model-driven approach for semantic data-as-a-service generation. In *WETICE*, pages 245–250. IEEE, 2020. 9.2

[TV13] Nírondes A. C. Tavares and Samyr Vale. A model driven approach for the development of semantic restful web services. In *Proceedings of International Conference on Information Integration and Web-Based Applications & Services*, IIWAS '13, pages 290–299, New York, NY, USA, 2013. Association for Computing Machinery. 9.1

A

CORE Metamodel

Figure A.1: CORE Metamodel



B

Acceleo Templates for Model-to-Code RESTify Transformations

Figure B.1: Generic pom.xml Acceleo Template

```
[comment encoding = UTF-8 /]
[module GenericPomGenerator('http://cs.mcgill.ca/sel/ram/3.0')]

[import ca::mcgill::sel::ram::generator::maven::dependencyBlocks /]

[template public generateGenericPom(aspect : Aspect)]
[comment @main/]
[file ('GenericPom.xml', false, 'UTF-8')]
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>
    <groupId>[aspect.structuralView.signature.groupId/]</groupId>
    <artifactId>[aspect.structuralView.signature.artifactId/]</artifactId>
    <version>[aspect.structuralView.signature.version/]</version>
    <packaging>jar</packaging>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
    </properties>

    <dependencies>
[for (a : ArtifactSignature | aspect.structuralView.dependencies)]
    [generateMavenDependency(a)/]
[/for]
    </dependencies>
</project>
[/file]
[/template]
```

Figure B.2: Helper Maven Depedency Acceleo Template

```
[comment encoding = UTF-8 /]
[module dependencyBlocks('http://cs.mcgill.ca/sel/ram/3.0')]

[**
 * @Author Max
 * Generates the a dependency block antrey for a given External-Artifact.
 * @param externalArtifact
 */]
[template public generateMavenDependency(a : ArtifactSignature)]
  <dependency>
    <groupId>[a.groupId/]</groupId>
    <artifactId>[a.artifactId/]</artifactId>
    <version>[a.version/]</version>
  </dependency>
[/template]
```

Figure B.3: JAX-RS Application Class Acceleo Template

```
[comment encoding = UTF-8 /]
[module ApplicationConfigGenerator('http://cs.mcgill.ca/sel/ram/3.0')]

[template public ApplicationConfigGenerator(aspect : Aspect)]
[comment @main/]
[file ('ApplicationConfig.java', false, 'UTF-8')]
package [aspect.structuralView.signature.groupId/].[aspect.structuralView.signature.artifactId/];

import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;
import java.util.HashSet;
import java.util.Set;

@ApplicationPath("/")
public class ApplicationConfig extends Application {
    private final Set<Class<?>> classes = new HashSet<>();

    public ApplicationConfig() {
[for (class : Classifier | aspect.structuralView.classes)]
    [if (class.name.toLower().contains('controller'))]
        classes.add([class.name/].class);
    [/if]
[/for]
    }

    @Override
    public Set<Class<?>> getClasses()
    {
        return classes;
    }
}
[/file]
[/template]
```

Figure B.4: Apache CXF Launcher Class Acceleo Template

```
[comment encoding = UTF-8 /]
[module ApacheCXFLauncherGenerator('http://cs.mcgill.ca/sel/ram/3.0')]

[template public ApacheCXFLauncherGenerator(aspect : Aspect)]
[comment @main/]
[file ('ApacheCXFLauncher.java', false, 'UTF-8')]
package [aspect.structuralView.signature.groupId/].[aspect.structuralView.signature.artifactId/];

import com.fasterxml.jackson.jaxrs.json.JacksonJaxbJsonProvider;
import javax.ws.rs.ext.RuntimeDelegate;
import org.apache.cxf.endpoint.Server;
import org.apache.cxf.jaxrs.JAXRSServerFactoryBean;
import java.util.ArrayList;
import java.util.List;

public class ApacheCXFLauncher {
    public static void main(String[ '[' /][ ']' /] args) {
        RuntimeDelegate delegate = RuntimeDelegate.getInstance();
        JAXRSServerFactoryBean bean = delegate.createEndpoint(new ApplicationConfig(), JAXRSServerFactoryBean.class);
        bean.setAddress("http://localhost:8080/");

        List<Object> providers = new ArrayList<>();
        providers.add(new JacksonJaxbJsonProvider());
        bean.setProviders(providers);

        Server server = bean.create();
        server.start();
    }
}
[/file]
[/template]
```

Figure B.5: Apache CXF pom.xml Acceleo Template

```
[comment encoding = UTF-8 /]
[module PomGeneratorApacheCXF('http://cs.mcgill.ca/sel/ram/3.0')]

[template public generateApacheCXFPom(aspect : Aspect)]
[comment @main/]
[file ('ApacheCXFPom.xml', false, 'UTF-8')]
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <packaging>jar</packaging>

    <properties>
        <apache.cxf.version>3.3.0</apache.cxf.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.apache.cxf</groupId>
            <artifactId>cxf-rt-frontend-jaxrs</artifactId>
            <version>${apache.cxf.version}</version>
        </dependency>

        <dependency>
            <groupId>org.apache.cxf</groupId>
            <artifactId>cxf-rt-transports-http</artifactId>
            <version>${apache.cxf.version}</version>
        </dependency>

        <dependency>
            <groupId>org.apache.cxf</groupId>
            <artifactId>cxf-rt-transports-http-jetty</artifactId>
            <version>${apache.cxf.version}</version>
        </dependency>

        <dependency>
            <groupId>com.fasterxml.jackson.jaxrs</groupId>
            <artifactId>jackson-jaxrs-json-provider</artifactId>
            <version>2.3.2</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-dependency-plugin</artifactId>
                <version>3.2.0</version>
                <executions>
                    <execution>
                        <id>copy-dependencies</id>
                        <phase>package</phase>
                        <goals>
                            <goal>copy-dependencies</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>

            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-jar-plugin</artifactId>
                <version>3.2.0</version>
                <configuration>
                    <archive>
                        <manifest>
                            <addClasspath>true</addClasspath>
                            <classpathPrefix>dependency/</classpathPrefix>
                            <mainClass>[aspect.structuralView.signature.groupId/].[aspect.structuralView.signature.artifactId/].ApacheCXFLauncher</mainClass>
                        </manifest>
                    </archive>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
[/file]
[/template]
```

Figure B.6: Eclipse Jersey pom.xml Acceleo Template

```
[comment encoding = UTF-8 /]
[module PomGeneratorEclipseJersey('http://cs.mcgill.ca/sel/ram/3.0')/]

[template public PomGeneratorEclipseJersey(aspect : Aspect)]
[comment @main/]
[file ('EclipseJerseyPom.xml', false, 'UTF-8')]
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <packaging>war</packaging>

    <properties>
        <jersey.version>2.35</jersey.version>
    </properties>

    <dependencies>
      <dependency>
            <groupId>org.glassfish.jersey.core</groupId>
            <artifactId>jersey-server</artifactId>
            <version>${jersey.version}</version>
        </dependency>

        <dependency>
            <groupId>org.glassfish.jersey.containers</groupId>
            <artifactId>jersey-container-servlet</artifactId>
            <version>${jersey.version}</version>
        </dependency>

        <dependency>
            <groupId>org.glassfish.jersey.inject</groupId>
            <artifactId>jersey-hk2</artifactId>
            <version>${jersey.version}</version>
        </dependency>

        <dependency>
            <groupId>org.glassfish.jersey.media</groupId>
            <artifactId>jersey-media-json-jackson</artifactId>
            <version>${jersey.version}</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-war-plugin</artifactId>
                <version>3.3.2</version>
                <configuration>
                    <failOnMissingWebXml>false</failOnMissingWebXml>
                </configuration>
            </plugin>

            <plugin>
                <groupId>org.eclipse.jetty</groupId>
                <artifactId>jetty-maven-plugin</artifactId>
                <version>9.4.43.v20210629</version>
            </plugin>
        </plugins>
    </build>
</project>
[/file]
[/template]
```

Figure B.7: JBoss RESTEasy pom.xml Acceleo Template

```
[comment encoding = UTF-8 /]
[module PomGeneratorJBossRESTEasy('http://cs.mcgill.ca/sel/ram/3.0')/]

[template public PomGeneratorJBossRESTEasy(aspect : Aspect)]
[comment @main/]
[file ('JBossRESTEasyPom.xml', false, 'UTF-8')]
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <packaging>war</packaging>

    <properties>
        <resteasy.version>3.15.2.Final</resteasy.version>
    </properties>

    <dependencies>
       <dependency>
            <groupId>org.jboss.resteasy</groupId>
            <artifactId>resteasy-servlet-initializer</artifactId>
            <version>${resteasy.version}</version>
        </dependency>

        <dependency>
            <groupId>org.jboss.resteasy</groupId>
            <artifactId>resteasy-jackson-provider</artifactId>
            <version>${resteasy.version}</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-war-plugin</artifactId>
                <version>3.3.2</version>
                <configuration>
                    <failOnMissingWebXml>false</failOnMissingWebXml>
                </configuration>
            </plugin>

            <plugin>
                <groupId>org.eclipse.jetty</groupId>
                <artifactId>jetty-maven-plugin</artifactId>
                <version>9.4.43.v20210629</version>
            </plugin>
        </plugins>
    </build>
</project>
[/file]
[/template]
```

Figure B.8: Spring Boot Launcher Class Acceleo Template

```
[comment encoding = UTF-8 /]
[module SpringBootLauncherGenerator('http://cs.mcgill.ca/sel/ram/3.0')/]

[template public SpringBootLauncherGenerator(aspect : Aspect)]
[comment @main /]
[file ('SpringBootLauncher.java', false, 'UTF-8')]
package [aspect.structuralView.signature.groupId/].[aspect.structuralView.signature.artifactId/];

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class SpringBootLauncher {

    public static void main(String[ '[' /][ ']' /] args) {
        SpringApplication.run(SpringBootLauncher.class, args);
    }
}
[/file]
[/template]
```

Figure B.9: Spring Boot pom.xml Acceleo Template

```
[comment encoding = UTF-8 /]
[module PomGeneratorSpringBoot('http://cs.mcgill.ca/sel/ram/3.0')]

[template public generateSpringBootPom(aspect : Aspect)]
[comment @main/]
[file ('SpringBootPom.xml', false, 'UTF-8')]
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.0.2.RELEASE</version>
    </parent>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
            <version>2.2.6.RELEASE</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <fork>true</fork>
                    <mainClass>[aspect.structuralView.signature.groupId/].[aspect.structuralView.signature.artifactId/].SpringBootLauncher</mainClass>
                </configuration>
                <executions>
                    <execution>
                        <goals>
                            <goal>repackage</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>

</project>
[/file]
[/template]
```

C

Generated RESTified Java/Maven Source Code for Supported REST Frameworks for BookStore Application

Figure C.1: GenericPom.xml

```
▼<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>
   <groupId>ca.mcgill.sel.restified</groupId>
   <artifactId>BookStore</artifactId>
   <version>1.0.0</version>
   <packaging>jar</packaging>
  ▼<properties>
     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
     <maven.compiler.source>1.8</maven.compiler.source>
     <maven.compiler.target>1.8</maven.compiler.target>
   </properties>
  ▼<dependencies>
    ▼<dependency>
       <groupId>eu.kartoffelquadrat</groupId>
       <artifactId>bookstoreinternals</artifactId>
       <version>1.2</version>
     </dependency>
   </dependencies>
 </project>
```

Figure C.2: SpringBootPom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.0.2.RELEASE</version>
  </parent>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
      <version>2.2.6.RELEASE</version>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
        <configuration>
          <fork>true</fork>
          <mainClass>ca.mcgill.sel.restified.BookStore.SpringBootLauncher</mainClass>
        </configuration>
        <executions>
          <execution>
            <goals>
              <goal>repackage</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

Figure C.3: Woven pom.xml (Spring Boot)

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.0.2.RELEASE</version>
  </parent>
  <groupId>ca.mcgill.sel.restified</groupId>
  <artifactId>BookStore</artifactId>
  <version>1.0.0</version>
  <packaging>jar</packaging>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
  </properties>
  <dependencies>
    <dependency>
      <groupId>eu.kartoffelquadrat</groupId>
      <artifactId>bookstoreinternals</artifactId>
      <version>1.2</version>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
      <version>2.2.6.RELEASE</version>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
        <configuration>
          <fork>true</fork>
          <mainClass>ca.mcgill.sel.restified.BookStore.SpringBootLauncher</mainClass>
        </configuration>
        <executions>
          <execution>
            <goals>
              <goal>repackage</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

Figure C.4: EclipseJerseyPom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <packaging>war</packaging>
  <properties>
    <jersey.version>2.35</jersey.version>
  </properties>
  <dependencies>
    <dependency>
      <groupId>org.glassfish.jersey.core</groupId>
      <artifactId>jersey-server</artifactId>
      <version>${jersey.version}</version>
    </dependency>
    <dependency>
      <groupId>org.glassfish.jersey.containers</groupId>
      <artifactId>jersey-container-servlet</artifactId>
      <version>${jersey.version}</version>
    </dependency>
    <dependency>
      <groupId>org.glassfish.jersey.inject</groupId>
      <artifactId>jersey-hk2</artifactId>
      <version>${jersey.version}</version>
    </dependency>
    <dependency>
      <groupId>org.glassfish.jersey.media</groupId>
      <artifactId>jersey-media-json-jackson</artifactId>
      <version>${jersey.version}</version>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-war-plugin</artifactId>
        <version>3.3.2</version>
        <configuration>
          <failOnMissingWebXml>false</failOnMissingWebXml>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.eclipse.jetty</groupId>
        <artifactId>jetty-maven-plugin</artifactId>
        <version>9.4.43.v20210629</version>
      </plugin>
    </plugins>
  </build>
</project>
```

Figure C.5: Woven pom.xml (Eclipse Jersey)

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>ca.mcgill.sel.restified</groupId>
  <artifactId>BookStore</artifactId>
  <version>1.0.0</version>
  <packaging>war</packaging>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
    <jersey.version>2.35</jersey.version>
  </properties>
  <dependencies>
    <dependency>
      <groupId>eu.kartoffelquadrat</groupId>
      <artifactId>bookstoreinternals</artifactId>
      <version>1.2</version>
    </dependency>
    <dependency>
      <groupId>org.glassfish.jersey.core</groupId>
      <artifactId>jersey-server</artifactId>
      <version>${jersey.version}</version>
    </dependency>
    <dependency>
      <groupId>org.glassfish.jersey.containers</groupId>
      <artifactId>jersey-container-servlet</artifactId>
      <version>${jersey.version}</version>
    </dependency>
    <dependency>
      <groupId>org.glassfish.jersey.inject</groupId>
      <artifactId>jersey-hk2</artifactId>
      <version>${jersey.version}</version>
    </dependency>
    <dependency>
      <groupId>org.glassfish.jersey.media</groupId>
      <artifactId>jersey-media-json-jackson</artifactId>
      <version>${jersey.version}</version>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-war-plugin</artifactId>
        <version>3.3.2</version>
        <configuration>
          <failOnMissingWebXml>false</failOnMissingWebXml>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.eclipse.jetty</groupId>
        <artifactId>jetty-maven-plugin</artifactId>
        <version>9.4.43.v20210629</version>
      </plugin>
    </plugins>
  </build>
</project>
```

Figure C.6: JBossRESTEasyPom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <packaging>war</packaging>
  <properties>
    <resteasy.version>3.15.2.Final</resteasy.version>
  </properties>
  <dependencies>
    <dependency>
      <groupId>org.jboss.resteasy</groupId>
      <artifactId>resteasy-servlet-initializer</artifactId>
      <version>${resteasy.version}</version>
    </dependency>
    <dependency>
      <groupId>org.jboss.resteasy</groupId>
      <artifactId>resteasy-jackson-provider</artifactId>
      <version>${resteasy.version}</version>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-war-plugin</artifactId>
        <version>3.3.2</version>
        <configuration>
          <failOnMissingWebXml>false</failOnMissingWebXml>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.eclipse.jetty</groupId>
        <artifactId>jetty-maven-plugin</artifactId>
        <version>9.4.43.v20210629</version>
      </plugin>
    </plugins>
  </build>
</project>
```

Figure C.7: Woven pom.xml (JBoss RESTEasy)

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>ca.mcgill.sel.restified</groupId>
  <artifactId>BookStore</artifactId>
  <version>1.0.0</version>
  <packaging>war</packaging>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
    <resteasy.version>3.15.2.Final</resteasy.version>
  </properties>
  <dependencies>
    <dependency>
      <groupId>eu.kartoffelquadrat</groupId>
      <artifactId>bookstoreinternals</artifactId>
      <version>1.2</version>
    </dependency>
    <dependency>
      <groupId>org.jboss.resteasy</groupId>
      <artifactId>resteasy-servlet-initializer</artifactId>
      <version>${resteasy.version}</version>
    </dependency>
    <dependency>
      <groupId>org.jboss.resteasy</groupId>
      <artifactId>resteasy-jackson-provider</artifactId>
      <version>${resteasy.version}</version>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-war-plugin</artifactId>
        <version>3.3.2</version>
        <configuration>
          <failOnMissingWebXml>false</failOnMissingWebXml>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.eclipse.jetty</groupId>
        <artifactId>jetty-maven-plugin</artifactId>
        <version>9.4.43.v20210629</version>
      </plugin>
    </plugins>
  </build>
</project>
```

Figure C.8: ApacheCXFPom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <packaging>jar</packaging>
  <properties>
    <apache.cxf.version>3.3.0</apache.cxf.version>
  </properties>
  <dependencies>
    <dependency>
      <groupId>org.apache.cxf</groupId>
      <artifactId>cxf-rt-frontend-jaxrs</artifactId>
      <version>${apache.cxf.version}</version>
    </dependency>
    <dependency>
      <groupId>org.apache.cxf</groupId>
      <artifactId>cxf-rt-transports-http</artifactId>
      <version>${apache.cxf.version}</version>
    </dependency>
    <dependency>
      <groupId>org.apache.cxf</groupId>
      <artifactId>cxf-rt-transports-http-jetty</artifactId>
      <version>${apache.cxf.version}</version>
    </dependency>
    <dependency>
      <groupId>com.fasterxml.jackson.jaxrs</groupId>
      <artifactId>jackson-jaxrs-json-provider</artifactId>
      <version>2.3.2</version>
    </dependency>
  </dependencies>
  <build>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-dependency-plugin</artifactId>
        <version>3.2.0</version>
      <executions>
        <execution>
          <id>copy-dependencies</id>
          <phase>package</phase>
         <goals>
           <goal>copy-dependencies</goal>
         </goals>
        </execution>
      </executions>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-jar-plugin</artifactId>
      <version>3.2.0</version>
      <configuration>
        <archive>
          <manifest>
            <addClasspath>true</addClasspath>
            <classpathPrefix>dependency/</classpathPrefix>
            <mainClass>ca.mcgill.sel.restified.BookStore.ApacheCXFLauncher</mainClass>
          </manifest>
        </archive>
      </configuration>
    </plugin>
  </plugins>
 </build>
</project>
```

Figure C.9: Woven pom.xml (Apache CXF)

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>ca.mcgill.sel.restified</groupId>
    <artifactId>BookStore</artifactId>
    <version>1.0.0</version>
    <packaging>jar</packaging>
  <properties>
      <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
      <maven.compiler.source>1.8</maven.compiler.source>
      <maven.compiler.target>1.8</maven.compiler.target>
      <apache.cxf.version>3.3.0</apache.cxf.version>
    </properties>
  <dependencies>
    <dependency>
        <groupId>eu.kartoffelquadrat</groupId>
        <artifactId>bookstoreinternals</artifactId>
        <version>1.2</version>
      </dependency>
    <dependency>
        <groupId>org.apache.cxf</groupId>
        <artifactId>cxf-rt-frontend-jaxrs</artifactId>
        <version>${apache.cxf.version}</version>
      </dependency>
    <dependency>
        <groupId>org.apache.cxf</groupId>
        <artifactId>cxf-rt-transports-http</artifactId>
        <version>${apache.cxf.version}</version>
      </dependency>
    <dependency>
        <groupId>com.fasterxml.jackson.jaxrs</groupId>
        <artifactId>jackson-jaxrs-json-provider</artifactId>
        <version>2.3.2</version>
      </dependency>
    </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-dependency-plugin</artifactId>
        <version>3.2.0</version>
       <executions>
        <execution>
            <id>copy-dependencies</id>
            <phase>package</phase>
          <goals>
            <goal>copy-dependencies</goal>
          </goals>
        </execution>
       </executions>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-jar-plugin</artifactId>
        <version>3.2.0</version>
       <configuration>
        <archive>
          <manifest>
            <addClasspath>true</addClasspath>
            <classpathPrefix>dependency/</classpathPrefix>
            <mainClass>ca.mcgill.sel.restified.BookStore.ApacheCXFLauncher</mainClass>
          </manifest>
        </archive>
       </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

Figure C.10: AssortmentImplController.java (Spring Annotations)

```java
package ca.mcgill.sel.restified.BookStore;

import eu.kartoffelquadrat.bookstoreinternals.AssortmentImpl;
import java.lang.Long;
import java.util.Collection;
import eu.kartoffelquadrat.bookstoreinternals.Assortment;
import eu.kartoffelquadrat.bookstoreinternals.BookDetailsImpl;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.bind.annotation.CrossOrigin;
// Start of user code for imports
// End of user code

/**
 * AssortmentImplController class definition.
 * Generated by the TouchCORE code generator.
 */
@RestController
@CrossOrigin
public class AssortmentImplController {

    @PutMapping(path="/bookstore/isbns/{isbn}")
    public void addBookToAssortment(@RequestBody BookDetailsImpl arg0) {
        Assortment instance = AssortmentImpl.getInstance();
        instance.addBookToAssortment(arg0);
    }

    @GetMapping(path="/bookstore/isbns/{isbn}")
    public BookDetailsImpl getBookDetails(@PathVariable("isbn") Long arg0) {
        Assortment instance = AssortmentImpl.getInstance();
        BookDetailsImpl bookDetails = instance.getBookDetails(arg0);
        return bookDetails;
    }

    @GetMapping(path="/bookstore/isbns")
    public Collection getEntireAssortment() {
        Assortment instance = AssortmentImpl.getInstance();
        Collection<Long> entireAssortment = instance.getEntireAssortment();
        return entireAssortment;
    }

    public static Assortment getInstance() {
        /* TODO: No message view defined */
        return null;
    }
}
```

Figure C.11: CommentsImplController.java (Spring Annotations)

```java
package ca.mcgill.sel.restified.BookStore;

import eu.kartoffelquadrat.bookstoreinternals.Comments;
import java.util.Map;
import eu.kartoffelquadrat.bookstoreinternals.CommentsImpl;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.CrossOrigin;
// Start of user code for imports
// End of user code

/**
 * CommentsImplController class definition.
 * Generated by the TouchCORE code generator.
 */
@RestController
@CrossOrigin
public class CommentsImplController {

    @PostMapping(path="/bookstore/isbns/{isbn}/comments")
    public void addComment(@PathVariable("isbn") long arg0, @RequestBody String arg1) {
        Comments instance = CommentsImpl.getInstance();
        instance.addComment(arg0, arg1);
    }

    @DeleteMapping(path="/bookstore/isbns/{isbn}/comments/{commentId}")
    public void deleteComment(@PathVariable("isbn") long arg0, @PathVariable("commentId") long arg1) {
        Comments instance = CommentsImpl.getInstance();
        instance.deleteComment(arg0, arg1);
    }

    @PostMapping(path="/bookstore/isbns/{isbn}/comments/{commentId}")
    public void editComment(@PathVariable("isbn") long arg0, @PathVariable("commentId") long arg1, @RequestBody String arg2) {
        Comments instance = CommentsImpl.getInstance();
        instance.editComment(arg0, arg1, arg2);
    }

    @GetMapping(path="/bookstore/isbns/{isbn}/comments")
    public Map getAllCommentsForBook(@PathVariable("isbn") long arg0) {
        Comments instance = CommentsImpl.getInstance();
        Map<Long, String> allCommentsForBook = instance.getAllCommentsForBook(arg0);
        return allCommentsForBook;
    }

    public static Comments getInstance() {
        /* TODO: No message view defined */
        return null;
    }

    @DeleteMapping(path="/bookstore/isbns/{isbn}/comments")
    public void removeAllCommentsForBook(@PathVariable("isbn") long arg0) {
        Comments instance = CommentsImpl.getInstance();
        instance.removeAllCommentsForBook(arg0);
    }
}
```

Figure C.12: GlobalStockImplController.java (Spring Annotations)

```java
package ca.mcgill.sel.restified.BookStore;

import eu.kartoffelquadrat.bookstoreinternals.GlobalStock;
import java.lang.Long;
import java.util.Map;
import java.util.Collection;
import eu.kartoffelquadrat.bookstoreinternals.GlobalStockImpl;
import java.lang.Integer;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.bind.annotation.CrossOrigin;
// Start of user code for imports
// End of user code

/**
 * GlobalStockImplController class definition.
 * Generated by the TouchCORE code generator.
 */
@RestController
@CrossOrigin
public class GlobalStockImplController {

    @GetMapping(path="/bookstore/stocklocations/{location}")
    public Map getEntireStoreStock(@PathVariable("location") String arg0) {
        GlobalStock instance = GlobalStockImpl.getInstance();
        Map<Long, Integer> entireStoreStock = instance.getEntireStoreStock(arg0);
        return entireStoreStock;
    }

    public static GlobalStock getInstance() {
        /* TODO: No message view defined */
        return null;
    }

    @GetMapping(path="/bookstore/stocklocations")
    public Collection getStoreLocations() {
        GlobalStock instance = GlobalStockImpl.getInstance();
        Collection<String> storeLocations = instance.getStoreLocations();
        return storeLocations;
    }

    @PostMapping(path="/bookstore/stocklocations/{location}/{isbn}")
    public void setStock(@PathVariable("location") String arg0, @PathVariable("isbn") Long arg1, @RequestBody Integer arg2) {
        GlobalStock instance = GlobalStockImpl.getInstance();
        instance.setStock(arg0, arg1, arg2);
    }

    @GetMapping(path="/bookstore/stocklocations/{location}/{isbn}")
    public int getStock(@PathVariable("location") String arg0, @PathVariable("isbn") Long arg1) {
        GlobalStock instance = GlobalStockImpl.getInstance();
        int stock = instance.getStock(arg0, arg1);
        return stock;
    }
}
```

Figure C.13: AssortmentImplController.java (JAX-RS Annotations)

```java
package ca.mcgill.sel.restified.BookStore;

import eu.kartoffelquadrat.bookstoreinternals.AssortmentImpl;
import java.lang.Long;
import java.util.Collection;
import eu.kartoffelquadrat.bookstoreinternals.Assortment;
import eu.kartoffelquadrat.bookstoreinternals.BookDetailsImpl;
import javax.ws.rs.PathParam;
import javax.ws.rs.Consumes;
import javax.ws.rs.Produces;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PUT;
// Start of user code for imports
// End of user code

/**
 * AssortmentImplController class definition.
 * Generated by the TouchCORE code generator.
 */
@Path("/")
public class AssortmentImplController {

    @PUT
    @Path("/bookstore/isbns/{isbn}")
    @Consumes("application/json")
    public void addBookToAssortment(BookDetailsImpl arg0) {
        Assortment instance = AssortmentImpl.getInstance();
        instance.addBookToAssortment(arg0);
    }

    @GET
    @Path("/bookstore/isbns/{isbn}")
    @Produces("application/json")
    public BookDetailsImpl getBookDetails(@PathParam("isbn") Long arg0) {
        Assortment instance = AssortmentImpl.getInstance();
        BookDetailsImpl bookDetails = instance.getBookDetails(arg0);
        return bookDetails;
    }

    @GET
    @Path("/bookstore/isbns")
    @Produces("application/json")
    public Collection getEntireAssortment() {
        Assortment instance = AssortmentImpl.getInstance();
        Collection<Long> entireAssortment = instance.getEntireAssortment();
        return entireAssortment;
    }

    public static Assortment getInstance() {
        /* TODO: No message view defined */
        return null;
    }
}
```

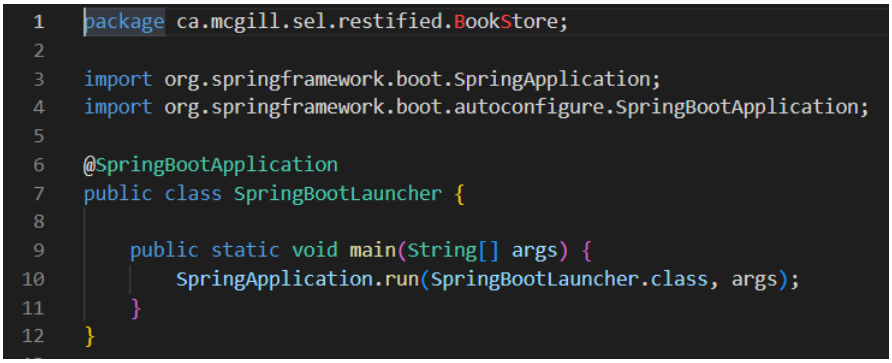
Figure C.14: CommentsImplController.java (JAX-RS Annotations)

```java
package ca.mcgill.sel.restified.BookStore;

import eu.kartoffelquadrat.bookstoreinternals.Comments;
import java.util.Map;
import eu.kartoffelquadrat.bookstoreinternals.CommentsImpl;
import javax.ws.rs.PathParam;
import javax.ws.rs.POST;
import javax.ws.rs.Produces;
import javax.ws.rs.Consumes;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.DELETE;
// Start of user code for imports
// End of user code

/**
 * CommentsImplController class definition.
 * Generated by the TouchCORE code generator.
 */
@Path("/")
public class CommentsImplController {

    @POST
    @Path("/bookstore/isbns/{isbn}/comments")
    @Consumes("application/json")
    public void addComment(@PathParam("isbn") long arg0,  String arg1) {
        Comments instance = CommentsImpl.getInstance();
        instance.addComment(arg0, arg1);
    }

    @DELETE
    @Path("/bookstore/isbns/{isbn}/comments/{commentId}")
    public void deleteComment(@PathParam("isbn") long arg0, @PathParam("commentId") long arg1) {
        Comments instance = CommentsImpl.getInstance();
        instance.deleteComment(arg0, arg1);
    }

    @POST
    @Path("/bookstore/isbns/{isbn}/comments/{commentId}")
    @Consumes("application/json")
    public void editComment(@PathParam("isbn") long arg0, @PathParam("commentId") long arg1,  String arg2) {
        Comments instance = CommentsImpl.getInstance();
        instance.editComment(arg0, arg1, arg2);
    }

    @GET
    @Path("/bookstore/isbns/{isbn}/comments")
    @Produces("application/json")
    public Map getAllCommentsForBook(@PathParam("isbn") long arg0) {
        Comments instance = CommentsImpl.getInstance();
        Map<Long, String> allCommentsForBook = instance.getAllCommentsForBook(arg0);
        return allCommentsForBook;
    }

    public static Comments getInstance() {
        /* TODO: No message view defined */
        return null;
    }

    @DELETE
    @Path("/bookstore/isbns/{isbn}/comments")
    public void removeAllCommentsForBook(@PathParam("isbn") long arg0) {
        Comments instance = CommentsImpl.getInstance();
        instance.removeAllCommentsForBook(arg0);
    }
}
```

Figure C.15: GlobalStockImplController.java (JAX-RS Annotations)

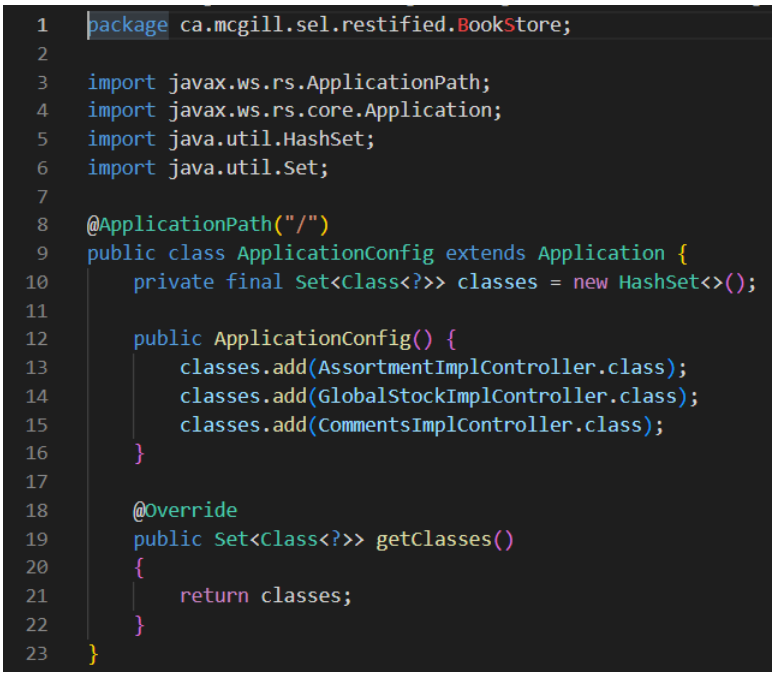
```java
package ca.mcgill.sel.restified.BookStore;

import eu.kartoffelquadrat.bookstoreinternals.GlobalStock;
import java.lang.Long;
import java.util.Map;
import java.util.Collection;
import eu.kartoffelquadrat.bookstoreinternals.GlobalStockImpl;
import java.lang.Integer;
import javax.ws.rs.PathParam;
import javax.ws.rs.POST;
import javax.ws.rs.Produces;
import javax.ws.rs.Consumes;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
// Start of user code for imports
// End of user code

/**
 * GlobalStockImplController class definition.
 * Generated by the TouchCORE code generator.
 */
@Path("/")
public class GlobalStockImplController {

    @GET
    @Path("/bookstore/stocklocations/{location}")
    @Produces("application/json")
    public Map getEntireStoreStock(@PathParam("location") String arg0) {
        GlobalStock instance = GlobalStockImpl.getInstance();
        Map<Long, Integer> entireStoreStock = instance.getEntireStoreStock(arg0);
        return entireStoreStock;
    }

    public static GlobalStock getInstance() {
        /* TODO: No message view defined */
        return null;
    }

    @GET
    @Path("/bookstore/stocklocations")
    @Produces("application/json")
    public Collection getStoreLocations() {
        GlobalStock instance = GlobalStockImpl.getInstance();
        Collection<String> storeLocations = instance.getStoreLocations();
        return storeLocations;
    }

    @POST
    @Path("/bookstore/stocklocations/{location}/{isbn}")
    @Consumes("application/json")
    public void setStock(@PathParam("location") String arg0, @PathParam("isbn") Long arg1,  Integer arg2) {
        GlobalStock instance = GlobalStockImpl.getInstance();
        instance.setStock(arg0, arg1, arg2);
    }

    @GET
    @Path("/bookstore/stocklocations/{location}/{isbn}")
    @Produces("application/json")
    public int getStock(@PathParam("location") String arg0, @PathParam("isbn") Long arg1) {
        GlobalStock instance = GlobalStockImpl.getInstance();
        int stock = instance.getStock(arg0, arg1);
        return stock;
    }
}
```

Figure C.16: SpringBootLauncher.java



```java
package ca.mcgill.sel.restified.BookStore;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class SpringBootLauncher {

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

Figure C.17: ApplicationConfig.java (JAX-RS)



```java
package ca.mcgill.sel.restified.BookStore;

import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;
import java.util.HashSet;
import java.util.Set;

@ApplicationPath("/")
public class ApplicationConfig extends Application {
    private final Set<Class<?>> classes = new HashSet<>();

    public ApplicationConfig() {
        classes.add(AssortmentImplController.class);
        classes.add(GlobalStockImplController.class);
        classes.add(CommentsImplController.class);
    }

    @Override
    public Set<Class<?>> getClasses()
    {
        return classes;
    }
}
```

Figure C.18: ApacheCXFLauncher.java

```java
package ca.mcgill.sel.restified.BookStore;

import com.fasterxml.jackson.jaxrs.json.JacksonJaxbJsonProvider;
import javax.ws.rs.ext.RuntimeDelegate;
import org.apache.cxf.endpoint.Server;
import org.apache.cxf.jaxrs.JAXRSServerFactoryBean;
import java.util.ArrayList;
import java.util.List;

public class ApacheCXFLauncher {
    public static void main(String[] args) {
        RuntimeDelegate delegate = RuntimeDelegate.getInstance();
        JAXRSServerFactoryBean bean = delegate.createEndpoint(new ApplicationConfig(), JAXRSServerFactoryBean.class);
        bean.setAddress("http://localhost:8080/");

        List<Object> providers = new ArrayList<>();
        providers.add(new JacksonJaxbJsonProvider());
        bean.setProviders(providers);

        Server server = bean.create();
        server.start();
    }
}
```