Run Then Specify: An Intelligent Framework For Building E-Business Applications

Ammar Joukhadar

Abstract: In this article, we present our approach for rapid, easy and safe web-based applications development. We implemented a new 2 Steps “Run then Specify” Model Driven framework, in which the development phase is postponed to be after (not before) the deployment phase. In this approach, the application development is based on a “descriptive and human readable” language. The language is composed of a set of predefined formal and readable business-components, which allows to manage the 3 tiers (client, business logic and storage) in one human readable and short sentence. The framework engine directly runs these business components; with no need for any further compilation or deployment. Moreover, this readable language enables the reconstruction of the business model when the analysis documents are lost or become obsolete. This approach was implemented and tested within our framework built using J2EE, Jboss and several databases including MySql database, Oracle, Postgres and Sql server. To evaluate this approach, we compared its performance to Spring Framework. The average Time-to-Learn of our Framework was 20% of that of Spring, the average Time-to-Program of our framework was 25% of that of Spring and the average Number-of Code Lines that the developers wrote using our framework was about 11% of that they wrote using Spring Framework.

Index Terms: Web-Based Development Specify and Deploy, 3 Tier language, Run then Specify, MDP, SOA, low code development, declarative languages, and intelligent languages.

1. INTRODUCTION

E-business systems are becoming more and more sophisticated because they involve a large number of functional and non-functional constraints. Actually, designing and deploying a 3-tier web-based application is a very complicated and time-consuming process. It takes time, efforts, and resources (CPU, RAM, and man/hour) (Munsi, Sehrawat & Jain, 2014). In fact, designing such applications needs advanced technologies, such as technical frameworks (Spielman, 2002) (Khandal, Meshram, & Mahatme, 2017), standard design patterns (Varhol 2002) (Deugo & Firesmith, 1998) (Khomh & Guéhéneuc, 2018), modern design patterns (3Tiers, AOP (Elrad, Filman, & Atef, 2001) (Černý, 2018), SOA, MDA (Mellor 2004) (Srai, Guerouate, Berbiche, & Lahsini, 2017), MVC, IoC (Al-Kubati et al. 2012) (Munsi, Sehrawat, & Jain, 2014), ORM, DAO, etc. (Lieberherr, Orleans, & Ovlinger, 2001) (Fowler, 2004), Incremental programming, generic programming, or code generators. Technical frameworks, such as Spring (Khandal, Meshram & Mahatme, 2017) (Munsi, Sehrawat, & Jain, 2014) (Douglas & Victoria, 2016), Struts (Varhol 2002) (Saxena 2014), Ruby on Rails (Verma 2014) (Persson 2016), can simplify the development process, but they could not free the developer from the burden of understanding the infrastructure and all the underlying patterns, especially when binding user interface with the business logic. Therefore, web-based systems development still need experts to use such frameworks. Applying Service Oriented Architecture (SOA) technology enhance incremental programming, but does not reduce maintenance time. Moreover, using Aspect Oriented Programming (AOP) (Tong 2015) (Garcia-Alonso, Berrocal, & Murillo, 2012) reduces maintenance burden, but not compilation and deployment time. On the other hand, deployment task is still time consuming, and compiling is risky, as the environment changes continuously, such as, source code compiler version, OS version, application server version, etc. Thus, actually, a maintenance task that takes about 10 minutes to be achieved may require 1 to 2 days, in order to recompile and to redeploy the corresponding new version. The objective of our approach is to annul the time needed for design and deployment to let developers focus on business analysis, as the faster we go-live, the less likely we are to go over the budget (Cindy Jutras, 2019). In order to mitigate the deployment burden, we developed the new “Run then Specify” approach which updated our previous “Specify and Deploy” approach (Joukhadar 2008). Our goal is to deploy an application only once, then add new services, in one step, by injecting formal business-components. These business components can be automatically executed by our engine (a kind of interpreter).

Our framework is also a low code, where we consider the code is not only the sentences written by the programmer, but also any action he needs to make or any text he needs to write (even if he is filling predefined templates) in order to create or modify his application. In fact, the more actions/text there are, the harder it is to maintain and modify afterwards.

* Ammar Joukhadar: associate Professor, Faculty of Information Technology Engineering, Damascus University, ajoukhadar@el-ixir.com, +963944509880.
Some Low code frameworks focuses on reducing code size needed for building specific parts of the application, like UI and the integration with the database (Waszkowski, 2019) (Mew & Field, 2018) (Golovin, 2017). Our framework focused on making these parts with zero code, especially the User Interface and Storage layer. User Interface is automatically just in time generated and bounded to its controller without even any settings. Moreover our framework builds the Storage Layer with its relations to the controllers, security, auditing and other aspect. The reason for focusing on zero code (and zero action) in parts like the UI is that the huge number of UIs makes any low code somehow huge and hard to maintain and modify. Moreover, the UI is generic, so it is generated just in time and not before the deployment, which significantly reduces code size and memory usage. This framework does not only provide zero non-functional code, it also provides built-in zero code ERP services, like Finance, Inventory, Balance, Budget, Archiving, Sales, Purchases, Fixed Assets and many more. These zero-code services allows for declarative integration between the different functional and non-functional services with no need to understand or modify the built-in services. Our framework passed through different stages. In its early stage, visual and interactive tools (such as UML/BPMN and class diagram editors) were used to implement RAD development paradigm. But, this was not useful for professionals, and made the incremental development harder and longer. That is why our framework has shifted from visual input of BPMN/class diagrams to direct textual input supervised by the system. However, we provided a read-only conversion from text to interactive BPMN/Class diagrams to improve readability, and to give multiple developers an integrated view of the whole system. That’s why Elixir’s motto is “If you can use it, you can program it” This paper is organized as follows: In the next section, we present our business architecture; who can do what, when and where. Then, we introduce our 3 tiers and our model driven declarative language. The main features of our approach will be described in the following section and finally we will show our experimental results and end by a conclusion.

2 OUR APPROACH
In this section, we present our proposed approach, by firstly presenting the business model, followed by the description of our analysis and development languages, the technical architecture, and the framework features.

2.1 Business Architecture
Our goal is develop an easy-to-use framework, which allows to develop services in only one step, i.e. a framework which is fast to use and easy to learn. This is why we developed a uniform architecture which does not make any difference between technical and business services (for instance, Tracing, Auditing, Alert, and Security, are technical services, whereas Accounting, Inventory, Money Transfer, and Billing are business services). We have an optimal learning curve because; 1) only one model is used to define business services as well as technical services and 2) only one language (Elixir TDL) is used to specify the details of these services at the 3 tiers levels (client, business logic and storage). Fig. 1 shows the business architecture of our solution.

Our framework has one GUI and one runnable for all types of users; Business Analyst, Developer, Administrator and Final User. The number of users is unlimited and they can work all together, so one can develop and maintain the application while it is running.

Business Analyst starts his work by developing a narrative analysis or CIM (Computational Independent Model) for the required service based on predefined templates, and then the CIM is validated by the end user. After the CIM has been validated by the end user, Developers can start to re-write it using Elixir language (a kind of Platform Independent language PIM) which can be interpreted by Elixir engine. The Administrator role is to manage resources, set technical-services’ configurations, such as alert, audit, tracing, security, etc. There are several types of end users according to their permissions; create, update, cancel, adjust settings, etc.

2.2 Elixir Language (From CIM to PIM)
This section presents the syntax of Elixir language. Business analysts and developers use similar syntaxes to create new business/technical services or to update existing ones. While business analysts use a kind of semi-natural languages, developers use a formal language. Both languages are declarative as explained in the next sections.

2.2.1 Elixir Analysis Language (CIM: Computational Independent Model)
Our analysis language is a description of business services using semi-natural language (pattern-based natural language). This description includes, object model and process model. An Object Model describes the fields, relations and constraints of an object type. Fig. 2 presents some examples of an object model CIM. Each line represents a field in the object model.

Each field has some properties like ID, Type, Mandatory, Unique and Editable. Other properties like how it’s calculated can be added in the Description. A Process Model describes who does what, when and on which data (business security)? This descriptive language has some predefined patterns (a pattern for each step) which enable an easy transformation to PIM. Step types include main entry, manual task, automatic task, background job, actions and view actions. We have introduced View actions and defined them using our Task Definition Language (TDL). TDL controls seamlessly all 3 tiers without any burden on the developer. Table 1 presents some examples of the process model steps.

2.2.2 Elixir Development Language (PIM)

The developer language is a formal (structured) language that has the same outlines of Elixir CIM. The advantage of using a structured (template based) analysis is the ability of being, automatically, interpreted and executed by Elixir Engine. PIM is composed of three models: Data Object, Process and Declarative Integration as described by the following paragraphs.

- Data Object Specification (DOS)

DOS describes business objects such as Person, Company, Student, Invoice, etc. Thus, it is a translation of the object model part of the CIM into a machine readable format. Properties such as the calculation formula, can be found in the description part of the table. Any property that does not exist in the CIM is a technical property, and is left for the developer (like Data source ID). DOS has three components: (1) Class description, (2) Field descriptions, and (3) Relation descriptions. Unlike general-purpose languages, Elixir DOS has extra Meta-Data that allows Elixir Engine to automatically generate database schemas and HTML views. Table 2 shows the properties included in the Meta Data.

- Process Specification

A Process is a sequence of steps (Manual Task, Automatic Task, Time Out, Interruption, Message, etc.) that lead to complete a service such as Sales, Procurement, Money Transfer, Foreign Currency Exchange, etc. We have developed our own scripting language, named Elixir which allows the developer to link process-events with DOS. The general form of Elixir TDL used to define a manual task is: Apply <actions> on <target> show <keys> where <condition>. This statement enables the end user to apply any of <actions> on any of the objects specified by <target> and that satisfy the <condition>. While executing this TDL, the user can see the <keys>. The TDL language describes the three tiers at the same time. For example, the statement: Apply [save, validate, cancel] on Sales show [id, counterpart, transactions] where status=pending. The general form of Elixir that defines an automatic task is: Apply <expression> on <target> when <target> is (created, updated, or deleted) while satisfying <condition>. In another words, the system will execute <expression> when the <target> is created, updated, or deleted while the target is satisfying the <condition>. For example: Apply generate payment on Sale when Sale is updated while status==valid. In this approach, the TDL is the machine readable representation of the (what, when and how) parts of process model CIM.

- Declarative Integration

Supporting incremental programming or SOA programming is not an easy task since services have to interact with each other. This is why we need a third component which allows us to integrate services together. Integration should not require any modification of any integrated services. Declarative integration, is a kind of a listener acting according to "Observer/Observed" design pattern rather than APIs. The following is a Business Integration example: Accounting impact: the accounting impact listens to any observed objects.
like a sale object and once it is validated, a new accounting transaction should be created. So, we can declare a new accounting impact to control the impact on accounting that each operation on an observed object has. In fact, the declarative integration is not limited to business services; it applies to technical services as well. Report generator settings and business security settings are also instances of declarative integration. Report generator is a setting that allows us to generate cross-cutting reports from non-related objects coming from different services (i.e., reports which we could not generate before the integration phase). Business Security is a setting that has two dimensions:
- Operation security: The security correspondent to "which role can execute which operation and in which conditions". For example, a Manager can cancel a sale if the sale is not delivered yet. This is the "who" part of the process model CIM.
- Information Security: The security correspondent to "which role can manage which data". For example, a Manager can, only, see sales belonging to its own agency. This is the business security part of the process model CIM.

2.2.3 Technical Architecture
The infrastructure of our framework is based on 3 Tier enterprise architecture. Fig. 3 presents the technical architecture of our system.

As the architecture shows, the services are located in the middle tier and specifically in the controller layer. As we mentioned before, services are created as structured and readable templates and registered by simply uploading these templates to the system. During the registration process, the system might need to create or update the corresponding table in the database (Joukhadar 2008). However, we should notice that our framework does not generate any code; as Elixir engine is just an interpreter that runs these templates as the same runtime process, the system might need to create or update the corresponding table in the database (Joukhadar 2008). However, we should notice that our framework does not generate any code; as Elixir engine is just an interpreter that runs these templates as they are and this is why our framework accepts instant code modification at runtime. This is has been achieved using the following mechanism:
- When the user displays the main menu, Elixir scans all processes and looks for all manual start-up events (Main Entry), in order to add them to the main menu.
- When someone clicks on task entry (Manual Task), Elixir scans all processes and looks for manual tasks that satisfy their own conditions.
- When someone manages an object, Elixir executes all automatic tasks and business rules registered for that object type.
- When a user clicks on a main-menu entry or on a manual task, Elixir generates an HTML interface based on the related TDL and Meta data.

To handle the performance issues, an automatic caching process (a kind of Just In Time Compiler) has been developed, to avoid reconstructing menus, or creating HTML pages several times. When a new workflow step is registered, Elixir clears its cache automatically in order to take the new update into account.

2.2.4 Elixir features
Elixir “Run then Specify” is a new generation of frameworks, which addresses the 3 tiers together. From a software engineering point of view, the framework has many key features:
1. Cooperative: Developers can work in parallel on the same runtime process, and different users can debug on the same runtime without blocking other developers.
2. Declarative: Elixir has no control structure, which makes code easy to develop, read and modify. There is no free text to write, as Elixir has its own predefined structured templates to fill.
3. 0 time deployment: Thanks to our “Run then Specify” approach, it is fast to run and test services, as we need 0 time to deploy the modified application.
4. Coherent and easy to use: There is only one entry point for the 3 tiers. We do not need different configurations for database, business tier or user interface. Any modification are instantly reflected on all tiers.
5. Easy to understand: There is no need to understand the infrastructure, therefore no need to any special experience in JavaScript, HTML, Enterprise Components, or Database.
6. Easy to manage: Our framework programs are quantifiable. A program is a set of templates; each template is a set of fields. It is very easy to measure the progress on any project, unlike the traditional programming languages where the code doesn’t say much about the progress and the technical issues are unquantifiable and may delay the project for a long time.
7. Easy to handover: Unlike traditional programs, Elixir programs can be handed over quickly between two engineers, which makes it suitable to develop long-term enterprise applications. This is because there is a direct mapping between user interface and source code. The source code is embedded within the runtime itself, so there is no need for any programmer.

<table>
<thead>
<tr>
<th>TABLE 3</th>
<th>EXPERIMENTAL RESULTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Criteria</td>
<td>Elixir</td>
</tr>
<tr>
<td>Learning Time</td>
<td>13 h</td>
</tr>
<tr>
<td>App Development Time</td>
<td>6 h</td>
</tr>
<tr>
<td>Code Size</td>
<td>120 lines</td>
</tr>
</tbody>
</table>
From a technical point of view, the framework features are:

1. Web based
2. Supports modern design patterns, such as MVC, AOP, MDA, Façade, Lazy List, 3Tiers, ORM, DAO.
3. Fault tolerant: It supports clustering and load balancing

3 EXPERIMENTS AND VALIDATION

The "Run then Specify" version of our Elixir framework has been used to build several integrated systems, such as Enterprise Resource Planning (including HR, Inventory, procurement and sales), Expenses Management, Student Information System, Hotels Management, Parking Rental application, Flights Management. These modules are stable and have been tested and deployed in multiple mediums and large companies. To evaluate the framework, we compared it with Spring Framework, which provides a programming and configuration model for modern Java-based enterprise applications - on any kind of deployment platform. For this purpose, we asked a set of 100 university students to develop the same application twice using Elixir and Spring. The application was a phone book where users can manage their contacts. The functional requirements for the required application were:

1. Contact CRUD (Create, Read, Update and Delete)
2. Merge numbers with the same contact

The non-functional requirements were:

1. Data and Business Security
2. Authorization
3. Authentication
4. Concurrent Access Management
5. Tracing

We have used different criteria to compare the two frameworks. For each framework, and for each student, we recorded (1) the “Learning time”: the time taken by the student to learn how to use the framework, (2) the “Development time”: the time taken by the student to develop the application, and (3) the “Manual Code size”: calculated by the number of words.

The average time to learn Spring was about 65 hours versus 13 hours for Elixir and the average time to develop using Spring was 24 hours versus 6 hours using Elixir. The average number of code lines written by the students in Spring was 1100 lines versus 120 for Elixir. Table 3 presents the results of the experiments. The values in the table are the average values for all the students.

Thus, the average time to learn our framework was 20% of that of Spring, the average time to program with our framework was 25% of that with Spring and the average number of code lines that the developers wrote using our framework was about 11% of that they wrote using Spring framework

4 CONCLUSION

In this paper, we presented the “Run then Specify” approach implemented using our Elixir framework that allows the enterprise applications development in two steps: Run then Specify. In this approach, we run the program once, then we specify as much services as we need. This is why “Run then specify” is effectively a two-step approach. Specification is written in a purely interpreted scripting language called “Elixir TDL”. Developing applications in two steps allows us to maintain only one document, which is the analysis, and to avoid issues related to software aging. Moreover, this approach allows us to avoid the risky compiling phase, which is time and effort consuming. The approach was evaluated by comparing the performance of our framework with that of Spring Framework. Our framework outperformed Spring using Time-to-Learn, Time-to-Program, and Number-of-Code-Lines criteria.

REFERENCES

[16] Srai, A., Guerouate, F., Berbiche, N., & Lahsini, H. D.


