2010

Scala Server Faces

Vikas Vittal Rao
San Jose State University

Scala Server Faces

A project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

Of the Requirements for the Degree

Master of Science, Computer Science

By

Vikas Vittal Rao

May 2010

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

Dr. Cay Horstmann

Dr. Chris Pollett

Dr. Araya Agustin

APPROVED FOR THE UNIVERSITY

Abstract:

Progress in the Java language has been slow over the last few years. Scala is emerging as one of the probable successors for Java with features such as type inference, higher order functions, closure support and sequence comprehensions. This allows object-oriented yet concise code to be written using Scala. While Java based MVC frameworks are still prevalent, Scala based frameworks along with Ruby on Rails, Django and PHP are emerging as competitors. Scala has a web framework called Lift which has made an attempt to borrow the advantages of other frameworks while keeping code concise.

Since Sun's MVC framework, Java Server Faces 2.0 and its future versions seem to be heading in a reasonably progressive direction; I have developed a framework which attempts to overcome its limitations. I call such a framework "Scala Server Faces". This framework provides a way of writing Java EE applications in Scala yet borrow from the concept of "convention over configuration" followed by rival web frameworks. Again, an Eclipse tool is provided to make the programmer's task of writing code on the popular Eclipse platform. Scala Server Faces, the framework and the tool allows the programmer to write enterprise web applications in Scala by providing features such as templating support, CRUD screen generation for database model objects, an Ant script to help deployment and integration with the Glassfish Application Server.

Table of Contents

Index of Figures

Index of Listings

1. Introduction

The Model-View-Controller (MVC) design pattern is considered to be the ideal architecture for developing interactive Enterprise web applications [11]. When such applications are written using Java based MVC based frameworks like Java Server Faces, Apache Struts or Spring; they seem to add a lot of overhead because they require the programmer to pay attention to unnecessary configuration protocols. Ruby on Rails demonstrated [2] that this was unnecessary by following the "convention over configuration" approach and this allowed programmers to develop web applications in a short amount of time.

Scala's Lift web framework [11] has made an attempt to follow Ruby on Rails in following convention and simplifying the setting up of a basic web based CRUD (create-read-update-delete) application [2]. However, most enterprise scale web applications require much more than a simple CRUD application. As the database transactions get more complex, a transactional framework such as Java Persistence API (JPA) which offers features such as object/relational mapping [4], Java language metadata annotations and/or XML descriptors to define the mapping between Java objects and a database; becomes necessary.

Again, a MVC framework like Sun's Java Server Faces (JSF) 2.0 [24] offers advantages like separation of behavior and presentation, component-level control over statefulness and events easily tied to server-side code.

However, other frameworks such as Ruby on Rails and Lift Web have attempted to provide the same advantages of JSF and JPA to an extent, by making decisions on the programmer's behalf and following the "convention over configuration" approach. Proponents of these frameworks believe that by following this approach, development happens at rapid rate compared to using Java based frameworks [18].

Since Java Server Faces 2.0 and future versions of the framework seem to be headed in the right direction [22], I have developed a framework which improves on JSF 2.0 using Scala as the programming language and following the "convention over configuration" approach. I picked Scala as the language of implementation because of its various advantages and features outlined well by its founder Martin Odersky[1] and in Section 3. I call this framework "Scala Server Faces" and it has the following features built on top of JSF 2.0:

- An Eclipse Plugin tool
- Scala Language support
- JPA support
- Command line support
- Integrate with Glassfish as the application server
- Generate CRUD screens for Entity classes
- Mixed Java and Scala projects

2. Related Work

Other web frameworks which use Scala as the base programming language or are beginning to support Scala are:

Lift is a web development framework which uses Scala as the programming language for writing web applications. Lift stresses the importance of security, maintainability, scalability and performance, while helping to improve developer productivity [12]. Lift borrows ideas from

many other web frameworks while also introducing new features of its own. Since Lift applications are written in Scala, all the Java libraries are available to be used as a part of the Lift application.

Some of the notable features of Lift are:

1. Follows the convention over configuration approach - minimal XML.
2. Excellent support for Ajax and Comet (server pushes updates to the client).
3. Templating system supports validated HTML - based on Apache wicket.
4. Runs on the JVM.

Lift attempts to separate presentation content and business logic by keeping logic out of the presentation layer. Lift provides a way to bind user-generated data into the presentation layer. This is possible because Lift's templating is built on the XML processing capabilities of the Scala Language, and allows things such as nested templates, injection of user-generated content, and data binding capabilities.

Every Lift application has a `Boot.scala` class which is responsible for the setup and configuration of the Lift framework. The `Boot` class is always located in the `bootstrap.liftweb` package and is shown here:

```
class Boot {
 def boot {
   // where to search snippet
   LiftRules.addToPackages("demo.helloworld")

   // Build SiteMap
   val entries = Menu(Loc("Home", "/", "Home")) :: Nil
   LiftRules.setSiteMap(SiteMap(entries:_*))
 }
}
```

Listing 1 – Lift's Boot.scala class

There are two basic configuration elements, placed in the boot method. The first is the `LiftRules.addToPackages` method which instructs Lift to base its searches in the `demo.helloworld` package. That would mean that snippets would be located in the `demo.helloworld.snippets` package, views would be located in the `demo.helloworld.views` package, etc. For multiple packages, `addToPackages` is called multiple times. The second item in the `Boot` class is for the creation of a navigation menu for the application.

The Lift template is shown below:

```
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:lift="http://liftweb.net/">
 <head>
   <meta http-equiv="content-type" content="text/html; charset=UTF-8" />
   <meta name="description" content="" />
   <meta name="keywords" content="" />

   <title>Welcome to Lift </title>
 </head>
 <body>
```

```
   <lift:bind name="content" />
   <lift:Menu.builder />
   <lift:msgs/>
 </body>
</html>
```

Listing 2 – Lift template

In the template above, the Scala packages containing classes are bound to the template using Lift's custom tag library. The `<lift:Menu.builder />` will iterate through the `Menu` entries from the `Boot.scala` class and create a navigation menu for the page, and the `<lift:bind name="content"/>` allows other XHTML snippets to be inserted in to the template.

Due to the above features and more, Lift has a growing community of users [25]. However, Lift lacks a testing framework and support for Internationalization. It also lacks Editor Support which reduces the number of programmers willing to use Lift as a framework for developing web applications.

Scalate is a template engine for generating text and markup based on the Scala language [28].

It supports multiple template systems:

- Scala Server Pages (SSP) which is like Java Server Pages (JSP).
- SCaml which is a Scala dialect of the Haml (HTML Abstraction Markup Language).

It also supports page templates which reduces redundancy in development. Scalate works well with other frameworks [26] and can be used in any web application or used in a standalone application to template things like emails.

Like JSP, Scalate allows calling Scala functions from inside the template as shown below:

```
object Cheese {
  def foo(productId: Int) =
    <a href={"/products/" + productId} title="Product link">Click here!</a>
}
```

Listing 3 – Scala class in a Scalate application

The above function creates a hypertext link using Scala's XML support. This can be invoked in the SSP template as shown below:

```
<%@ var user: User %>
<% import Cheese._ %>

<p>
  Hi ${user.name},${foo(123)}
</p>
```

Listing 4- Scalate's Scala Server Pages

However, Scalate suffers from the same problems as JSP and does not provide "separation of concerns", that is; it embeds calls to the business logic in the presentation layer. When these pages are handed to a designer, there is a chance bugs get introduced into the code.

Play is a Java Web development framework which has now extended its support to the Scala language [27]. It makes an attempt to follow the "convention over configuration" approach of Ruby on Rails by supporting features such as "hot deployment of code". Again, it tries to reduce "state" in its business logic and also provides a Groovy like expression syntax for the view pages. Play also provides a super class layer over the Java Persistence API taking care of the entity manager and other configurations.

Every Play application written in Scala has an `Application.scala` class which is responsible for the setup and configuration of the application. This class is located inside the `controllers` package and is shown here:

```
import play._
import play.mvc._

object Application extends Controller {
  def index = render()
}
```

Listing 5- Play framework's controller object

The Play controller can use Scala "`Traits`" to combine several interceptors as shown below:

```
import play.__
import play.mvc.__

trait Secure extends Controller {

  @Before
  def check {
    session("user") match {
    name: String => info("Logged as %s", name)
                _ => Security.login
     }
  }
}
```

After defining the trait `Secure` above, it can be used in the application controller from Listing 5.

```
object Application extends Controller with Secure {
  def index = "Hello world"
}
```

Listing 6- Play framework can combine interceptors

However, Play framework for Scala is still in development mode and the framework developers currently discourage its use for developing a complete application [27].

3. Why Scala?

Scala is a programming language designed to express common programming patterns in a concise and type-safe way. It unifies and generalizes object-oriented and functional programming, enabling Java and other programmers to be more productive. Code sizes are typically reduced [1] when compared with equivalent Java programs.

Eye-tracking experiments [30] show that for program comprehension, average time spent per word of source code is constant. So, roughly, half the code means half the time necessary to understand it.

Due to these reasons and Scala's features, companies who would depend on Java for business critical applications are turning to Scala to boost their development productivity, applications scalability and overall reliability.

For example, at Twitter, the social networking service, decided to migrate from Ruby due to its performance problems. Robey Pointer moved Twitter's core message queue from Ruby to Scala. They chose Scala over Java because although Java overcame Ruby on Rails' performance limitations, it was too verbose. These concise 1500 lines of Scala code is now an open source project under the name project Kestrel [16].

Some of Scala's features which make it a potential successor for Java are:

1. **Scala is object-oriented** - Scala is an object-oriented language in the sense that every value is an object. Types and behavior of objects are described by classes and traits. Classes are extended by sub-classing and a flexible mixin-based composition mechanism as a replacement for multiple inheritance.
2. **Scala is also a functional language** – In Scala, every function is a value. Scala provides a lightweight syntax for defining anonymous functions, it supports higher-order functions, it allows functions to be nested, and supports currying. Scala's case classes and its built-in support for pattern matching are modeled on algebraic types used in many functional programming languages. Again, Scala's pattern matching ability naturally extends to the processing of XML data. In the same context, sequence comprehensions can be used to formulate queries. These features of Scala make it an ideal language for developing applications like web services.
3. **Scala is statically typed** - Scala is equipped with a type system that enforces statically that abstractions are used in a safe and coherent manner. Scala's type system supports the following:
    a. generic classes,
    b. variance annotations,
    c. upper and lower type bounds,
    d. inner classes and abstract types as object members,
    e. compound types,
    f. explicitly typed self references, and
    g. Polymorphic methods.
4. **Scala is extensible** - Scala provides language mechanisms that make it easy to add new constructs to the language in form of libraries:
    a. any method can be used as an infix or postfix operator, and
    b. Closures are constructed automatically based on the expected type (target typing).

    Use of the above two features facilitates the definition of new statements without having to extend the syntax and without using macro-like facilities.
5. **Scala interoperates with Java and C#** - Scala is designed to interoperate well with Java and C#. Scala has the same compilation model (separate compilation, dynamic class

loading) like Java and allows access to thousands of existing Java libraries. The Microsoft .NET Framework is also supported.

To demonstrate an example of how Scala is compact and reduces boilerplate from Java code, consider the example below:

```
class Person {
  private String firstName;
  private String lastName;
  private int age;

 public Person(String firstName, String lastName, int age) {
   this.firstName = firstName;
   this.lastName = lastName;
   this.age = age;
 }

 public void setFirstName(String firstName) { this.firstName = firstName; }
 public void String getFirstName() { return this.firstName; }
 public void setLastName(String lastName) { this.lastName = lastName; }
 public void String getLastName() { return this.lastName; }
 public void setAge(int age) { this.age = age; }
 public void int getAge() { return this.age; }
}
```

The above Java class can be written in Scala like this:

```
class Person(var firstName: String, var lastName: String, var age: Int)
```

Listing 7 – Demonstration of the brevity of Scala code

In the above Scala class, the constructor is the argument list to the class, where each parameter is declared as a variable (`var` keyword). It automatically generates the equivalent of getter and setter methods, like the attribute accessors in Ruby. The most important thing to note is the brevity of the code.

4. Technical Background

Java Platform, Enterprise Edition (Java EE) 6 is the latest industry standard for enterprise Java computing. These standards are developed through a community process [15] and built on top of the of Java Platform, Standard Edition (Java SE), Java EE adds libraries and system services that support the scalability, accessibility, security, integrity, and other requirements of enterprise applications.

The goal of the Java EE platform is to reduce the complexity of developing enterprise applications. Towards this end, the platform provides a development model, APIs and a runtime environment allowing developers to focus on functionality alone.

In a multi-tiered application, the functionality of the application is separated into isolated functional areas, called “tiers”. Typically, multi-tiered applications have a “client tier”, a “middle tier”, and a “data tier” (also called the “enterprise information systems tier”). The

client tier consists of the presentation layer that makes requests to the middle tier. The middle tier's business logic handles client requests and process data, storing it in the data tier.

I will expand on the “web tier” and the “data tier” in the next two sections.

4.1 Java EE web development

The web tier of Java EE development consists of components that handle the interaction between clients and the business tier. Its primary tasks are the following:

- Dynamically generate content in various formats for the client.
- Collect input from users of the user interface and return results from the business tier components.
- Control the order in which pages are displayed on the client.
- Perform some validation logic and hold data temporarily in Java beans.

The Java Server Faces 2.0 framework, which is developed under JSR – 314 establishes the standard for building server-side user interfaces. The main components of Java Server Faces technology are as follows:

- A "web bean" for representing user interface (UI) components and managing their state; handling events, server-side validation, and data conversion; defining page navigation; supporting internationalization and accessibility; and providing extensibility for all these features

- Two Java Server Pages (JSP) custom tag libraries for expressing UI components within a JSP page and for wiring components to server-side objects

Web beans are instantiated by using the concept of "dependency injection" and the mapping of the UI with the Java class is controlled using annotations. To define a web bean:

```
@ManagedBean (name="webBeanName")
@SessionScoped
public class SomeWebBean {
  //implementation of the web bean
}
```

Listing 8 – Java managed bean

Using one of the following annotations: `@javax.inject.Named("webBeanName)` or `@javax.annotation.ManagedBean(name="webBeanName")`, the user can refer to the managed bean from the UI. For example:

```
<p>Current Tasks:</p>
<h:dataTable value="#{webBeanName.tasks}" var="row" border="1">
  <h:column>
    <h:outputText value="#{row}"/>
  </h:column>
</h:dataTable>
```

Listing 9 – Accessing the Managed Bean from the XTHML page

These beans can be stored within the request, session, or application scopes using the `@SessionScoped`, `@RequestScoped` or `@ApplicationScoped` annotations. Based on these annotations, the instances of the bean are made visible to other beans.

Some of the other key features provided by the Java Server Faces 2.0 specification include:

1. **Bean Validation**: Allows data from the user interface to be validated before it's used in the business logic of the application and when the data is stored in a database. This makes validation simple and avoids redundant validations. Consider a simple class which uses this feature:

```
public class Address {
  @NotNull @Size(max=30)
  private String addressline1;

  @Size(max=30)
  private String addressline2;
   ..........
 }
```

Listing 10 – Validator Bean

In the above example, the `@NotNull` annotation specifies that the annotated element, `addressline1`, must not be null. The `@Size` annotation imposes the limit that the annotated strings, `addressline1` and `addressline2`, should not be longer than the specified maximum of 30 characters.

2. **Template based system using Facelets:** Facelets is a powerful but lightweight declaration language that the programmer can use to develop JSF pages and to build component trees. Facelets are written using XHTML markup language and provide several advantages over the traditional use of JSP. Facelets also enables code reuse and hence reduce the time to develop and deploy applications. Using templates, the programmer can avoid creating similarly constructed pages multiple times.

   Here is an example of a template page:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
          xmlns:ui=http://java.sun.com/jsf/facelets
          xmlns:h="http://java.sun.com/jsf/html">

<head>
  <title>
    <ui:insert name="title">Page Title</ui:insert>
  </title>
</head>
  <body>
    <div>
```

```
      <ui:insert name="Links"/>
    </div>
</body>
</html>
```

Listing 11 – JSF 2.0 Facelets Template

The Facelets tag library contains a templating tag, `<ui:insert>`. To implement templating, create a template page that includes the `<ui:insert>` tag, then create a client page that uses the template. In the client page, use the `<ui:composition>` tag to point to the template and `<ui:define>` tags to specify content to insert into the template.

3. **Ajax support:** JSF 2.0 has built-in support for Ajax. With Ajax, web applications can retrieve data from the server asynchronously in the background without interfering with the user experience and behavior of the existing page.

4. The most important feature provided by Java Server Faces is its **component model**. It provides a rich architecture for managing component state, processing component data, validating user input and handling events.

Due to the above mentioned features and many others, Java Server Faces 2.0 is an excellent web development framework and will continue to be so over the next few years.

4.2 Java OR Mapping Frameworks

The Java Object Relational (OR) mapping frameworks usually form the "data tier" of Java EE applications and are used to manage these three programming activities:

1. Connect to a data source, like a relational database
2. Send SQL queries to the database
3. Retrieve results from the database in response to the query

The following code fragment written using the Java Database Connectivity (JDBC) API gives a simple example of these three steps:

```
Connection con =
DriverManager.getConnection("jdbc:Driver:driverName","username","password");
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM Table1");
List<ClassName> list = new ArrayList<ClassName>();
while (rs.next()) {
  //Process each result set item and add to list
}
```

Listing 12 – JDBC database code snippet

This short code fragment instantiates a `DriverManager` object to connect to a database driver and log into the database, instantiates a `Statement` object that queries the database; instantiates a `ResultSet` object that retrieves the results of the query, and executes a simple `while` loop, which retrieves and processes those results.

This can be done either using the JDBC API or more advanced OR mapping frameworks such as JPA or Hibernate. JPA has been well adopted and is recognized as one of the standard mechanisms for OR persistence [19], so I have used JPA as the OR mapping API through out my project.

In a large enterprise application, OR frameworks like JPA and Hibernate are better because:

1. JPA is not tightly tied with any underlying database, where as JDBC is tightly tied with the underlying database. "Write once, persist anywhere" using JPA can be achieved by changing the dialect in XML configuration file where as in JDBC, the code needs to be changed.
2. They help avoid low level JDBC and SQL code, instead they leverage object oriented programming and object model usage.
3. JPA and Hibernate also provide high end performance features such as caching, lazy loading of results and query optimizations.

The code snippet from listing 6 can be written in JPA can be as below:

```
@PersistenceContext EntityManager em = null;
Query query = em.createQuery("SELECT x from Table1 x");
@SuppressWarnings(value = "unchecked")
List<Question> result = (List<Question>) q.getResultList();
```

Listing 13 – JPA code snippet

In the above example, the server handles the instantiation of the `EntityManager` and uses the XML configuration file to retrieve the database related parameters such as the underlying database name, type, credentials and the database driver to be used.

Some of the key features of the Java Persistence API are:

1. The JPA framework provides "**Java Persistence Query Language**" (JPQL) to query entities and their persistence state. In the Java EE 6 edition, there have been a number of enhancements to the JPQL.
2. Another feature introduced in JPA 2.0 is the **Criteria API**, an API for dynamically constructing object-based queries. In essence, the Criteria API is the type-safe equivalent of JPQL. With it, the Java compiler can verify for correctness at compile time. This was not possible with JPQL.
3. JPA also supports "**pessimistic locking**". Locking is a technique for handling transaction concurrency in the database. A **Transaction** is a unit of work in the database. The transaction manages what happens within that unit of work, and when an error occurs the transaction can roll back any changes performed. When two or more database transactions concurrently access the same data, locking is used to ensure that only a single transaction can change the data at a time. There are two kinds of locking: **optimistic** and **pessimistic**. While JPA 1.0 only supported optimistic locking, JPA 2.0 now also supports pessimistic locking. In pessimistic locking, a transaction that reads the data imposes a lock on it. Other transactions cannot change the data until the first transaction completes and commits.

4.3 Scala and Java EE

Scala provides various ways of improving upon the existing Java EE platform. The Scala programming language can be used to write Java EE components such as web beans, stateful and stateless session beans and entity beans. The attributes of Scala as defined in Section 3 are what make it suitable replacement.

Consider a managed bean written in Java:

```
@ManagedBean(name = "quiz")
@SessionScoped
public class QuizMB {
  @EJB private QuizSB quizSB;

  private int score;
  private int currentQuestionIndex;
  private List<Question> questions;
  private Integer response;

  public int getScore() {
    return score;
  }

 public Question getCurrentQuestion() {
   if (questions == null) questions = quizSB.getQuestions();
   return questions.get(currentQuestionIndex);
 }

 public Map<String, Integer> getCurrentChoices() {
   Map<String, Integer> r = new LinkedHashMap<String, Integer>();
   for (Choice c : getCurrentQuestion().getChoices()) {
     r.put(c.getText(), c.getId());
   }
     return r;
 }

 public Integer getResponse() {
   return response;
 }

 public void setResponse(Integer response) {
   this.response = response;
 }

 public String submit() {
   if (currentQuestionIndex == 0) score = 0;
     if (getCurrentQuestion().getAnswer().getId() == response.intValue())
       score++;
       if (currentQuestionIndex < questions.size() - 1) {
         currentQuestionIndex++;
         response = null;
         return null;
       }
       else {
         currentQuestionIndex = 0;
```

```
            return "done";
          }
        }
    }
```

Listing 14 – Java Quiz Managed Bean

I have rewritten the above Java class in Scala in listing 20 while comparing JSF with Scala Server Faces. The Scala equivalent of the above class is 20 lines shorter. Scala accomplishes this by removing getter and setters, supporting Java annotations and allowing a functional approach to programming where relevant.

Again, writing and maintaining JPA queries become much simpler because of Scala's terse syntax. Consider the Java method below for example:

```
public List<A> findAll(EntityManager em, String queryName,
                                  Map<String, Object> params) {
  Query query = em.createNamedQuery(queryName);
  Iterator it = mp.entrySet().iterator();
  while (it.hasNext()) {
    Map.Entry pairs = (Map.Entry)it.next();
    query.setParameter(pairs.getKey(),pairs.getValue());
  }
  List<Question> result = (List<Question>) q.getResultList();
  return result;
}
```

Listing 15 – JPQL method to create a named query

The above Java method creates a `NamedQuery` using an `EntityManager` object, iterates through a `Map` and sets the query parameters. Rewriting the same method in Scala:

```
def findAll[A](em : EntityManager, queryName : String,
                params : Pair[String,Any]*) : java.util.List[A] = {
  val query = em.createNamedQuery(queryName)
  params foreach { param => query.setParameter(param._1, param._2) }
  val result = query.getResultList().asInstanceOf[java.util.List[A]]
  result
}
```

Listing 16 – JPQL method rewritten in Scala

The Java method when re-written in Scala becomes almost half its original size, yet easy to understand and debug. In the later sections, I will explain and provide examples of how components such as web beans, session beans and entity beans can be written in a simpler manner using Scala.

5. Implementation of Scala Server Faces

5.1 What is Scala Server Faces?

My definition of Scala Server Faces (SSF) refers to the writing of Java EE applications based on the "convention over configuration" approach with Scala as the programming language and using JSF 2.0 as the web development framework.

It also entails an Eclipse based plugin which makes this approach possible by providing JPA support, pre-configuring all the relevant XML files, an Ant build file which supports mixed Java and Scala projects and reduces development time by generating CRUD (Create-Read-Update-Delete) screens for entity classes.

Finally, Scala Server Faces also comes pre-integrated with the Glassfish application server. These features of SSF helps in improving programmer productivity while reducing bugs and application deployment time.

5.2 The components of Scala Server Faces

The components and technologies which are incorporated into Scala Server Faces are:

1. **Java Server Faces 2.0 -** JSF is a request-driven MVC web framework based on component driven UI design model. A detailed explanation has been provided in Section 4.1.
2. **Java Persistence API** - The JPA framework allows the programmer to effectively manage object relational mapping in applications. A detailed explanation of its core features has been provided in Section 4.2.
3. **The Scala Programming Language** - Scala is a programming language which fuses object-oriented and functional programming. It is aimed at the construction of components and component systems [1]. A detailed explanation of its core features has been provided in Section 3.
4. **Eclipse Plugin** - Using a code editor for developing Enterprise level applications is not only the current industry standard, but also a necessity due to intertwined references in code and other aspects of programming. Among all the editors used for Java development, Eclipse is the most popular [15]. Scala Server Faces provides a plugin for the Eclipse platform thus giving the programmer a way of writing Java EE applications in Scala, on top of the Eclipse web tools platform.
5. **Integration with the Glassfish server** - The GlassFish V3 application server implements the Java EE 6 platform specification. Features like fast startup time, redeploy-on-save, session state persistence and dynamic language support make it an ideal platform to deploy Enterprise web applications written in Scala.
6. **Ant build file** – Scala Server Faces projects are pre configured with an Ant build file which allows not only for clean deployment of web applications to the server, but also for the possibility of having mixed Java and Scala projects which is unique.

5.3 Features of Scala Server Faces

The Scala Server Faces framework currently provides the following set of features:

1. Follows the "convention over configuration" approach.

2. Ability to create a “Scala Server Faces project” which creates a project in the workspace with the proper package structure and pre-set classpath.
3. A configurable `build.xml` ant deployment file.
4. The ability to mix Java and Scala code in the same project.
5. A sample session bean with appropriate Scala annotations
6. Pre configured to run on the Glassfish server.
7. Uses XHTML as the standard templating engine instead of JSP.
8. CRUD (create-read-update-delete) screen generation for Entity Classes.
9. Preconfigured Java Persistence framework with a `persistence.xml` file.

5.4 The Scala Server Faces Eclipse Plugin

I developed the Scala Server Faces plugin for Eclipse using the Eclipse Plug-in Development Environment (PDE), which is an integrated development environment for developing extensions to the Eclipse platform. Using the PDE is like using Eclipse to develop Eclipse. This allows developers to use the existing tools to extend the platform itself.

The Scala Server Faces (SSF) plugin is built on top of the “**Eclipse Web Tools**” Plugin and the “**Scala IDE**” plugin for Eclipse.

The **Scala IDE plugin** provides important features such as syntax highlighting, inferred type and scaladoc hovers, hyper linking to definitions, code completion, error and warning markers, indentation, brace matching and debugger support [7].

The **Web Tools Plugin** provides features such as Server tools, HTML, CSS and JavaScript tools, XML and DTD tools and so on. Thus, building the SSF plugin on top of these plugins allowed me enhance the pre-existing useful feature set.

I will elaborate below on how the plugin was implemented.

1. **Create a new Scala Server Faces project wizard**- Extend the new project wizard by adding the following XML to the `plugin.xml` file.

```
<extension point="org.eclipse.ui.newWizards">
  <category
    id="customplugin.category.wizards"
    name="Scala Server Faces Wizard">
  </category>
  <wizard
    id = "customplugin.wizard.new.custom"
    name = "Dynamic Scala Server Faces Project"
    class="customplugin.CustomProjectNewWizard"
    category="customplugin.category.wizards">
    <description>Create a Scala Server Faces project</description>
  </wizard>
</extension>
```

Listing 17 – Extending the new project wizard

This extends the "New Project Wizard" as shown in the image below.

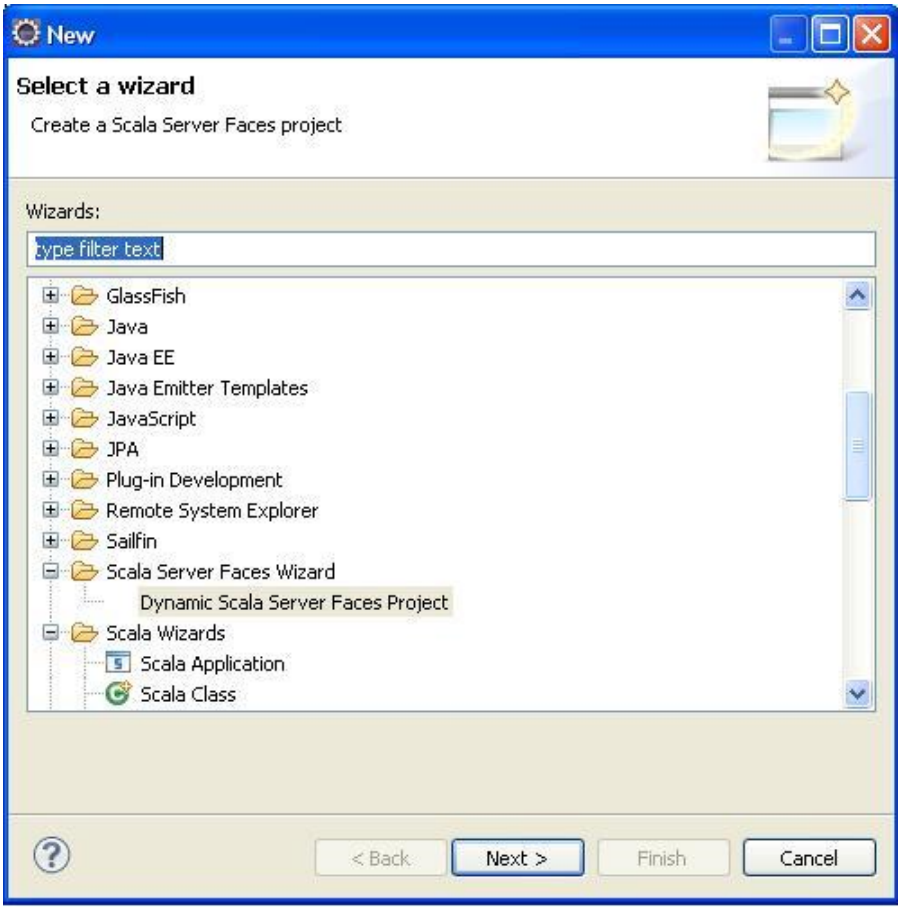

Figure 1 – New Project Wizard

2. **Create the Project structure and its components on the file system** - Define a class which extends the `Wizard` class and implements the `INewWizard` and `IExecutableExtension` interfaces. In the class below, when the user clicks the "Finish" button, the `performFinish()` method is invoked which checks the current configuration and user input parameters and starts creating the appropriate files, folders and settings. It also adds "Scala nature" to the project.

```
public class ScalaServerFacesWizard extends Wizard implements
INewWizard,IExecutableExtension {
..............
/**
 * Read project name,location from user input and create the project
 * structure on the File system.
 * Also update the user perspective to be Java EE.
 */
public boolean performFinish() {
  String name = page.getProjectName();
  URI location = null;
  if (!page.useDefaults()) {
    location = page.getLocationURI();
```

```
  }
  //create project on the file system
  createProject(name, location);
  BasicNewProjectResourceWizard.updatePerspective(config);
  return true;
}
/**
 * Based on the project name and location, create the appropriate folder
 * structure, set the context path for the sun-web.xml file.
 * Also set the project nature, classpath and preferences.
 * @param projectName
 * @param location
 * @return IProject
 */
public IProject createProject(String projectName, URI location) {
  project = createBaseProject(projectName, location);
  try {
   addNature(project);
   String[] paths = {"src", "WebContent/WEB-INF",
               "WebContent/WEB-INF/lib","src/META-INF","build",".settings" };
   addToProjectStructure(project, paths);
   String[] files = {"WebContent/index.xhtml","WebContent/WEB-INF/web.xml",
                "WebContent/WEB-INF/faces-config.xml","persistence.xml",
                 "build.properties", "build.xml","WebContent/WEB-INF/beans.xml",
                 "WebContent/WEB-INF/sun-web.xml"};
   createFiles(project, files);
   setContextPath(project, projectName); // for the sun-web.xml file.
   setProjectPreferences(project, projectName);
  } catch (Exception e) {
     e.printStackTrace();
     project = null;
    }

  return project;
}
.....................
}
```

Listing 18 – Plugin code to extend the new project wizard

3**. Display the new project in the Eclipse project explorer tab** - When the user clicks the "Finish" button, the project is created and its structure looks like in figure 2 below. This structure is created as a result of the files generated along with the project. The project structure is specified two fields: `paths` and `files`. These arrays contain the structure of the project.

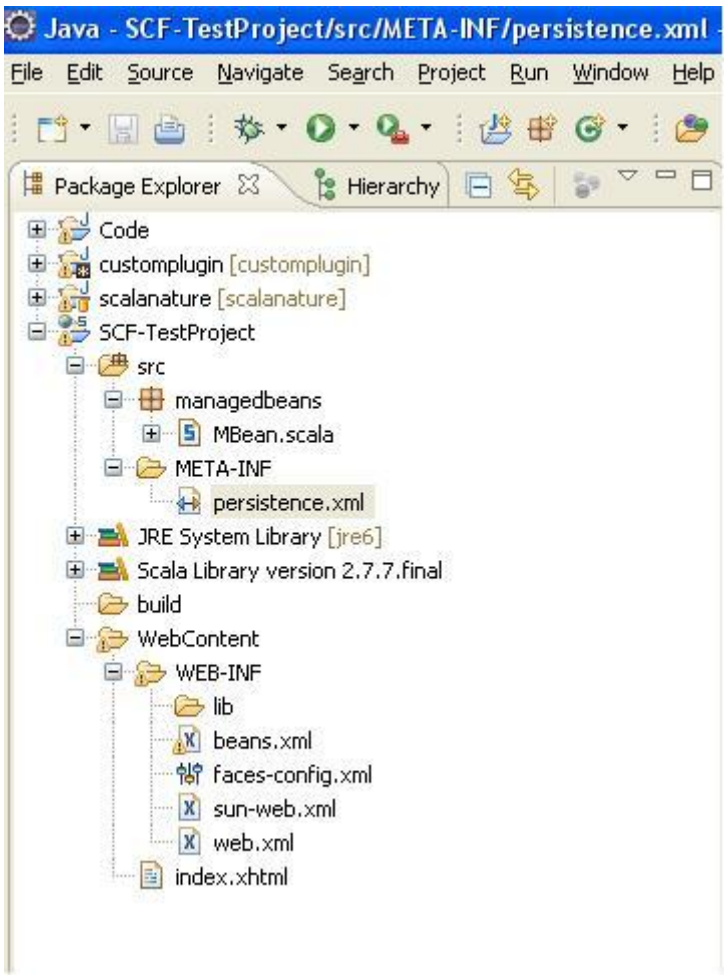

Figure 2 – Scala Server Faces Project Structure

4. **Add a menu option to generate CRUD screen for Entity classes** - To do this, extend the new project wizard by adding the following XML to the `plugin.xml` file.

```
<extension point="org.eclipse.ui.popupMenus">
  <objectContribution
    id="org.eclipse.ui.articles.action.contribution.popup.object"
    objectClass="org.eclipse.core.resources.IFile"
    adaptable="true"
    nameFilter="*.scala">
      <action
        id="customplugin.generateCrud"
        label="Create CRUD screens"
        menubarPath="additions"
        class="customplugin.ObjectAction1Delegate">
      </action>
  </objectContribution>
</extension>
```

Listing 19 – XML to provide CRUD screen generation menu

This allows the user to right click on any file with an extension ".scala" or ".java" to "right click -> create CRUD screens" as shown below.

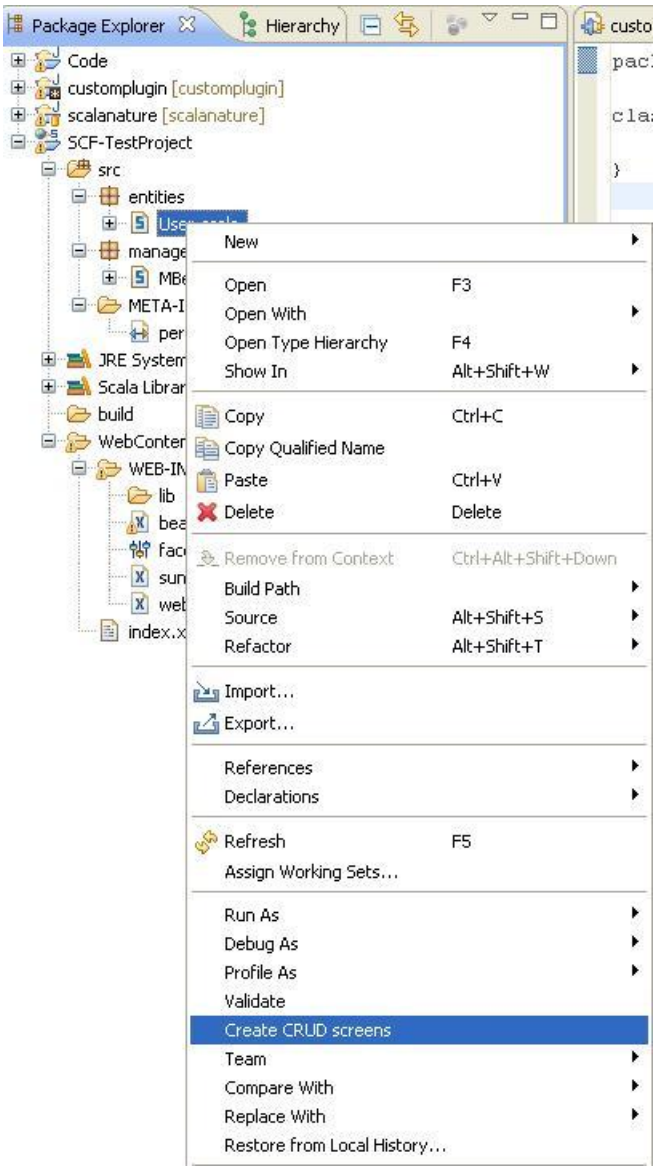

Figure 3 – Create CRUD screens option

5. **Create the Scala controller class and CRUD files inside the project**: Define a class which extends the `Wizard` class and implements the `IObjectActionDelegate` interface. In the `ObjectAction1Delegate` class below, the `setActivePart()` method initializes the `IFile` on which the CRUD screens are to be generated. When the option is clicked, if the class is

annotated with `@Entity` and there exists a `persistence.xml` file with the appropriate database parameters, I create a controller for that class along with the CRUD screen files.

```
public class ObjectAction1Delegate implements IObjectActionDelegate {
  private IFile type = null;

  public void setActivePart(IAction arg0, IWorkbenchPart arg1) {}
  public void run(IAction action) {
    if (this.type != null) {
      createCrudScreens();
    }

  private void createCrudScreens() {
    try {
      IFile persistenceFile =
                  type.getProject().getFile( "src/META-INF/persistence.xml" );
     if(persistenceFile.exists()){
       //get database credentials
     }

    ICompilationUnit cu = JavaCore.createCompilationUnitFrom(type);
    IType[] types = cu.getAllTypes();
    for(IType t : types) {
    String annotation = t.getAnnotation("Entity").getElementName();
    if(annotation != null) {
    String path = "src/managed";
    createFolder(type.getProject().getFolder(path));

    String code = CRUDUtil.getManagedBeanCode(t.getElementName(),
    t.getPackageFragment().getElementName(), "managed");
    IFile file = type.getProject().getFile(
                       "src/managed/"+t.getElementName()+"Controller"+".scala");
    createFile(file,code);
    IField[] fields = t.getFields();

    String viewPath = "WebContent/pages/"+t.getElementName();
    createFolder(type.getProject().getFolder(viewPath));
    . . .
    createFile(createListFile,listPage);
    createFile(createCreatePage,createPage);
    createFile(createEditFile,editPage);
  }

public void selectionChanged(IAction action, ISelection selection) {
  this.type = null;
  if (selection instanceof IStructuredSelection) {
    IStructuredSelection iss = (IStructuredSelection) selection;
    Object selected = iss.getFirstElement();
    if (selected instanceof IFile) {
      this.type = (IFile) selected;
    }
  }
}
```

Listing 20 –Creating SSF controller class and CRUD screens

`List.xhtml`, `Edit.xhtml` and `Create.xhtml` files are created under a folder named after the class. The generated controller class handles the business logic and functions as both the stateful session bean and the managed bean and is written in Scala. The created screen to list entities looks like below:

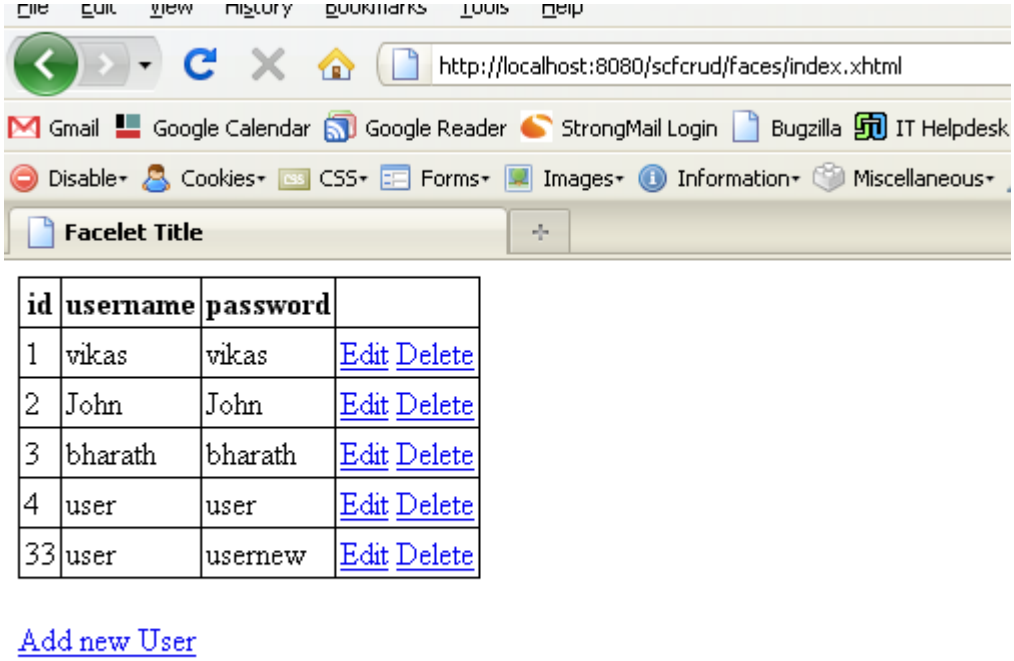

Figure 4 – Generated CRUD screen

5.5 Developing Java EE applications using Scala Server Faces

In the next three subsections, I will demonstrate how one can write Java EE applications using Scala Server Faces.

5.5.1 Writing Managed Beans using SSF

Writing managed beans in Scala provides all the advantages of Scala in the context of a Java based web application framework. This allows us to write managed beans in Scala and point to them from the `faces-config.xml` file or through annotations. Scala supports Java based annotations which allows us to annotate a class as a managed bean. We can also specify the scope under which the managed bean is available. Let us look at the sample implementation of a managed bean written in Scala:

```
import javax.faces.bean.ManagedBean
import javax.faces.bean.SessionScoped
import scala.reflect._
```

```
@ManagedBean{val name = "scalaSB"}
@SessionScoped
class ScalaMB {

  @BeanProperty var response : String = ""
  def submit() : String = {
    response = //do something
    null    //returns null
  }
}
```

The above class written in Java is shown below:

```
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;

@ManagedBean
@SessionScoped
public class ScalaMB {

  private String response = "";

  public String getResponse() {
    return response;
  }

  public void String setResponse(String response) {
    this.response = response;
  }

  public String submit() {
     response = //do something
    return null;
  }
}
```

Listing 21 –Comparing Managed Beans in Java and Scala

The above class can replace the Scala managed bean because of Scala's support for Java annotations such as `@ManagedBean` and `@SessionScoped`. Scala introduces a `@BeanProperty` annotation which generates getter and setter methods for the variables annotated with it. This allows the web server to access values to and from the user interface. Also, the Scala class requires almost half the number of lines of code as the Java managed bean.

5.5.2 Entity Beans, Session Beans and JPA using SSF

Java EE 5 introduced annotation support for applications which enables the control of injecting dependencies, resources, services, and life-cycle notifications using annotations instead of XML. Annotations associate meta-information with definitions. Scala supports Java annotations. The upcoming Scala 2.8 release will support nested annotations which increases its support for JPA.

Scala annotations follow the syntax: `@C or @C(a1, .., an)`. Here, `C` is a constructor of a class `C`, which must conform to the class `scala.Annotation`[29]. All given constructor arguments `a1, .., an` must be constant expressions (i.e., expressions on numeral literals, strings, class literals, Java enumerations and one-dimensional arrays of them).

It is this support for annotations which makes Scala very interoperable with Java EE applications. Consider the Scala managed bean from listing 21:

```
import javax.ejb.EJB
import javax.faces.bean.ManagedBean
import javax.faces.bean.SessionScoped
import scala.reflect._
import session._

@ManagedBean{val name = "test"}
@SessionScoped
class ScalaMB {
  @EJB private[this] var slsb : ScalaSB = _
  @BeanProperty var response : String = ""
  def submit() : String = {
    response = slsb.getLanguage()
    null
  }
}
```

Listing 22 –Scala session bean instantiation

Notice the addition of the line: `@EJB private[this] var slsb : ScalaSB = _`

The Scala session bean gets instantiated using "dependency injection". This is managed by the Application Server and the JSF framework. Dependency injection (DI) is a technique for supplying an external dependency to a software component. By using `private[this]`, the object's access is limited to the instantiated class without which dependency injection does not work for the Scala class. The benefit of using the dependency injection approach is the reduction of boilerplate code in the application objects since all work to initialize or setup dependencies is handled by a provider component.

The Scala implementation of the Session Bean is as below:

```
import java.util.ArrayList;
import java.util.{List => JList}
import javax.ejb.Stateless;
import javax.ejb.Remote;
import javax.persistence._
import edu.sjsu.simplequiz.entity.Choice;
import edu.sjsu.simplequiz.entity.Question;

@Stateless
class QuizSB {

  @PersistenceContext private[this] var em: EntityManager = _
  def getQuestions():JList[Question]= {
    val q:Query = em.createQuery("SELECT x FROM Question x");
```

```
    @SuppressWarnings(Array("unchecked"))
    val result = q.getResultList().asInstanceOf[JList[Question]]
    return result;
   }
}
```

Listing 23 –Scala Session Bean using Dependency Injection for EJB

Again, in the above session bean class, dependency injection handles the instantiation of the entity manager:

```
@PersistenceContext private[this] var em: EntityManager = _
```

Using this injected `EntityManager` object, JPQL queries can be written:

```
val q:Query = em.createQuery("SELECT x FROM Question x");
```

The entity bean `Question` looks like below:

```
@Entity class Question {

  @Id @GeneratedValue @BeanProperty var id:Int = 0;
  @BeanProperty var text:String = _;
  @OneToMany{val cascade = Array(CascadeType.ALL),val fetch = FetchType.EAGER}
  var choices: java.util.List[Choice] = new java.util.ArrayList[Choice]();
  @OneToOne @BeanProperty var answer:Choice = _;

  def setChoices(newvalue : java.util.ArrayList[Choice]){
      choices = newvalue;
  }

  def getChoices():java.util.List[Choice]=choices
}
```

Listing 24 –Scala Entity Bean

Finally, another major advantage of writing JPA code using Scala is in dealing with composite primary keys.

Using composite primary keys can be quite tedious in Java, because of having to implement an `@IdClass` for the key that is serializable and overrides `equals` and `hashCode`. In java this amounts to approximately a whole page of boilerplate code for every composite key. However, the requirements for an `@IdClass` is similar to the properties of a scala case class. This is because Scala auto-generates `toString`, equals and `hashCode` methods to the case class. So in Scala, all this boilerplate can be simplified as below:

```
case class CompositeKey(var some_id: Int, var other_id: Int)
```

Listing 25 – CompositeKey in Scala

5.5.3 Facelets, Web Beans, Stateful Session beans and JPA in a SSF application

In this section, I will demonstrate how a Java EE application composed of Facelets templates, named web beans, stateful session beans and JPA can be built using Scala Server Faces. This application queries the database for values of a table and renders it using a JSF data table.

Scala Server Faces provides the default template as in Listing 11. Using this, all pages which use the same template can inherit that template using the `<ui:composition>` tag to point to the template and `<ui:define>` tags to specify content to insert into the template. So, using the default `template.xhtml`, the view page can be defined as below:

```
<ui:composition template="/template.xhtml">
  <ui:define name="title">
    <h:outputText value="#{bundle.title}"/>
  </ui:define>
  <ui:define name="body">
    <h:form>
      <h:dataTable value="#{userController.items}" var="item" border="0"
        cellpadding="2" rules="all" style="border:solid 1px">
        <h:column>
          <f:facet name="header">
            <h:outputText value="id"/>
          </f:facet>
            <h:outputText value="#{item.id}"/>
        </h:column>
        <h:column>
          <f:facet name="header">
            <h:outputText value="username"/>
          </f:facet>
            <h:outputText value="#{item.username}"/>
        </h:column>
        <h:column>
          <f:facet name="header">
            <h:outputText value="password"/>
          </f:facet>
            <h:outputText value="#{item.password}"/>
        </h:column>
       </h:dataTable>
     </h:form>
  </h:body>
  </ui:define>
</ui:composition>
```

Listing 26 – Login screen using Facelets template

Another thing to note is that I use a `#{bundle}` for all our text displayed on the UI. This enables Scala Server Faces to support "Internationalization" and develop applications for any language of choice. This feature is enabled in the faces-config.xml file as shown below:

```
<resource-bundle>
    <base-name>/Bundle.properties</base-name>
    <var>bundle</var>
</resource-bundle>
```

Listing 27 – JSF resource bundle to support internationalization

For this application, I combine the stateful session bean and web bean into one as shown below:

```
import scala.reflect._
import java.util.ArrayList;
import java.util.{List => JList}
import javax.inject.Named
import javax.enterprise.context.SessionScoped
import javax.ejb.Stateful;
import javax.persistence._
import entities.User

@Named (val value = "userController")
@SessionScoped
@Stateful
@LocalBean
class UserController  {

  @BeanProperty var items: JList[User] = _
  @PersistenceContext private[this] var em: EntityManager = _

  @SuppressWarnings(Array("unchecked")) def load:JList[User]= {
    val q:Query = em.createQuery("SELECT x FROM User x");
    items = q.getResultList().asInstanceOf[JList[User]]
    return items;
  }
}
```

Listing 28– Scala managed bean and stateful session bean

Notice the brevity of code yet the functionality provided. The above code functions both as the interaction between the user interface and the web bean, while functioning as a session bean which uses an entity manager to query the underlying database. This is possible due to the new `@Named` annotation introduced in Java EE 6.

The `@Named` annotation is a new feature implemented in Java specification request (JSR) 299 which deals with context and dependency injection. In the case of a web application, JSR-299 allows the use of any Java EE component (entity beans and session beans in this example) in conjunction with JSF managed beans. Entity Java beans are by nature transactional, while the JSF managed beans are not. JSR-299 solves this transactional gap in a Java EE application by allowing the developer to replace the JSF managed bean with an entity java bean. One addition to the configuration due to the use of named web beans is an empty XML file called `beans.xml` under the WEB-INF folder.

The above application was easier to develop using Scala Server Faces because it provided:

1. A default template.
2. All the configuration files: `faces-config.xml`, `sun-web.xml`, `beans.xml` and `web.xml` preconfigured.
3. A resource bundle to support internationalization.
4. A pre-existing session bean which could be extended.

6. Comparison of Scala Server Faces with other Web Frameworks

6.1 Comparing Scala Server Faces and Java Server Faces

Although Scala Server Faces is built on Java Server Faces, the SSF framework's features make it a better solution to develop enterprise web applications with. The primary argument against using Java based MVC frameworks has always been that they are too hard to configure and get the application running for the first time [18]. In contrast, Scala Server Faces comes with all the benefits of using both a framework such as JSF, but tools to improve programmer productivity along with functional Scala as the programming language.

To compare the two frameworks, I chose the Java Quiz example application from the Core Java Server Faces [20] book.

Developing a simple JSF 2.0 based quiz application.

1. Create a "New Dynamic Web Application" in Eclipse with the name of `SQuiz`.
2. Create two XHTML files called `index.xhtml` and `done.xhtml`.
3. Add the glassfish module library to the project's "classpath".
4. Create a managed bean called `QuizMB` under the `src/managed` folder.
5. Create a session bean called `QuizSB` under the `src/session` folder.
6. Create two entities, `Question` and `Choice` under the `src/entities` folder.
7. Create a `persistence.xml` file under the `src/META-INF/` folder.
8. Create a `faces-config.xml` file.
9. Create an empty `beans.xml` file under the `WEB-INF` folder.
10. Right click on the project and choose "Run as -> Run on Server". Select the application server and you should see from the Console view that the Glassfish server starts and then the executing index page appears in the web browser.

Developing a simple SSF based quiz application

1. Create a "New Scala Server Faces project" using Eclipse with the name of `SQuiz`.
2. Create two XHTML files called `index.xhtml` and `done.xhtml`.
3. Create a managed bean called `QuizMB` under the `src/managed` folder.
4. Create a session bean called `QuizSB` under the `src/session` folder.
5. Create two entities, `Question` and `Choice` under the `src/entities` folder.
6. Right click on the project and choose "Run as -> Run on Server". Select the application server and you should see from the Console view that the Glassfish server starts and then executing index page appears in the web browser.

The SSF project comes pre-configured with JPA, JSF 2.0 and the "classpath" pre-set so steps 3, 7, 8 and 9 are not necessary.

While the XHTML pages are the same, I compare the business logic (Java and Scala code) used by the two applications below:

Java code:

```
@ManagedBean(name = "quiz")
@SessionScoped
public class QuizMB {
  @EJB
  private QuizSB quizSB;

  private int score;
  private int currentQuestionIndex;
  private List<Question> questions;
  private Integer response;

  public int getScore() {
    return score;
  }

  public Question getCurrentQuestion() {
    if (questions == null) questions = quizSB.getQuestions();
    return questions.get(currentQuestionIndex);
  }

  public Map<String, Integer> getCurrentChoices() {
    Map<String, Integer> r = new LinkedHashMap<String, Integer>();
    for (Choice c : getCurrentQuestion().getChoices()) {
      r.put(c.getText(), c.getId());
    }
    return r;
  }

  public Integer getResponse() {
    return response;
  }

  public void setResponse(Integer response) {
    this.response = response;
  }

  public String submit() {
    if (currentQuestionIndex == 0) score = 0;
    if (getCurrentQuestion().getAnswer().getId() == response.intValue())
      score++;
    if (currentQuestionIndex < questions.size() - 1) {
      currentQuestionIndex++;
      response = null;
      return null;
    }
    else {
      currentQuestionIndex = 0;
      return "done";
    }
  }
}

@Stateless
public class QuizSB {
  @PersistenceContext
```

```
    private EntityManager em;

    public List<Question> getQuestions() {
      Query q = em.createQuery("SELECT x FROM Question x");
      @SuppressWarnings(value = "unchecked")
      List<Question> result = (List<Question>) q.getResultList();
      return result;
    }
}
```

Rewriting the above Java code in Scala below:

```
@ManagedBean{val name="quiz"} @SessionScoped class QuizMB {
  @EJB private[this] var quizSB: QuizSB = _
  @BeanProperty var score:Int = 0
  var currentQuestionIndex:Int = 0
  var questions: JList[Question] = _
  @BeanProperty var response = ""

  def getCurrentQuestion() = {
    if (questions eq null)  questions = quizSB.getQuestions();
    questions.get(currentQuestionIndex);
  }

  def getCurrentChoices() : Map= {
    var r = new LinkedHashMap[String,Int]();
    var choices = Conversions.convertList(getCurrentQuestion().choices)
    choices.foreach{c => r.put(c.text, c.id)}
    r
  }

  def submit():String = {
    if (questions == null) return null;
    if (currentQuestionIndex == 0) score = 0;
    if (getCurrentQuestion().answer.id == response.asInstanceOf[Int]) score+=1;
    if (currentQuestionIndex < questions.size() - 1) {
      currentQuestionIndex+=1;
      return "next";
    }
    else {
      currentQuestionIndex = 0;
      return "done";
    }
  }
}

@Stateless class QuizSB {
  @PersistenceContext private[this] var em: EntityManager = _
  def getQuestions():JList[Question]= {
    val q:Query = em.createQuery("SELECT x FROM Question x");
    @SuppressWarnings(Array("unchecked"))
    val result = q.getResultList().asInstanceOf[JList[Question]]
    result
   }
}
```

Listing 29 – Java Quiz business logic written in Java and Scala

The above Scala code is significantly shorter than Java yet provides the same functionality. Including the Entity classes, the Scala code is shorter by about 40 %. Again, configuration files of the `faces-config.xml`, `build.xml`, `persistence.xml` files are created with the SSF project while JSF requires the user to configure all of it increasing startup time and chances of making a mistake. The Ant build file provided by SSF also helps combine Java and Scala classes into the same project, which helps the use of pre-existing Java code to be used as a part of the SSF project.

Comparing JSF and Scala Server Faces

	Parameter	SSF	JSF
1	Number of classes	4	4
2	Lines of Code	105	178
3	Steps to get started	6	10
4	Testing Framework	Yes	Yes
5	Editor support	Yes	Yes
6	CRUD screen generation	Yes	No
7	Internationalization support	Yes	Yes
8	Deployment Help	Yes	No
9	JPA Setup support	Yes	No

Listing 30 – comparing JSF and Scala Server Faces

6.2 Comparing Scala Server Faces and the Lift Web Framework

I introduced Lift in section 3 as a web framework built which uses a Model-View-Controller architecture to develop applications using the Scala programming language.

I compare the projects which Lift and Scala Server Faces auto-generate below.

To develop a sample Lift application

The following steps are required to create, configure and deploy a Lift application:

1. To create a Lift application, execute this command on the terminal:

```
mvn org.apache.maven.plugins:maven-archetype-plugin:1.0-alpha-7:create
-DarchetypeGroupId=net.liftweb
-DarchetypeArtifactId=ift-archetype-blank
-DarchetypeVersion=1.0
-DremoteRepositories=http://scala-tools.org/repo-releases
-DgroupId=PROJECTNAME -DartifactId=LiftLogin
```

Listing 31 –Maven command to generate Lift Project

   This creates "`LiftLogin`" directory.

2. The above project generated is not ready to be imported into Eclipse for development. To make it suitable for Eclipse from the project folder, run the command:

```
mvn eclipse:eclipse
```

To import this project up in Eclipse:
Import -> General -> Existing Projects into workspace -> Project
This creates a `LiftLogin` project in the Eclipse workspace. The folder structure created is as follows:
`src/main/scala` directory contains the Scala source code.
`src/main/webapp` directory contains `HTML`, `JavaScript`, `CSS`, and the `WEB-INF` folder.

3. The application generated comes with template screens for a Login application (implementation provided by Lift)

4. To build and run the generated application: `mvn install`.

5. Run the application on the server using: `mvn jetty:run`

The code generated for the user authentication looks like below with the `MetaMegaProtoUser` class providing most of the implementation out of the box.

```
/**
 * The singleton that has methods for accessing the database
 */
object User extends User with MetaMegaProtoUser[User] {
  override def dbTableName = "users"
  override def screenWrap =
                Full(<lift:surround with="default" at="content">
                       <lift:bind /></lift:surround>)
  // define the order fields will appear in forms and output
  override def fieldOrder = List(id, userName, password, textArea)

  }

/**
 * An O-R mapped "User" class that includes first name, last name, password and
we add a "Personal Essay" to it
 */
class User extends MegaProtoUser[User] {
  def getSingleton = User

  // define an additional field for a personal essay
  object textArea extends MappedTextarea(this, 2048) {
    override def textareaRows  = 10
    override def textareaCols = 50
    override def displayName = "Personal Essay"
  }

}
```

Listing 32 –Lift User authentication

Developing a sample Scala Server Faces application

1. Create a "New Scala Server Faces project" using Eclipse with the name of `SSFLogin`.
2. The application generated comes with template screens for a Login application.
3. Right click on the project and choose "Run as -> Run on Server".

Since Lift and SSF applications are both written using Scala, the business logic code for both are similar except that SSF uses Java Server Faces components and follows the Java EE 6 specification for application development. However, Lift has its disadvantages such as the lack of a testing framework, no editor support and no support for internationalization. Again, Lift's CRUD (Create-Read-Update-Delete) screen generation using the `Mapper` functionality is still basic yet complicated to use. Lift also lacks editor support.

Finally, SSF allows continuous development without needing a server restart by providing "hot deployment" of code while Lift requires a build and install for the changes to be reflected on the server.

Comparing Lift and SSF

	Parameter	SSF	Lift Web
1	Number of classes	1	2
3	Steps to get started	3	4
4	Testing Framework	Yes	No
5	Editor support	Yes	No
6	CRUD generation	Yes	Yes(basic)
7	Internationalization support	Yes	No
8	Hot Deployment	Yes	No
9	Deployment Help	Ant	Maven

Listing 33 –Comparing Lift and Scala Server Faces

6.3 Comparing Scala Server Faces with Ruby on Rails

I introduced Ruby on Rails in section three as a MVC web framework built for the Ruby programming language. Assuming the Ruby programming language is already installed; the following steps are required to create a Ruby on Rails (Rails) application:

1. Create a Rails application using the command: `rails -d mysql ProjectName>`. "`ProjectName`" will be the name of the sub-directory the app is created in.
2. In the created application's directory, execute the command: `ruby script/server`. The welcome screen is accessible at: `http://localhost:3000`.
3. Create the `UserController` class (shown below).
4. Create the `Login` view page.
5. Point the browser to: `http://myserver.dev:3000/user/login` to see the login screen.

The Ruby code for the `UserController` class looks like below:

```
class UserController < ApplicationController
def authenticate
  @user = User.new(params[:userform])
  valid_user = User.find(:first,:conditions =>
              ["user_name = ? and password = ?",@user.user_name, @user.password])
       if valid_user
         session[:user_id]=valid_user.user_name
         redirect_to :action => 'private'
       else
         flash[:notice] = "Invalid User/Password"
         redirect_to :action=> 'login'
       end
end
def login
  end

  def private
  if !session[:user_id]
    redirect_to :action=> 'login'
    end
  end
end
```

Listing 34 –Ruby on Rails User Authentication

I have already explained the steps to create a SSF application in the prior two sections. To write the above Ruby controller in Scala:

```
@Named {val value="loginBean"} @SessionScoped @Stateful @LocalBean
class LoginBean  {
   @BeanProperty var username:String = _
   @BeanProperty var password:String = _
   @PersistenceContext private[this] var em: EntityManager = _

   @SuppressWarnings(Array("unchecked")) def authenticate:Boolean= {
     val q:Query = em.createQuery("SELECT p FROM Person p WHERE p.username=:username
                                                    AND p.firstName = :password");
     q.setParameter("username", username);
     q.setParameter("password", password");
     val user = q.getSingleResult();
     (user eq null)
   }
}
```

Listing 35 –Scala Server Faces User Authentication

Comparing SSF and Ruby on Rails

	Parameter	SSF	Ruby on Rails
1	Number of classes	2	2

2	Lines of Code	15	29
3	Steps to get started	3	6
4	Testing Framework	Yes	Yes
5	Editor support	Yes	Yes
6	CRUD generation	Yes	Yes
7	Internationalization support	Yes	Yes
8	Hot Deployment	Yes	Yes
9	Deployment Help	Yes	Yes

Listing 36 –Comparing SSF and Ruby on Rails

Although Ruby on Rails has the advantages of being concise and easy to set up, it has a few disadvantages:

1. Tests [17] have shown that Ruby on Rails is consistently slower than Lift and Java web applications.
2. Its security features are not provided out of the box and require a plugin.
3. Active Record, the OR Mapping framework followed by Rails does not scale as expected for complex applications and database queries in comparison with Java OR frameworks like Hibernate and JPA [21].

5.4 Results of the comparison

Summarizing the comparisons from the last three sections, we notice that Java Server Faces required the most number of steps to get started of all the web frameworks.

Lift web, although it is compact and follows the convention over configuration approach, misses some of the key features to make it a successful Scala based web framework. In terms of performance, Ruby on Rails is consistently slower [17].

Finally, the most important thing to note is that Scala Server Faces compares well and does better in some cases with all the web frameworks.

	Parameter	Java Server Faces	Scala Server Faces	Ruby on Rails	Lift
1	Number of classes	Same	Same	Same	Same
3	Steps to get started	10	4	6	5
4	Testing Framework	Yes	Yes	Yes	No
5	Editor support	Yes	Yes	Yes	No
6	CRUD generation	No	Yes	Yes	Yes
7	Internationalization	Yes	Yes	Yes	No
8	Hot Deployment	Yes	Yes	Yes	No
9	Deployment Help	Yes	Yes	Yes	Yes

Listing 37 –Summary of the Web Framework comparison

7. Conclusion

While the Java language has been progressing slowly over the last few years, frameworks such as Ruby on Rails, PHP, Django and other light weight frameworks are emerging as competitors to the Java based frameworks. Although the advantages of Java and the Java virtual machine are well acknowledged in a production environment, opponents to its use point to the slow development time, redundant code and the overuse of XML configuration files as reasons not to use the Java based frameworks. At the same time, the Scala programming language, with its ability to run on the JVM and yet having features like type inference, higher order functions, closure support and sequence comprehensions is a promising replacement to Java. Again, the state of Java EE 6 and its components are headed in the right direction from features such as reduced configuration using annotations, context and dependency injection and EJB 3.1 [22].

In this project, I have used Scala to develop Java EE applications and also provide a tool which tries to provide the same features as some of its rival frameworks such as Ruby and Lift. By following the "convention over configuration" approach and demonstrating best practices using Scala, I have shown that there is a reduction in the number of steps to get applications deployed and running, reducing errors by pre-configuring the development environment for the programmer and at the same time increasing programmer productivity. Also, by doing redundant tasks such as creating CRUD screens along with the related business logic, providing an Ant build tool to combine Java and Scala code in the same project, support development on the Eclipse platform and JSF 2.0 support, this framework is immediately useful for any Java EE programmer who wishes to develop Enterprise applications using the Scala programming and Eclipse.

References

1. M. Odersky and al, “An overview of the scala programming language”. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

2. Michael Bächle, Paul Kirchberg, "Ruby on Rails," IEEE Software, vol. 24, no. 6

3. Chris Richardson, “ORM in Dynamic Languages”, ACM Volume 6,Issue 3, ACM Press, May/June 2008

4. Elizabeth J. O'Neil, "Object/relational mapping 2008: hibernate and the entity data model (edm)", Proceedings of the 2008 ACM SIGMOD international conference on Management of data, June 09-12, 2008, Vancouver, Canada

5. Marjan Mernik, Jan Heering, Anthony M. Sloane, "When and How to Develop Domain-Specific Languages", ACM Computing Surveys (CSUR), Volume 37 Issue 4, ACM Press, December 2005.

6. Arie van Deursen, Paul Klint, Joost Visser, "Domain-Specific Languages: An Annotated Bibliography", ACM SIGPLAN Notices, June 2000.

7. Sean McDirmid, EPFL and Martin Odersky, EPFL,"The Scala Plugin for Eclipse", Proceedings of the Eclipse Technology eXchange Workshop at ECOOP 2006, Nantes, July 3-7, 2006.

8. Debasish Ghosh, Steve Vinoski, "Scala and Lift — Functional Recipes for the Web", IEEE Internet Computing archive - Volume 13, Issue 3 (May 2009)

9. Tony C Shan, Winnie W Hua, "Taxonomy of Java Web Application Frameworks", e-Business Engineering ICEBE '06. IEEE International Conference on Pub Date: Oct. 2006

10. Jeff Offutt, "Quality Attributes of Web Software Applications", IEEE Software, vol. 19, no. 2, pp. 25-32, Mar./Apr. 2002, doi:10.1109/52.991329.

11. Avraham Leff, James T. Rayfield, "Web-Application Development Using the Model/View/Controller Design Pattern", edoc, pp.0118, Fifth IEEE International Enterprise Distributed Object Computing Conference, 2001.

12. Ghosh, D.; Vinoski, S., "Scala and Lift Functional Recipes for the Web", Internet Computing, IEEE, 5 May 2009

13. Adriaan Moors, Frank Piessens, Martin Odersky , "Generics of a higher kind", Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications, Pages: 423-438 Year of Publication: 2008

14. Eclipse community survey 2009 - http://www.eclipse.org/org/press-release/Eclipse_Survey_2009_final.pdf

15. JSR 313: JavaTM Platform, Enterprise Edition 6 (Java EE 6) Specification - http://jcp.org/en/jsr/detail?id=313

16. Project Kestrel – Robey Pointer's open source project - http://github.com/robey/kestrel

17. http://blog.lostlake.org/index.php?/archives/45-A-real-world-use-of-lift.html#extended

18. Viswa Viswanathan, "Rapid Web Application Development: A Ruby on Rails Tutorial", IEEE Software, vol. 25, no. 6, pp. 98-106, Nov./Dec. 2008, doi:10.1109/MS.2008.156

19. Mike Jordan, "A comparative study of persistence mechanisms for the Java platform", Sun Microsystems Laboratories Technical Report: SERIES13103, June 2000.

20. David Geary, Cay Horstmann: "Core Java Server Faces", Second Edition, Prentice-Hall, 2007 ISBN 0-13-173886-0

21. http://www.theserverside.com/news/1364757/Hibernate-vs-Rails-The-Persistence-Showdown

22. http://weblogs.java.net/blog/robc/archive/2009/12/01/java-ee-6-platform-approved-today

23. JSR 299: Contexts and Dependency Injection for the Java EE platform - http://jcp.org/en/jsr/detail?id=299

24. JSR 314: Java Server Faces 2.0 - http://www.jcp.org/en/jsr/detail?id=314

25. Scala in the Enterprise - http://www.scala-lang.org/node/1658

26. http://scalate.fusesource.org/documentation/scalate-embedding-guide.html

27. Play framework using Scala - http://www.playframework.org/documentation/1.1-trunk/scala

28. Scalate - http://scalate.fusesource.org/

29. A tour of Scala: Annotations - http://www.scala-lang.org/node/106

30. G. Dubochet, "Computer Code as a Medium for Human Communication: Are Programming Languages Improving?", 21st Annual Psychology of Programming Interest Group Conference, pages 174-187, Limerick, Ireland, 2009.