WHITEPAPER

INGRES

DEVELOPING APPLICATIONS USING AN OPEN SOURCE TECHNOLOGY STACK

BY DOMENIC MANGANO

INGRES CORPORATION

TABLE OF CONTENTS:

1 INTRODUCTION – WHY OPEN STANDARDS?

Today, there is a wide array of technologies and tools available to software developers that can be used to develop web based applications. When selecting products and tools to build and support their IT systems, enterprises need to rely on a proven foundation to be assured that their applications can be built in accordance with good architectural principles and goals.

Technologies built upon open standards typically encompass good architectural traits as they, by definition, need to promote interoperability. Building applications upon open standards provides a level of insurance to enterprises that means they need not be locked into a specific vendor's product and that the investment in an application need not be lost should the enterprise decide to change some part of their infrastructure (e.g. an application server or database server).

In order to achieve the maximum benefit of developing applications to open standards, some design discipline must be applied. For a decade or more, building applications using a layered architecture has become a ubiquitous principle, because it encourages a clear separation of responsibility between the components in an application, such as locating business logic in a dedicated layer and not spread throughout the application. This approach is beneficial in decoupling the core of an application from the User Interface (UI) and data storage technologies used by the application. This, in turn, makes the application more flexible to change than would otherwise be possible, which is a key trait for enterprise applications.

Best practice now recommends that application developers should use standards-based interfaces, because this both increases the portability of an application and lowers the cost of maintenance through the availability of developers with more generalised skills. A common example of where this discipline is not applied is where enterprises put most of their business logic into stored procedures rather than application code. This makes both maintenance (scaling the application becomes expensive because application servers are cheaper than database servers, per CPU) and portability (because stored procedures are always proprietary) more costly than necessary. On occasion it may make sense for specific application functions to use proprietary features of a particular product (e.g. where there is a definite performance benefit), but such instances should be the exception rather than the rule.

In recent times enterprises have come to realise the benefits that Open Source software can offer. Widely used Open Source products, such as Red Hat Enterprise Linux and Ingres, have allayed prejudices about reliability and security flaws which might have resulted from the Open Source development model. Best of breed products that are developed under an Open Source license can be every bit as reliable, secure and scalable as their commercial counterparts.

1.1 Advantages of an Integrated Stack

As with the commercial software market there is a bewildering choice of competing products in the Open Source software market. Deciding on a combination of Open Source products that can be used to form an enterprise-capable software stack can be a formidable task and not without risk, for example how can enterprises know that the selected products will work well when combined? In answer to this problem, some Open Source product vendors partner to offer a complete application platform that is already integrated and tested, thus reducing this risk.

When enterprises select an application platform that is based on pre-integrated Open Source software, they gain the benefits of enhanced productivity and predictability. Open Source products have frequently been used by enterprises during the development of their applications - even if the application will eventually run on commercial products in a different production environment - because this reduces some of the drain of software license costs. With enterprises looking to lower the overall cost of their IT systems, the up-front license costs associated with commercial software products are one source of cost that offers no business benefit for an enterprise, as additional fees still need to be paid to receive support from the product vendor. For this reason, many enterprises are today looking to replace commercial products with Open Source alternatives for production use as well as in development.

In addition to these cost related arguments, Open Source software has also proven itself in terms of supportability, security, resilience, performance, high availability and other attributes needed to qualify as being enterprise class. This means that it is now being used much more frequently as the basis of true enterprise class application infrastructure.

Using a pre-integrated software stack in this scenario delivers productivity gains partly because developers and system administrators need to spend less time installing and configuring disparate products, and partly because using the same products from development through production reduces the effort associated with testing and configuring the application for the different environments. By using a supported stack, enterprises looking to replace their software infrastructure with Open Source software will see the additional benefit of predictability. Using the same application platform in development, pre-production and production environments reduces the possibility of differences between the application server and database products introducing unforeseen problems.

This white paper is a short tutorial that outlines how to set up and develop a simple layered application using the JBoss open source application technology platform based on the Java Enterprise Edition platform (JEE), and using the Ingres RDBMS for persistent data storage. This tutorial application will run on JBoss middleware and make use of the JBoss Seam framework.

Both Ingres and JBoss middleware are used by enterprises in mission and business critical applications so can be considered as mature and proven technologies and consequently a good foundation upon which applications can be developed and run in production environments.

1.2 Background to Products Used Within the Stack

The JBoss Server project has been active since 1999. Over that time JBoss Application Server (JBoss AS) has become the most widely used Java Enterprise server, and has moved with the times with the objective of supporting the most current JEE specifications. At the time of publication the most current version of JBoss AS is 5.0, which complies with the JEE 5 specifications.

The Ingres RDBMS dates back to the inception of relational database technology. Based on code originally developed as part of a research project at University of California, Berkeley, the database product continued to be developed as a commercial entity until it was open sourced by Computer Associates in 2004. Over its lifetime, the core database has maintained a strong adherence to industry standards such as SQL and provides a wide range of connectivity options including JDBC, ODBC, Python, Perl, PHP and Ruby.

The JBoss Seam framework is comparatively new, and primarily exists to enhance the JEE programming model by providing additional features to support more complex use cases that can arise with fully featured Rich Internet Applications (RIA). The Seam framework integrates AJAX, Java Server Faces (JSF), Facelets, Enterprise Java Beans, Java Persistence API (JPA), Java Portlets and Business Process Management to provide a fully featured web development framework.

Seam builds upon core JEE technologies but incorporates additional technologies to offer a unified programming model and more flexibility targeted at web applications. Seam treats JSF as a first class citizen and enhances the JSF approach to provide useful features not yet covered by the JSF specification. Additional features include contextual state management, support for RESTful URLs, an enhanced exception handling mechanism and a convention over configuration approach that alleviates most of the burden associated with creating and maintaining XML configuration files.

2 PREREQUISITES

The practical elements of the tutorial assume the reader has basic experience of the following:

- Developing Java applications with Eclipse and the Web Tools plugin
- Familiarity with JEE, primarily web components (JSF, Servlets, JSP), EJB and JPA entities and JEE deployment modules
- Familiarity with Facelets

The following software components are used in the tutorial - please refer to the Resources section for information on downloading the software:

- Java Development Kit (1.5 or above)
- Ingres RDBMS 9.2.0
- JBoss AS 5.0.0 GA

- JBoss Seam 2.1.0 SP1
- Eclipse Ganymede JEE Build
- JBoss Tools 3 CR1 (an Eclipse plugin)

It is suggested that the software be installed in the order presented in the list above, however with the exception of the JBoss Tools Eclipse plugin which requires that Eclipse is installed and running, there is no strict order in which the software must be installed. The Eclipse IDE comes bundled with an embedded Java Runtime so the IDE will run without an external Java Runtime or Development Kit. Note that the last four components are also packaged together within the commercially available JBoss Developer Studio.

Source code for the domain entity classes and a pre-built EAR module with the complete example application, along with deployment instructions, can be found at:

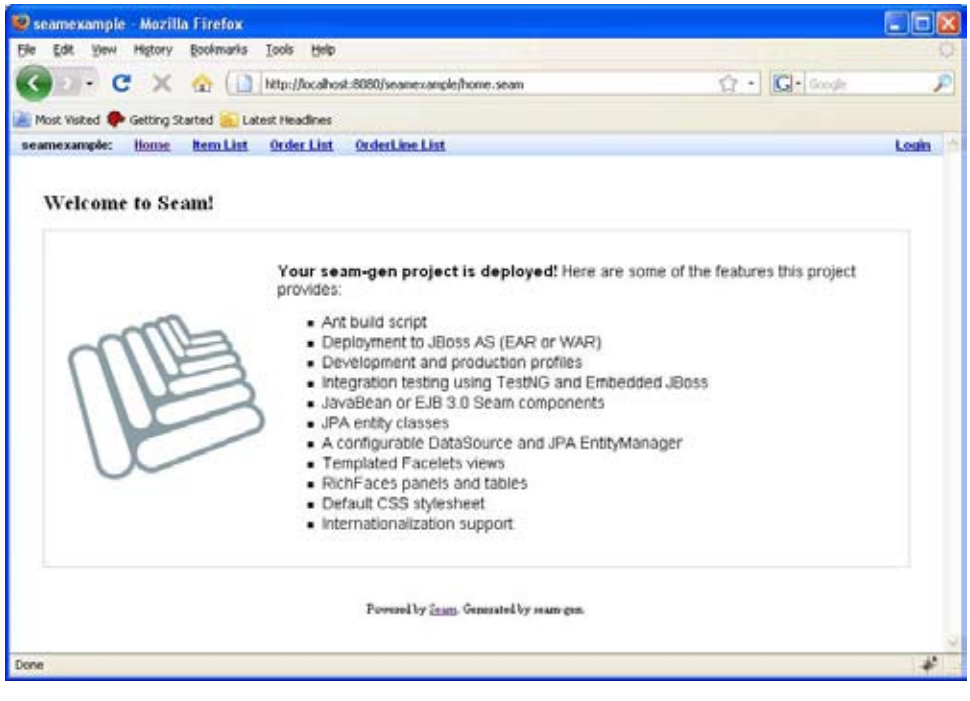
http://community.ingres.com/wiki/JBossToolsSeamExample

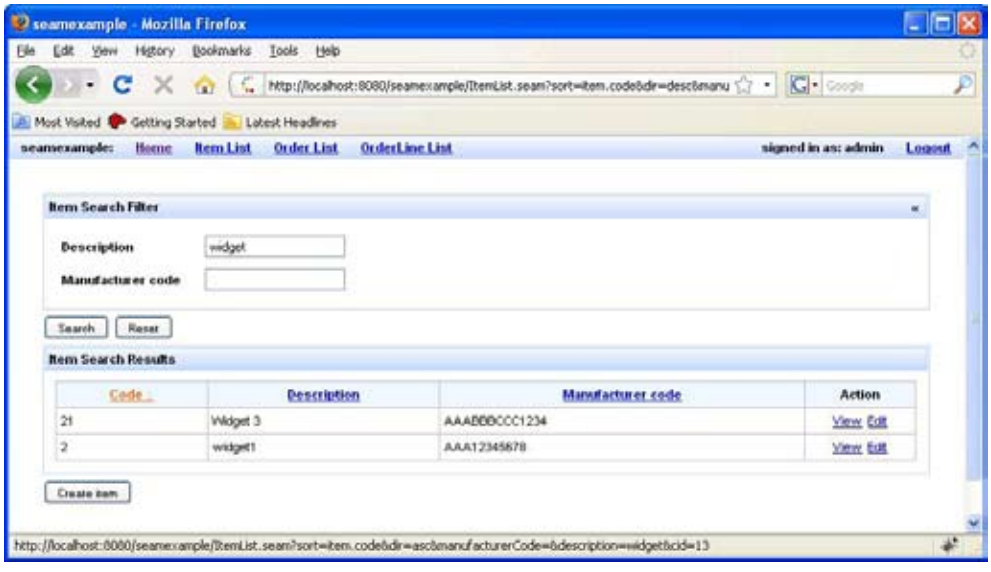
3 TUTORIAL

This section includes a step-by-step guide to creating a skeleton Seam application using the Eclipse IDE which can serve as a starting point for developing a fully featured web application.

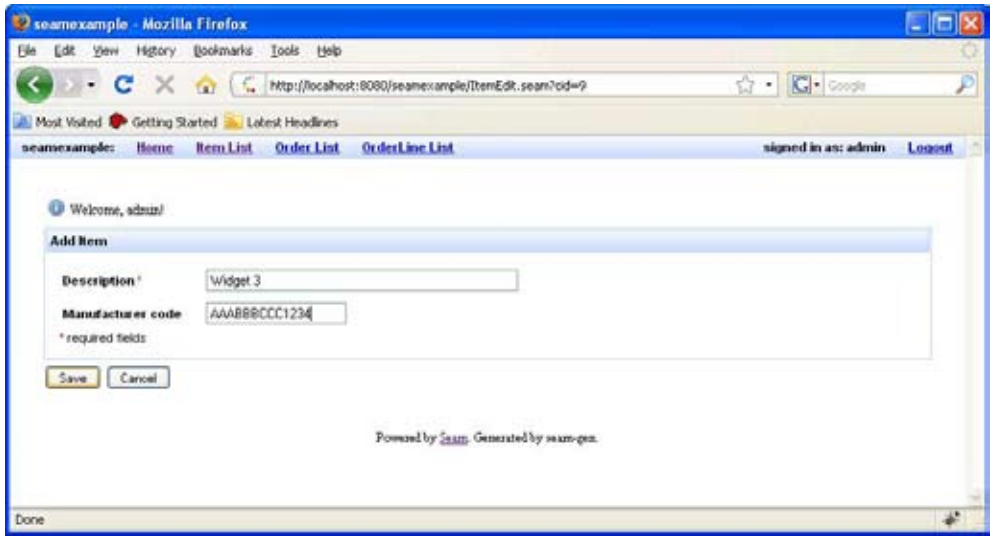
The application allows a user to view and edit products and orders via list and detail web pages.

The home page is a simple default page that outlines features offered by Seam. The hyperlinks at the top of the page are used to navigate to the screens we will create.



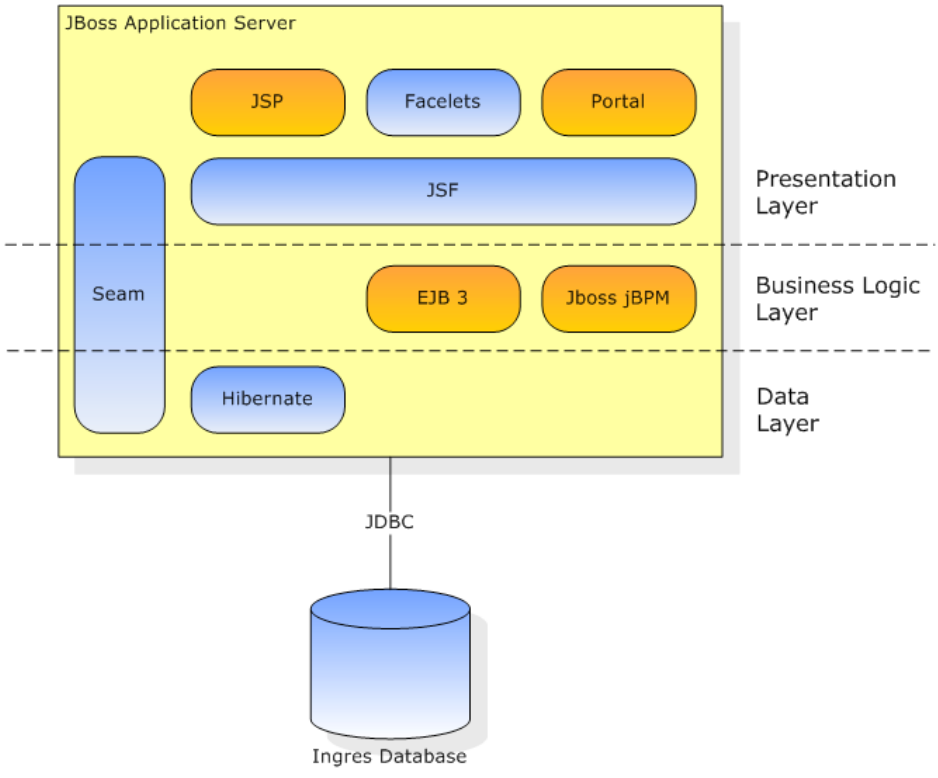


The detail page can be used to add new entities to the system or editing existing entities.



3.1 System Architecture

The following diagram outlines the core architectural components of a Seam application. A subset of the components will be used for the sample application (the components coloured blue).



The role of each architectural component can be described as:

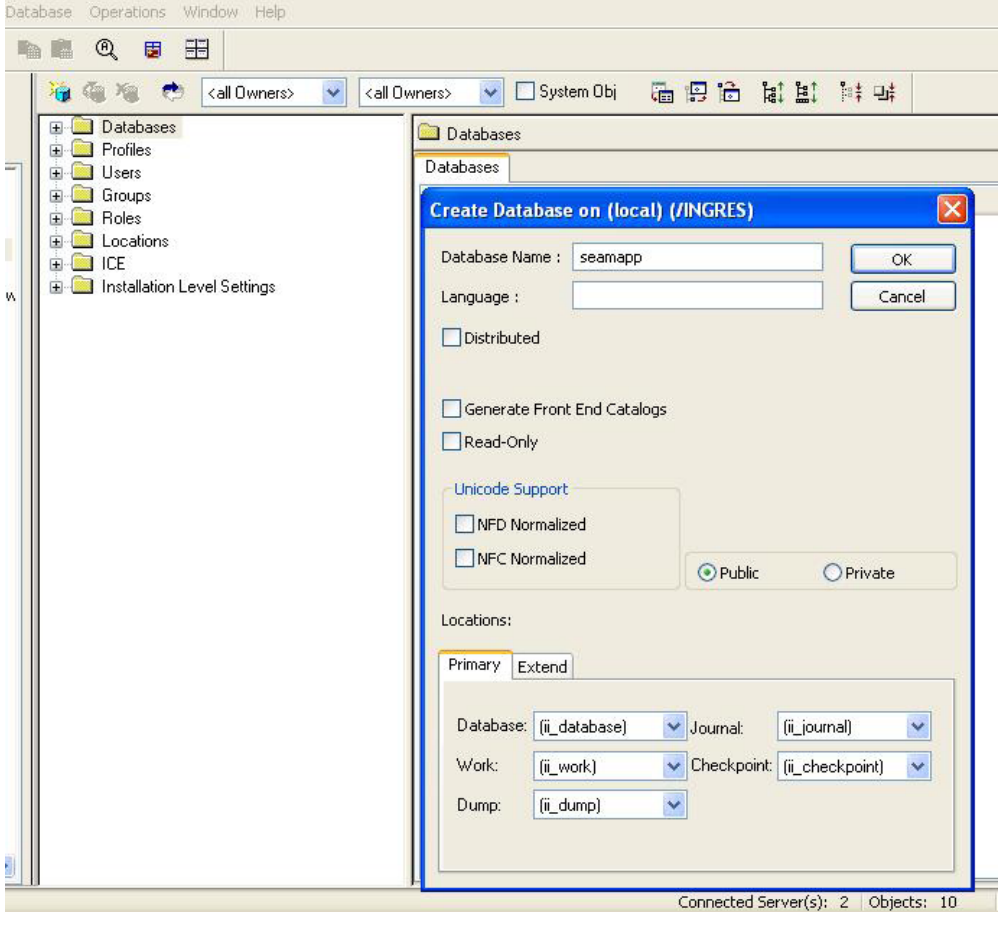
- Facelets – Assembles the JSF UI components to form the view to be presented in response to a client request;
- JSF – Provides the UI components used to construct the user interface;
- Seam – The Seam framework provides functionality that can straddle each layer of the application. A primary function of Seam is the management of requests from the client (browser), directing requests to an appropriate controller and ensuring that the controller is in an appropriate state to service a request (e.g. component dependencies are satisfied). The dependency and state management functionality provided by Seam should be viewed as an infrastructure function as these features might be used in other layers of the application;
- Hibernate – Manages the ORM mapping layer, allowing the application to use an object interface and interact with the database;
- JBoss AS – The application server is an infrastructure component that provides services to hosted applications. The application server is a host for the Seam application providing additional services such as transaction management and security;
- Ingres – Persistent storage of the application data.

3.2 Create the database

The first step is to create the database to be used for the tutorial. The database is to be called *seamapp*.

Database creation on Ingres can be achieved in one of two ways, using either the Visual DBA tool or the **createdb** utility.

If using Visual DBA, connect to the server by selecting the database server in the Nodes explorer and using the Node -> Connect menu. This will display the object browser for the database server. Right-click the 'Databases' node in the database objects browser. Complete the relevant details in the wizard and click OK.



If using the createdb command-line utility, the following command is used to create the database:

createdb seamapp

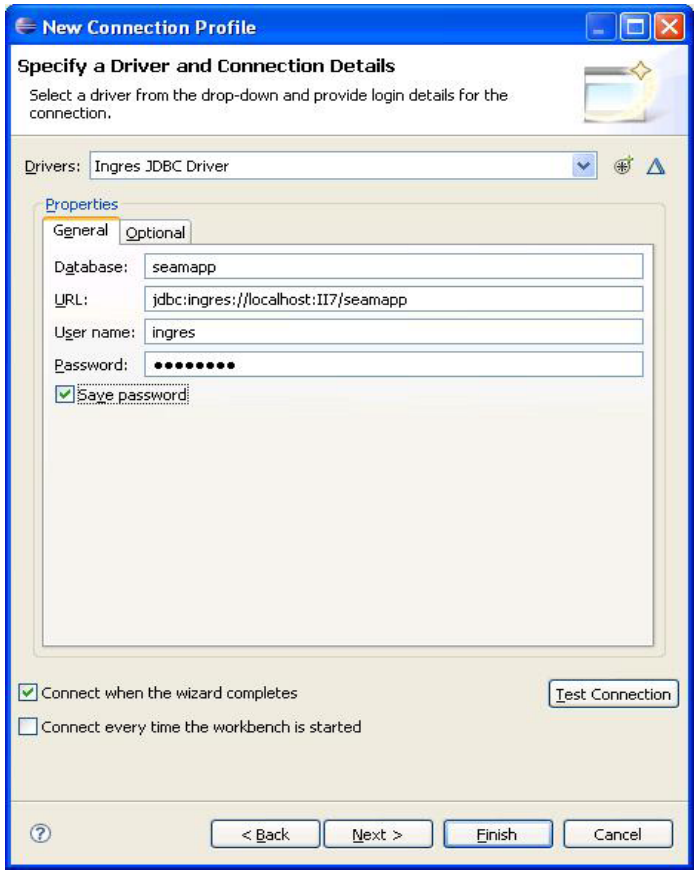
3.3 Register the database with the Eclipse Data Source Explorer

The Eclipse Data Tools plugin provides the capability to explore and interact with JDBC compliant databases from within the IDE. It provides a subset of features offered by the Ingres Visual DBA tool but with the convenience of being able to access these directly within the IDE.

The Ingres JDBC driver must be registered with Eclipse to allow the Data Source Explorer to display the database. This can be done prior to registering the database as outlined in Appendix B of this document but can also be done at the same time a new Connection Profile is created. The following steps assume that the Ingres JDBC driver is already registered.

Ensuring that the 'Data Source Explorer' view is currently visible, right click the Databases node and select the 'New…' option to create a new Connection profile.

Select Generic JDBC from the list and change the name to something suitable such as 'Ingres DB'. Progress through the wizard and complete the details as appropriate. The user login details required are those of an operating system user with Ingres access permissions, on the machine hosting the database server, and by default you should enter the details of the user who installed Ingres. The connection can be tested using the 'Test Connection' button available on the second page of the wizard.



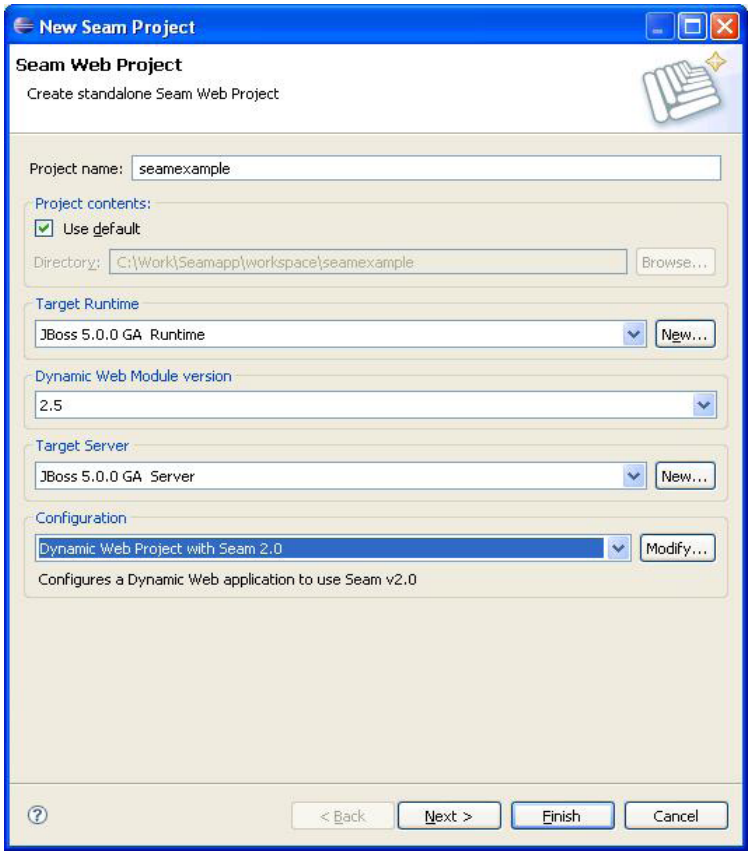
The Data Source Explorer can be used to examine database table structures and any data contained within the tables.

3.4 Create a new Seam project

The Seam Web Project wizard, included with the JBoss Tools plugin, is to be used to create the skeleton application structure. An alternative approach would be to use the 'seam-gen utility' that is included as part of the Seam installation.

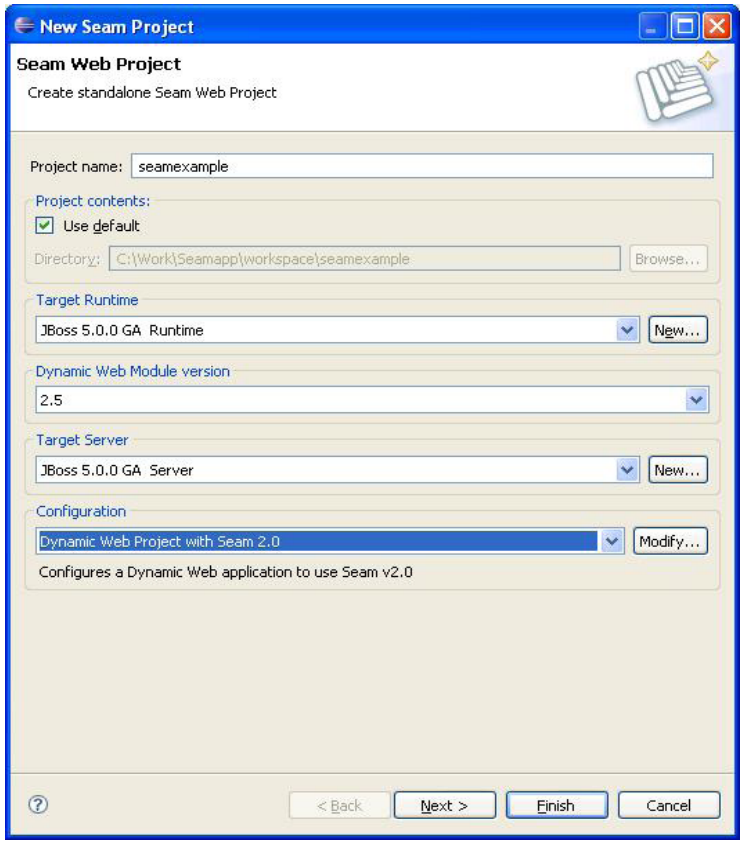
First ensure that the Seam Perspective is enabled. This can be done via the Window -> Open Perspective -> Other dialog and selecting Seam from the listing. With the Seam perspective enabled, the Seam Web Project wizard can be accessed from the File -> New menu.

The application will be named 'seamexample'. Complete the wizard details as required, being sure to provide a name for the project and that the selected 'Configuration' option is a 'Dynamic Web Project with Seam 2.0'. The default choices should be acceptable provided a JBoss runtime and server have been previously configured as outlined in Appendix B.

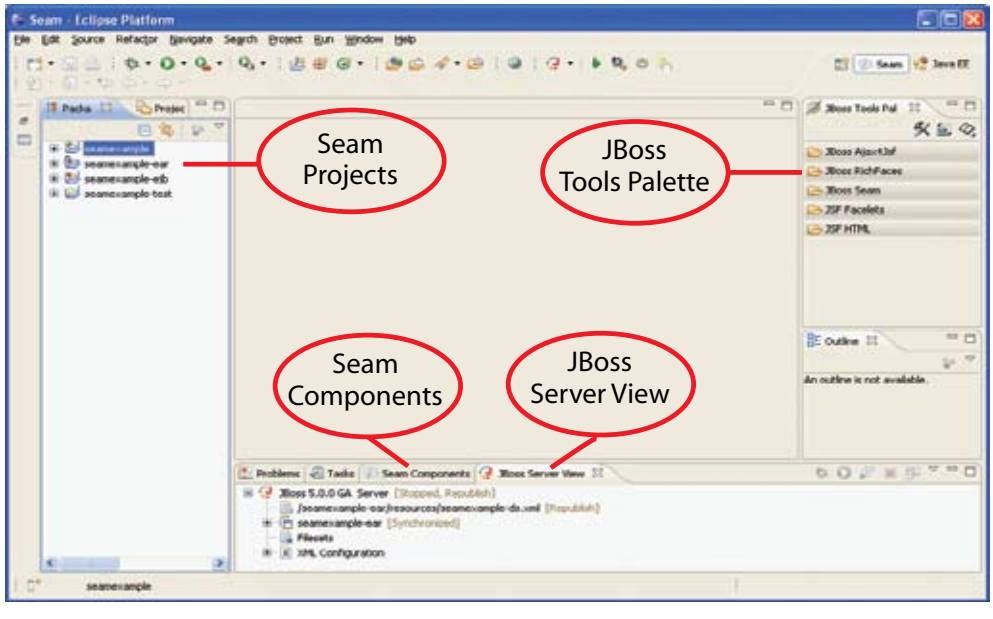


Proceed through the next two wizard pages, accepting the default options.

Proceed to the final page of the wizard and complete the remaining details. For the General options, select the Seam runtime (note that the CR release of the JBoss Tools plugin did not recognise a previously installed instance so a new one had to be created). Complete the remaining details enabling the 'EAR deploy' option, and for the database options select the Connection profile created earlier.



At this point it is worth reviewing what has been created by the project generation wizard.



Firstly examining what can be seen in the Package Explorer view.
The Seam Web Project wizard has created 4 separate Eclipse projects artefacts.

- The seamexample project is created as a WTP Dynamic Web Project configured to include the Seam facet. Building this project will create a WAR archive. Components that will be included in this project include the Facelets view templates and Seam configuration files.
- The seamexample-ejb project is created as a WTP EJB project. Building the project will create a Jar file. Components that will be included in this project include Session EJBs and any JPA Entities.
- The seamexample-test project is a standard Java project that will include all of the TestNG test cases.
- The seamexample-ear has created a WTP EAR archive project that will build the JEE enterprise archive to be deployed to the server. The Ear archive will include the build artefacts from the Web and EJB projects.

3.5 Seam Perspective Features

With the Seam perspective enabled it should be possible to see a couple of design aids provided by the JBoss Tools plugin, as highlighted on the screenshot above.

3.5.1 JBoss Tools Palette

This view is a palette of UI components that can be used during the design of the JSF UI. JBoss Tools provides a visual designer for building JSF UIs, where the palette components can be dragged onto a form designer to build up the UI.

3.5.2 JBoss Server View

This view is an extension of the WTP plugin's Server view. It facilitates control of server instances registered with the IDE and provides the capability to:

- Start and stop the server;
- Start the server in debug mode;
- Change the startup configuration for the server, and;
- View applications deployed in the server.

The JBoss Server view allows the server configuration file (jboss-service.xml) to be edited from within the IDE.

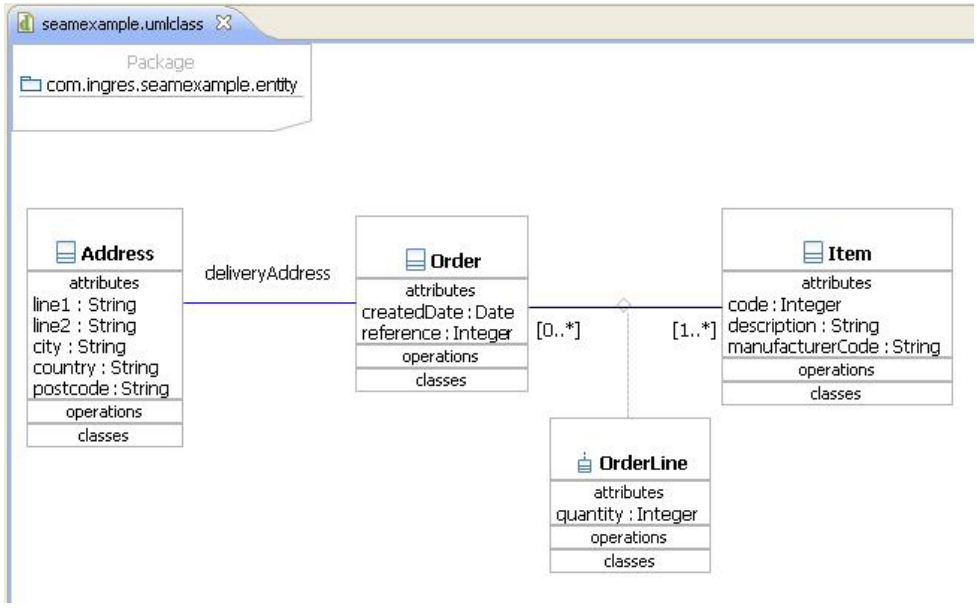
3.5.3 Seam Components

This view displays the contents of the components.xml file included in any Seam projects, supporting the visual editing of this file.

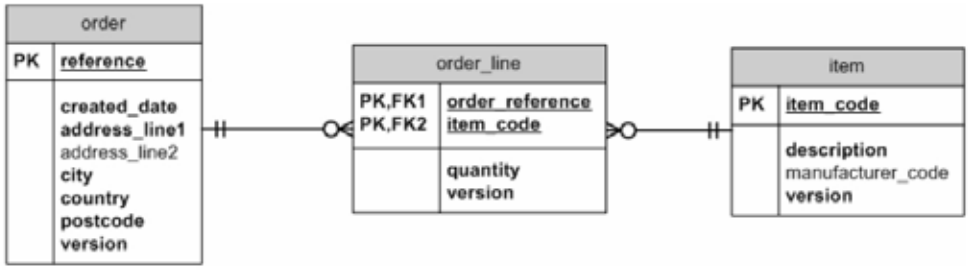
As will be shown in the next section, the Seam perspective offers handy wizards for creatingnew Seam artefacts to be included in the application. When non-Seam components are required, switching to the Java EE perspective can be useful to get access to context sensitive content creation wizards, for example EJB creation wizards for EJB projects or Servlet or Filter creation wizards for Web projects. Otherwise, these wizards can be accessed using the 'Other' option that should be visible from the 'New' context menu.

3.6 Creating Seam components

It is now time to create the components for the application. Like 'seam-gen', JBoss Tools provides the ability to generate most of the artefacts needed for a simple data maintenance function. Artefacts can be generated from a set of existing JPA entities or an existing database schema. The tutorial will focus on using the first approach, of first creating the application domain model, annotating the model classes and then using JBoss Tools to generate the UI templates from the domain classes. The code generation is a one-off process so it is better to spend some time getting the domain model as near as possible to what will be required for the application. The domain model to be used is detailed by the following UML class diagram.



The database Entity-Relationship model is detailed in the following diagram.



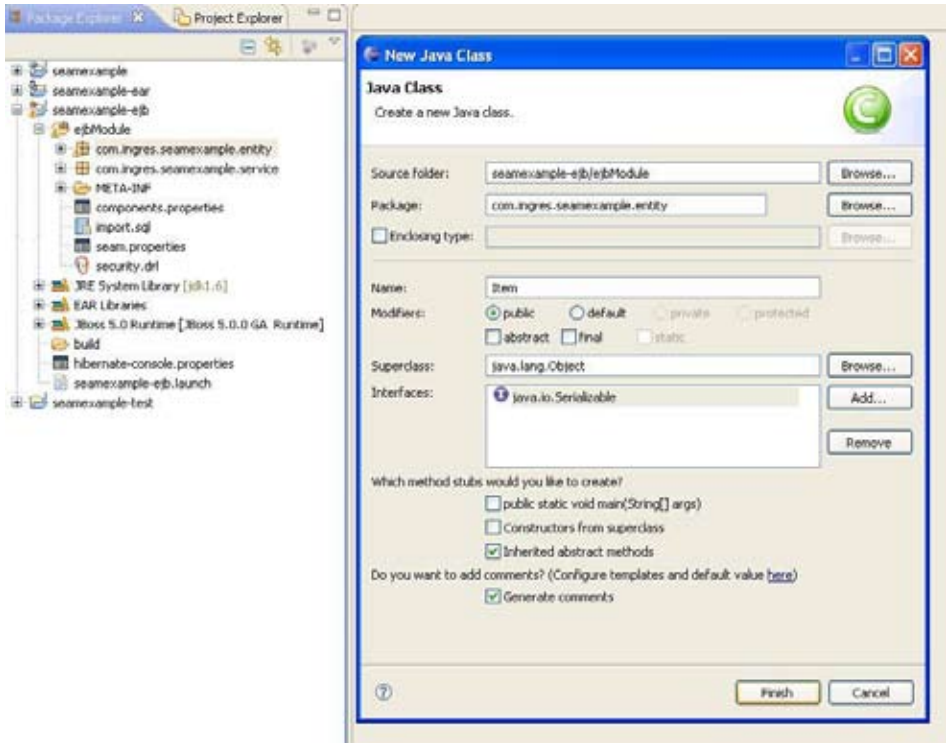
The process to be used to create the entity classes for the application will be:

- Create a new Java class for the entity
- Add JPA annotations
- Add Hibernate Validator annotations

The full source code for the domain classes can be found in the appendix. In the following section, code snippets are included to highlight the use of JPA and Hibernate Validator annotations.

3.6.1 Creating the Entity classes

Create a new class using the Eclipse class wizard. Note that the CR version of the JBoss Tools wizard did not create the empty com.ingres.seamexample.entity package, so this can be either be created prior to the creation of the new class or can be created by the New Java Class wizard providing the package field is first populated with the package name.
If the package exists, right-clicking against the package in the 'Package Explorer' view and selecting New->Class will initialise the New Java Class wizard with the current package information.



Create new classes for the Item, Order and OrderLine entities and also for an OrderLineId class, which is not contained within the UML class model but which will be used as the OrderLine identifier and will contain the compound primary key values that form the OrderLine primary key. Note that the classes should also implement the java.io.Serializable interface.

3.6.2 Adding JPA / Hibernate Validator Annotations

In this section annotated code fragments from the entity classes will be included with a description of the annotations that have been added to create the Object-Relational Mapping (ORM) and data validation constraints.

First up is the Address class. This, in fact, is not modelled as an entity class at all, but instead it has been mapped as value type that will be associated with an Order entity.

```
@Embeddable
public class Address implements Serializable {

	private static final long serialVersionUID = 1L;

	@Column(name = "address_line1", length = 40)
	     @NotEmpty
	private String line1;

	@Column(name="address_line2", length = 40)
	private String line2;

	@Column(length = 30)
	@NotEmpty
	private String city;

	@Column(length = 30)
	@NotEmpty
	private String country;

	@Column(length = 9)
	@NotEmpty
	@Pattern(regex = "[A-Z]{1,2}[0-9]{1,2}? [0-9]{1,2}[A-Z]{1,2}")
	private String postcode;

	     … get/set methods
}
```

The Address class is marked with an @Embeddable annotation. Embeddable classes are sometimes referred to as component or value types because the values for an Address object are to be persisted but the object does not have a unique identity within the system.
The Item class is presented next.

```
@Entity
public class Item implements Serializable {

	private static final long serialVersionUID = 1L;

	@Id
	@GeneratedValue
	private Integer code;

	@Column(name = "manufacturer_code", length = 20)
	private String manufacturerCode;

	@Column(length = 50)
	@NotEmpty
	@Length(max = 50)
	private String description;

	@Version
	private Timestamp version;

	… get/set methods

}
```

The @Entity annotation marks this as a persisted object that requires a unique identifier within the system.

The @Id annotation marks this field as the identifying property of the entity. The @GeneratedValue annotation specifies that the identity value is to be provided by some other component the first time the entity is persisted. JPA supports a number of key generation strategies, and sticking with the default generation strategy (GeneratorType.AUTO) will result in the default generation type specified by the dialect for the target database. In the case of Ingres, sequences will be used as the generation mechanism.

The @Column annotation specifies the field mapping between properties of the class and columns of the related database table, which will default to be named 'item' (as no @Table annotation to override the default table name mapping has been included). The @Column annotation allows various aspects of the column mapping to be specified (e.g. column length, nullability) and these become relevant when the database schema is created from the mapping metadata.

The @NotEmpty is a Hibernate Validator feature that specifies the property value can be neither null nor an empty value.

The @Length annotation is a Hibernate Validator feature that can be used to specify minimum and maximum length of a String.

The @Pattern annotation is a Hibernate Validator feature that in this instance uses an example regular expression to validate the postal code according to UK rules (provided just as an example).

The @Version annotation specifies that an optimistic locking strategy is to be used to control concurrent updates of the entity. The mapped property is typically an integer number or timestamp.

The code for the Order class is next.

```
@Entity
public class Order implements Serializable {

      private static final long serialVersionUID = 1L;

      @Id
      @GeneratedValue
      private Integer reference;

      @Column(name="created_date")
      @Temporal(TemporalType.DATE)
      @NotNull
      private Date createdDate;

      @Embedded
      @NotNull
      private Address deliveryAddress;

      @OneToMany(mappedBy = "order", cascade = CascadeType.REMOVE)
      private Collection<OrderLine> orderLines;

      @Version
      private Timestamp version;

      public Order() {
              this.setDeliveryAddress(new Address());
              this.setOrderLines(new ArrayList<OrderLine>());
      }

      … get/set methods

}
```

The Order class contains a number of new annotations.

The @Temporal annotation is used to specify the mapping of a date and time value. The range of possible values originates from the TemporalType enumeration and can be a date, time or timestamp value.

The @NotNull annotation is a Hibernate Validator feature that requires the property to have a value.

The @Embedded annotation is used to identify the class as a value type whose mapped fields are to be persisted as part of the entity so in this case the mapped fields from the address object will be persisted within the order table.

The @OneToMany is a collection type mapping, linking an Order to many OrderLine instances. The 'mappedBy' attribute identifies the property on the OrderLine class that this entity is mapped to and the 'cascade' attribute specifies what is to happen to the child dependency when the Order entity is modified.

The OrderLine class has been modelled as an Association Class. The association between Order and Item could have potentially been mapped as a bi-directional many-to-many association but as the association includes additional attributes (e.g. quantity) then a separate entity is preferred.

```
@Entity
@Table(name = "order_line")
public class OrderLine implements Serializable {

      private static final long serialVersionUID = 1L;

      @EmbeddedId
      @NotNull
      private OrderLineId id;

      @ManyToOne(fetch = FetchType.LAZY)
      @JoinColumn(name = "item_code",
                  nullable = false,
                  insertable = false,
                  updatable = false)
      @NotNull
      private Item item;

      @Column(nullable = false)
      @Min(1)
      private int quantity;

      @ManyToOne(fetch = FetchType.LAZY)
      @JoinColumn(name = "order_reference",
                  nullable = false,
                  insertable = false,
                  updatable = false)
      private Order order;

      @Version
      private Timestamp version;


        … get/set methods

}
```

The @Table annotation illustrates how to override the default table name mapping.

Use of the @EmbeddedId annotation is described further on.

The @ManyToOne identifies the property as the many end of a many-to-one association. The identity value could be inferred from the associated property type but the database mapping has been altered to use more intuitive column names for the database table. The @JoinColumn annotation is used to describe the mapping. Setting the nullable, insertable and updatable attributes to false ensures the entity cannot be directly persisted to the database but must first be associated with valid Order and Item entities before it may be persisted.

The @Min annotation is a Hibernate Validator feature used to ensure the quantity ordered for a specific item is one or more.

A composite identifier is required for the OrderLine entity, so the @EmbeddedId is used to specify a separate class is to be used as the entity identifier. In the case of the OrderLine class the identity value is derived from the associated Item code and Order reference values. This is represented in the database by the order_line table requiring a compound primary key mapping based on foreign key references to records in the order and item tables. There are a couple of

ways to specify a compound key mapping with JPA, but in this case a separate class OrderLineId (the code for which is provided below) has been created to model the identity. The class is marked as @Embeddable and must adequately override the equals and hashCode methods from the java.lang.Object class.

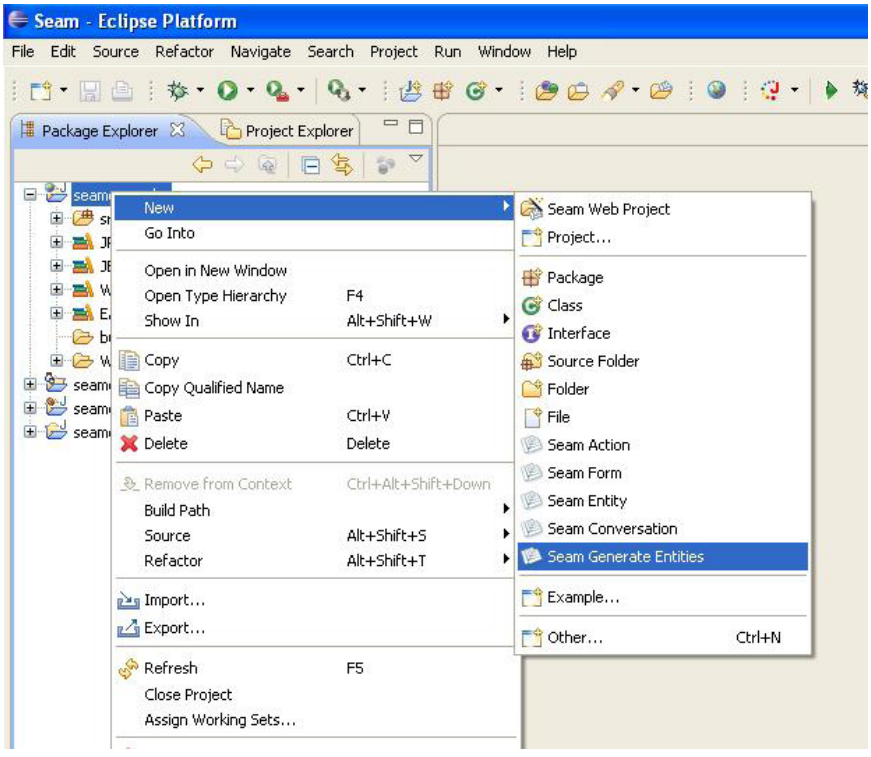
```
@Embeddable
public class OrderLineId implements Serializable {

      private static final long serialVersionUID = 1L;

      @Column(name = "order_reference", nullable = false)
      private Integer orderReference;
      @Column(name = "item_code", nullable = false)
      private Integer itemCode;

        … get/set methods


}
```

This completes the creation of the domain model. As a general point it should be remembered that where possible good object-oriented design principles such as encapsulation should be followed as closely as possible. It is a good practice to only expose as much of the entity class as necessary to client code, for example, the version property is only of interest to the ORM tool and should never be set directly by front-end code so this property should use the most restrictive access level which in the case of JPA is the protected access level. Similarly, if the ORM tool is exclusively responsible for assigning new entity identities then make the mutator (set) method as restrictive as possible.

3.6.3 Generate the Seam artefacts

The code generation features provided by the 'seam-gen' utility and JBoss Tools are a great way to quickly produce a prototype for an application. The code generation procedure will use the JPA mapping and Hibernate Validator annotations present in the entity classes to influence the generation process. For example, JPA Collection type associations will be displayed as tabular entries in the page of the parent entity and front-end validation constraints will be derived from the validation annotations.

To generate Seam artefacts using JBoss Tools, start the 'Seam Generate Entities' wizard. Assuming the Seam perspective is enabled, this can be displayed by selecting the Seam project in the Package Explorer view, right-click to display the New menu and select the Seam Generate Entities option.



In the resulting wizard, select 'Use existing entities' and click Finish, leaving the remaining options at their defaults. As the Seam project was configured to be an EAR project, JBoss Tools will expect the entity classes to be located in the seamexample-ejb project.

At this point it is worth examining the artefacts produced by the code generation process.

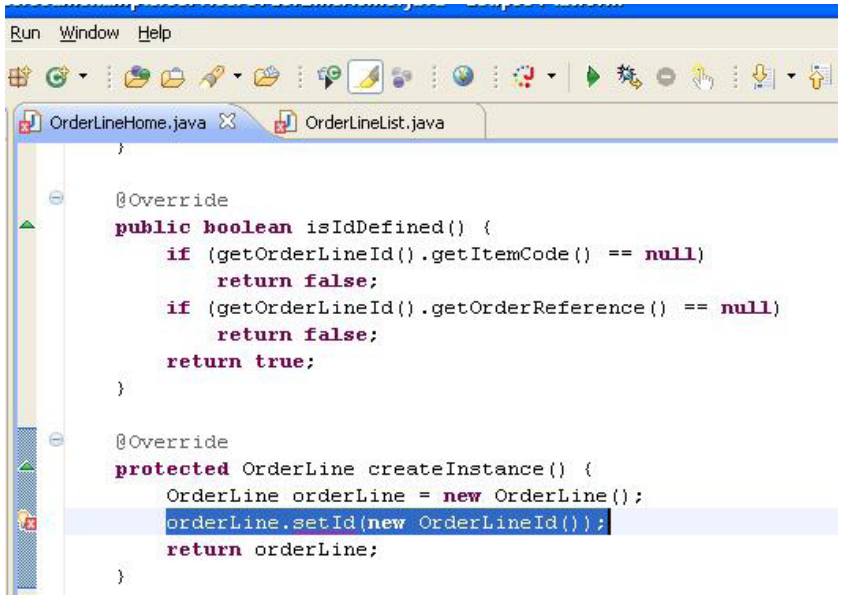
In the seamexample-ejb project, look in the com.ingres.seamexample.service package. The package will contain a Home and List class for each entity. These extend classes provided by the Seam Application Framework to provide data maintenance (CRUD) capabilities for the entities.

The accompanying UI artefacts will be located in the WebContent directory of the seamexample project. The generation process will have created Edit and List Facelets view templates (.xhtml files) for each entity class and also a Seam pages.xml file which can be used to define fine-grained rules to dictate the flow between pages. A menu entry will also have been added to the WebContent/layout/menu.xhtml template.

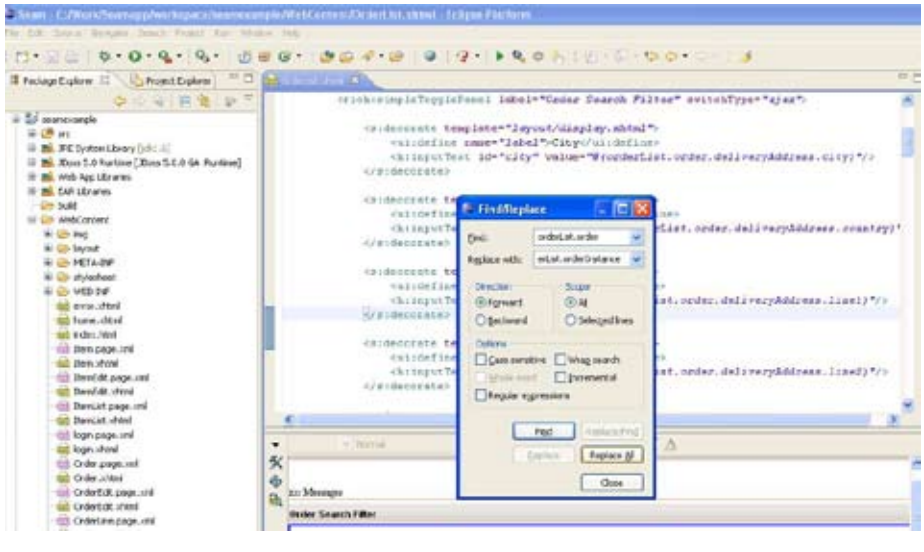
3.6.4 Verify the generated code

The generated code will only serve as a starting point for an application. It is probable that the code will need to be checked and possibly modified to get the application behaving as required. For example, the generated pages may need amending to display fields in a preferred order or layout, and the generated classes will need to be checked for compilation errors that may occur if the class or property names result in conflicts.

In the case of the domain model used in the tutorial some code modifications will be necessary. The OrderLineHome and OrderLineList classes in the com.ingres.seamexample.service package will need to be modified, because the generated classes include a call to OrderLine's setId method and the OrderLine class restricts access to this method so this results in a compilation failure. As the default constructor of the OrderLine class ensures that the id property is instantiated, the assignment of an empty id value by the OrderLineHome and OrderLineList classes is redundant and so should be removed.

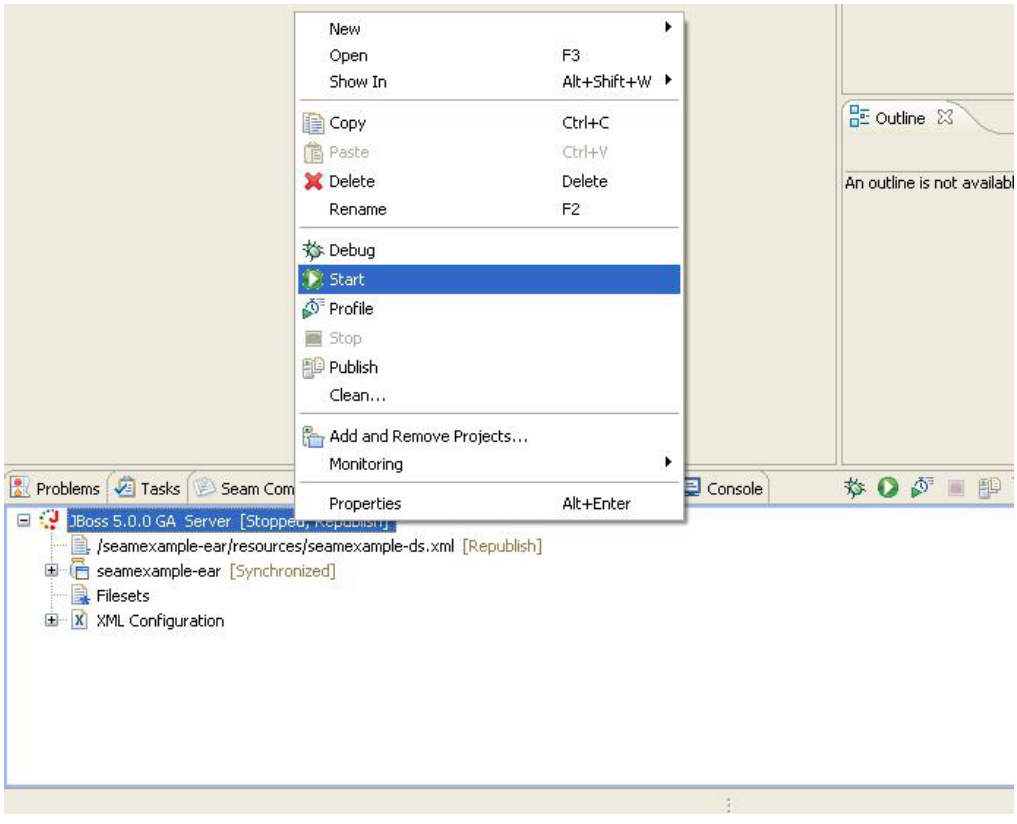


The use of Order as a class name causes a conflict in the OrderList class with the inherited getOrder method of EntityQuery (which is a Seam Application Framework class). The getOrder method in OrderList needs to be renamed to something else (e.g. getOrderInstance) to resolve the conflict. Any action bindings in the generated UI templates (OrderList.xhtml) will also need to be updated to call the renamed method.



3.7 Running the application

The JBoss Tools plugin will deploy the EAR module to the JBoss server identified during the project creation step. The server can be started in the JBoss Server View by selecting the server instance and using the context menu options or the icons at the top right-hand side of the view.



Once the application server has started the home page for the application can be accessed using the URL:

http://localhost:8080/seamexample

3.8 Summary

This tutorial has demonstrated that with appropriate technology choices and tools, it is possible to undertake rapid application development without sacrificing important enterprise features that ensure that mission critical applications can run reliably, securely and scale to meet user demand.

The Seam framework is a fully featured web application framework that is an Open Source product that builds upon open standards. This allows the application to be deployed to any number of JEE application servers, either lightweight containers such as Tomcat or fully fledged application servers like JBoss that offer distributed transaction management, messaging capabilities and clustering features for fault-tolerance and scalability. The abstraction offered by JPA on JDBC prevents lock-in to a specific persistent storage mechanism, allowing the database server to be swapped for another JPA capable solution with little or no code changes, hence delivering the benefit of avoiding vendor lock-in at both application server and database levels.

APPENDIX A DOMAIN MODEL SOURCE

Address.java

```
package com.ingres.seamexample.entity;

import java.io.Serializable;

import javax.persistence.Column;
import javax.persistence.Embeddable;

import org.hibernate.validator.NotEmpty;
import org.hibernate.validator.Pattern;

/**
 * Represents a UK property address.
 */
@Embeddable
public class Address implements Serializable {

	private static final long serialVersionUID = 1L;

	@Column(name = "address_line1", length = 40)
	@NotEmpty
	private String line1;

	@Column(name="address_line2", length = 40)
	private String line2;

	@Column(length = 30)
	@NotEmpty
	private String city;

	@Column(length = 30)
	@NotEmpty
	private String country;
	@Column(length = 9)
	@NotEmpty
	@Pattern(regex = "[A-Z]{1,2}[0-9]{1,2}? [0-9]{1,2}[A-Z]{1,2}")
	private String postcode;

	public String getLine1() {
		return line1;
	}
	public void setLine1(String line1) {
		this.line1 = line1;
	}
	public String getLine2() {
		return line2;
	}

	public void setLine2(String line2) {
		this.line2 = line2;
	}

	public String getCity() {

		return city;
	}

	public void setCity(String city) {
		this.city = city;
	}

	public String getCountry() {
		return country;

	}
```

```
	public void setCountry(String country) {
		this.country = country;
	}

	public String getPostcode() {
		return postcode;

	}
```

Item.java

```
package com.ingres.seamexample.entity;

import java.io.Serializable;
import java.sql.Timestamp;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Version;

import org.hibernate.validator.Length;
import org.hibernate.validator.NotEmpty;

/**
 * Represents a product offered for sale.
 */
@Entity
public class Item implements Serializable {

	private static final long serialVersionUID = 1L;

	@Id
	@GeneratedValue
	private Integer code;


	@Column(name = "manufacturer_code", length = 20)
	private String manufacturerCode;

	@Column(length = 50)
	@NotEmpty
	@Length(max = 50)
	private String description;

	@Version
	private Timestamp version;

	public String getManufacturerCode() {
		return manufacturerCode;

	}

	public void setManufacturerCode(String manufacturerCode) {
		this.manufacturerCode = manufacturerCode;
	}

	public String getDescription() {

		return description;
	}
```

```
	public void setDescription(String description) {
		this.description = description;
	}

	public Integer getCode() {
		return code;
	}

	protected Timestamp getVersion() {
		return version;
	}

	protected void setVersion(Timestamp version) {
		this.version = version;
	}

	protected void setCode(Integer code) {
		this.code = code;
	}
}
```

Order.java

```
package com.ingres.seamexample.entity;

import java.io.Serializable;
import java.sql.Timestamp;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Date;

import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Embedded;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;
import javax.persistence.Version;

import org.hibernate.validator.NotNull;

/**
 * Represents a customer order.
 */
@Entity
public class Order implements Serializable {

	private static final long serialVersionUID = 1L;

	@Id
	@GeneratedValue
	private Integer reference;

	@Column(name="created_date")
	@Temporal(TemporalType.DATE)
	@NotNull
	private Date createdDate;

	@Embedded
	@NotNull
	private Address deliveryAddress;
```

```
	@OneToMany(mappedBy="order", cascade = CascadeType.REMOVE)
	private Collection<OrderLine> orderLines;

	@Version
	private Timestamp version;

	public Order() {
		this.setDeliveryAddress(new Address());
		this.setOrderLines(new ArrayList<OrderLine>());
	}

	public Date getCreatedDate() {
		return createdDate;
	}

	public void setCreatedDate(Date createdDate) {
		this.createdDate = createdDate;
	}

	public Address getDeliveryAddress() {
		return deliveryAddress;
	}

	public void setDeliveryAddress(Address deliveryAddress) {
		this.deliveryAddress = deliveryAddress;
	}

	public Integer getReference() {
		return reference;
	}

	public Collection<OrderLine> getOrderLines() {
		return orderLines;
	}

	protected Timestamp getVersion() {
		return version;
	}

	protected void setVersion(Timestamp version) {
		this.version = version;
	}

	protected void setReference(Integer reference) {
		this.reference = reference;
	}

	protected void setOrderLines(Collection<OrderLine> orderLines) {
		this.orderLines = orderLines;
	}
}
```

OrderLineId.java

```
package com.ingres.seamexample.entity;

import java.io.Serializable;

import javax.persistence.Column;
import javax.persistence.Embeddable;

/**
 * OrderLine Identifier. Composite identifier joining an item to an order.
 */
@Embeddable
public class OrderLineId implements Serializable {

	private static final long serialVersionUID = 1L;

	@Column(name = "order_reference", nullable = false)
	private Integer orderReference;
	@Column(name = "item_code", nullable = false)
	private Integer itemCode;

	public OrderLineId() {

	}

	public OrderLineId(Integer orderReference, Integer itemCode) {
		this.setOrderReference(orderReference);
		this.setItemCode(itemCode);
	}

	public Integer getOrderReference() {
		return orderReference;
	}

	public void setOrderReference(Integer orderReference) {
		this.orderReference = orderReference;
	}

	public Integer getItemCode() {
		return itemCode;
	}

	public void setItemCode(Integer itemCode) {

		this.itemCode = itemCode;
	}

	@Override
	public boolean equals(Object obj) {

		if (this == obj) {
			return true;
		}

		if (obj == null) {
			return false;
		}

		if (!(obj instanceof OrderLineId)) {
			return false;
		}


		OrderLineId other = (OrderLineId)obj;

		return (this.getOrderReference() == other.getOrderReference()) &&
			(this.getItemCode() == other.getItemCode());
```

```
   }

   @Override
   public int hashCode() {

       int code = 17;
       code *= 43;
       code += this.orderReference == null ? 0 : this.orderReference
       hashCode();
       code += this.itemCode == null ? 0 : this.itemCode.hashCode();
       return code;
       }
}
```

OrderLine.java

```
package com.ingres.seamexample.entity;

import java.io.Serializable;
import java.sql.Timestamp;

import javax.persistence.AttributeOverride;
import javax.persistence.AttributeOverrides;
import javax.persistence.Column;
import javax.persistence.EmbeddedId;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.Table;
import javax.persistence.Version;

import org.hibernate.validator.Min;
import org.hibernate.validator.NotNull;

/**
 * Represents an element of an order for a specific item. Detailing the
quantity required.
 */
@Entity
@Table(name = "order_line")
public class OrderLine implements Serializable {

       private static final long serialVersionUID = 1L;

       @EmbeddedId
       @AttributeOverrides( {
              @AttributeOverride(name = "orderReference", column = @Column(name
= "order_reference", nullable = false)),
              @AttributeOverride(name = "itemCode", column = @Column(name =
"item_code", nullable = false))
       })
       @NotNull
       private OrderLineId id;

       @ManyToOne(fetch=FetchType.LAZY)
       @JoinColumn(name = "item_code", nullable = false, insertable = false,
updatable = false)
       @NotNull

       private Item item;
```

```
	@Column(nullable = false)
	@Min(1)
	private int quantity;

	@ManyToOne(fetch=FetchType.LAZY)
	@JoinColumn(name = "order_reference", nullable = false, insertable =
false, updatable = false)
	private Order order;

	@Version
	private Timestamp version;

	public OrderLine() {
		this.id = new OrderLineId();
	}

	public OrderLine(Order order, Item item, int quantity) {

		this();

		if (order == null || item == null) {
			throw new IllegalArgumentException("Null value not
allowed for mandatory parameters order/item");
		}

		this.setOrder(order);
		this.id.setOrderReference(order.getReference());
		this.setItem(item);
		this.id.setItemCode(item.getCode());
		this.setQuantity(quantity);
	}

	public Item getItem() {
		return item;
	}

	public void setItem(Item item) {
		this.item = item;
	}

	public int getQuantity() {
		return quantity;
	}

	public void setQuantity(int quantity) {
		this.quantity = quantity;
	}

	public Order getOrder() {
		return order;
	}

	public void setOrder(Order order) {
		this.order = order;
	}

	public OrderLineId getId() {
		return id;
	}

	protected Timestamp getVersion() {
		return version;
	}

	protected void setVersion(Timestamp version) {
		this.version = version;
	}

	protected void setId(OrderLineId id) {
		this.id = id;
	}

}
```

APPENDIX B CONFIGURING ECLIPSE

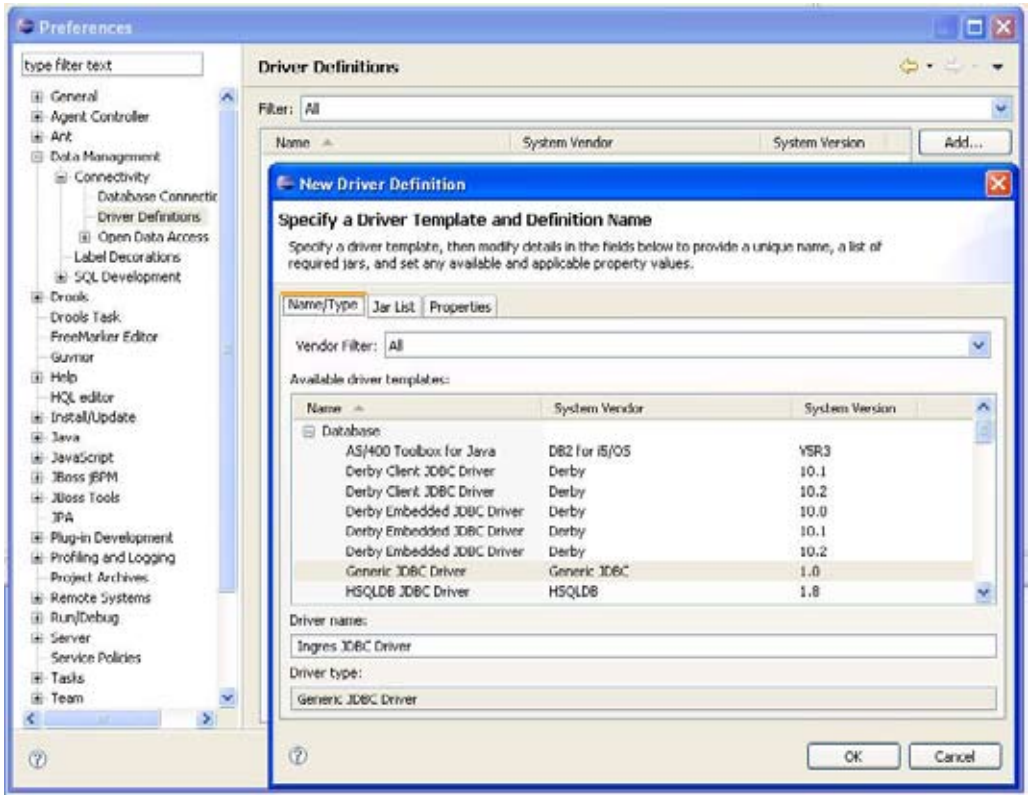
- Installing JBoss Tools

The following Wiki page details how to install JBoss Tools into an Eclipse installation:
http://jboss.org/community/docs/DOC-10044

- Registering the Ingres JDBC Driver

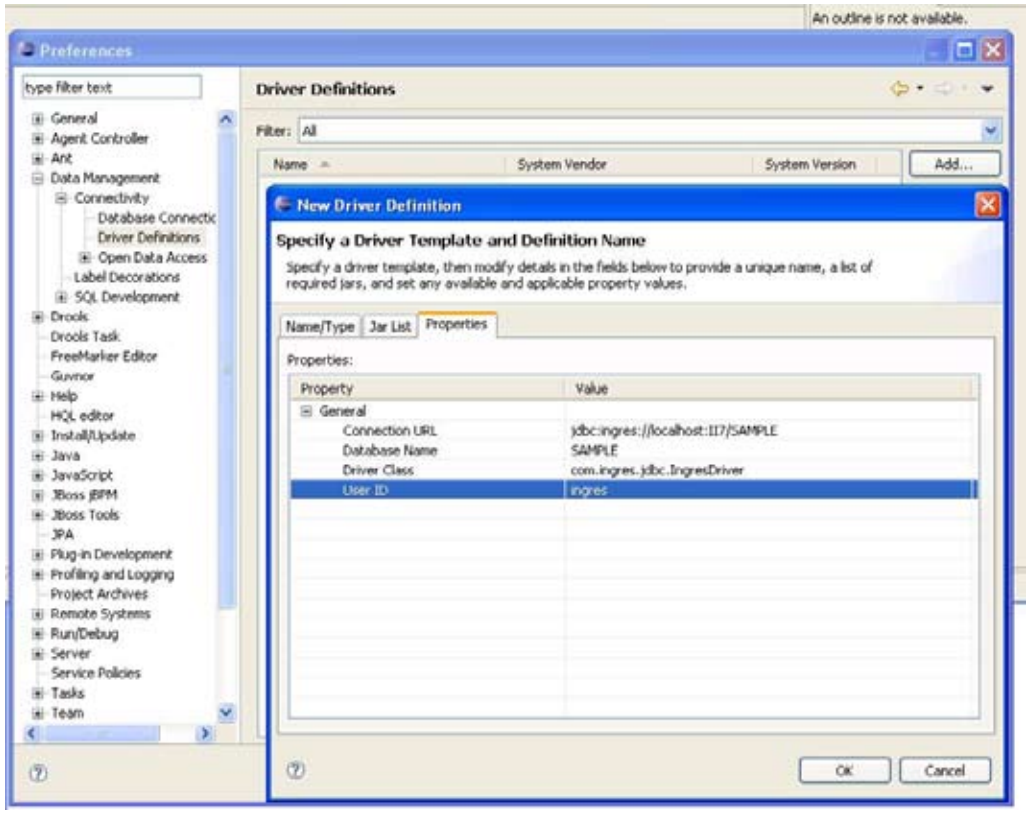
To add a new JDBC driver definition:

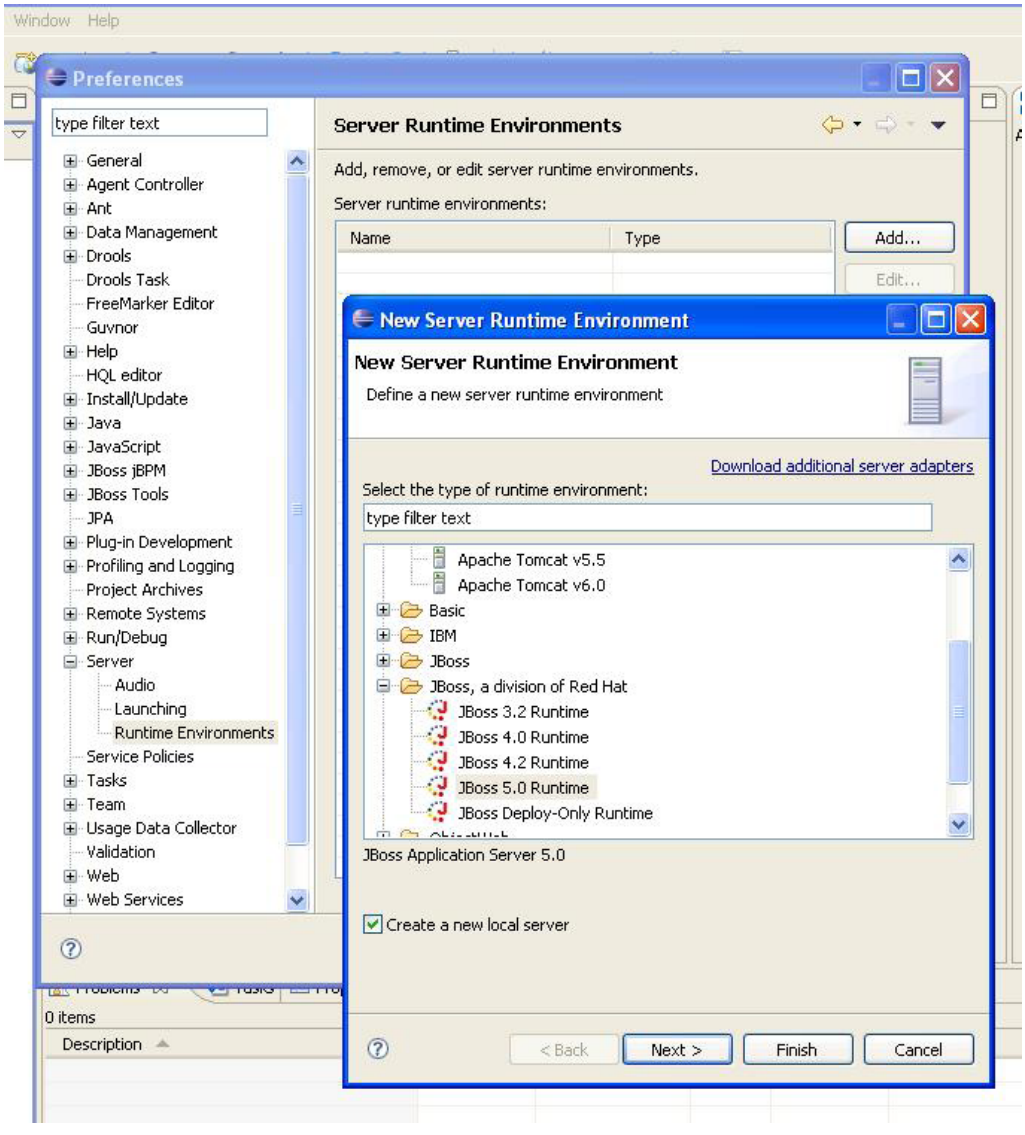
- Open the Eclipse Preferences dialog, accessed via the Window -> Preferences option.
- Navigate to the Data Management -> Driver Definitions option.
- Click the Add button to add a new driver definition.
- Select 'Generic JDBC Driver' from the driver templates list. Change the driver name to something meaningful such as Ingres JDBC Driver.



- Click in the Jar List tab and add the Ingres JDBC driver Jar file (iijdbc.jar). The default location for the Jar file is in the ingres/lib directory of the Ingres installation.
- Click on the properties tab and complete the template details.
  Connection URL: jdbc:ingres://localhost:II7
  Driver Class: com.ingres.jdbc.IngresDriver

  Note: II7 denotes the TCP/IP listen port of the Ingres Data Access Server. II7 is the value used for a default installation, where II denotes the Ingres installation identifier. The listen port value can be verified using the 'Ingres Configuration Manager' utility or the 'Configure-By-Forms' (cbf) command line utility. The listen port information is found in the Data Access Servers node.

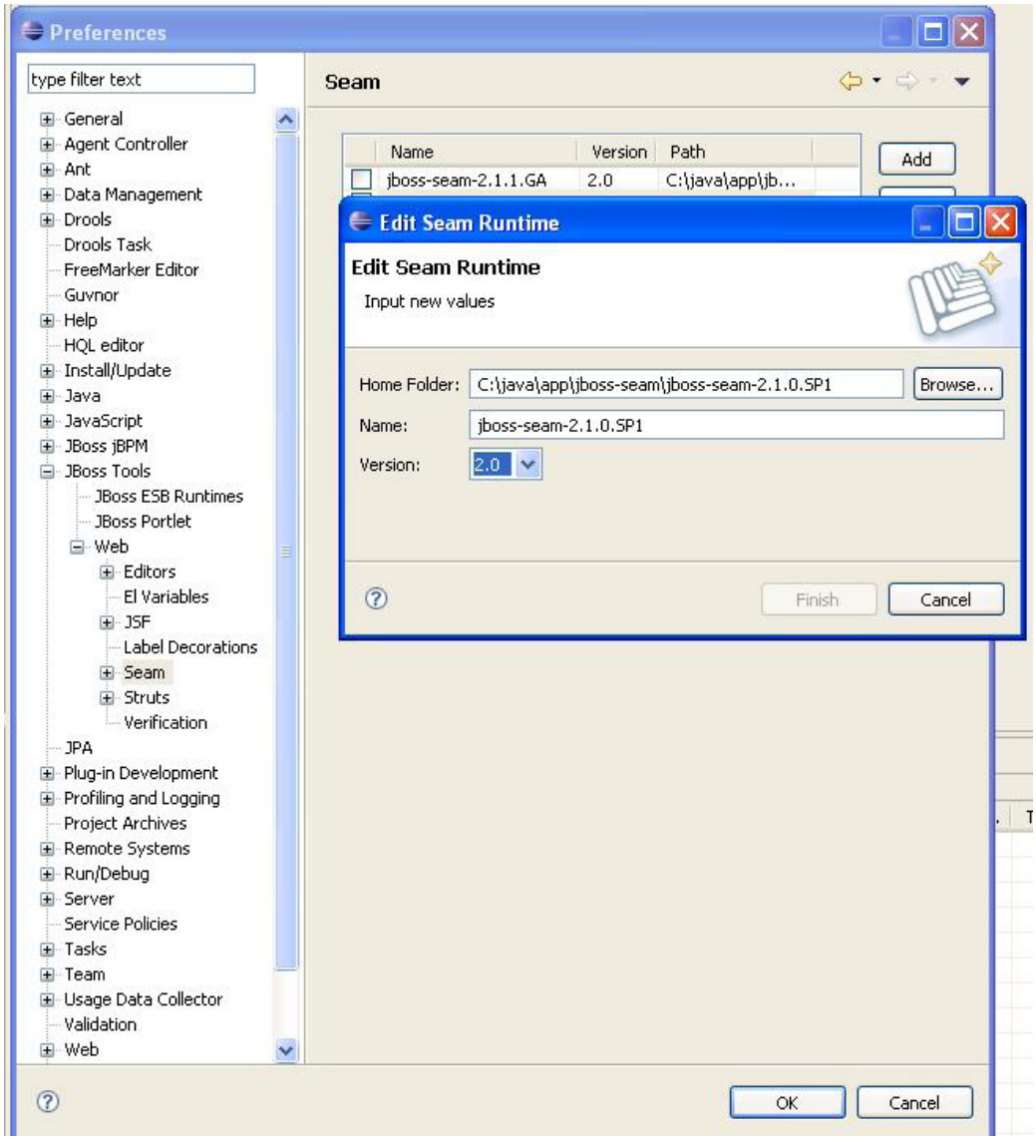




- Registering the JBoss Server

  To register a JBoss server instance with Eclipse:

  - Open the Eclipse Preferences dialog, accessed via the Window->Preferences option.
  - Navigate to the Server -> Runtime Environments option.
  - Add a new JBoss server runtime, by clicking the Add button and selecting the JBoss 5.0 Runtime option available under the JBoss, a division of Red Hat branch. Be sure to check the 'Create a new local server' option.
  - Complete the remaining steps of the wizard by clicking Next and editing the wizard options as required. The 'default' option for the server configuration is fine.



- Registering a Seam installation

To register a Seam installation with Eclipse:

- Open the Eclipse Preferences dialog, accessed via the Window->Preferences option.
- Navigate to the JBoss Tools -> Web -> Seam option.
- Click the Add button to add a new Seam instance.
- Complete the details as required.

APPENDIX C RESOURCES

Links are provided to sites that provide for further information about downloading and using the technologies and tools used in this tutorial.

Ingres: http://www.ingres.com
JBoss AS, JBoss Seam Framework, JBoss Tools: http://www.jboss.org
Eclipse Ganymede: http://www.eclipse.org/ganymede
Sun Java Development Kit: http://java.sun.com

NOTES

NOTES

NOTES

About Ingres Corporation

Ingres Corporation is a leading provider of open source database management software and support services. Ingres powers customer success with the flexibility, cost savings, and innovation that are hallmarks of an open source deployment. Ingres supports its customers with a vibrant community and world class support, globally. Based in Redwood City, California, Ingres has major development, sales, and support centers throughout the world, and more than 10,000 customers in the United States and internationally. For more information, visit www.ingres.com.

Ingres Corporation
500 Arguello Street, Suite 200
Redwood City, California 94063
USA
Phone: +1.650.587.5500

Ingres Europe Limited
St. Martin's Place, 51 Bath Road
Slough, SL1 3UF
United Kingdom
Phone: +44 (0) 1753 559500

Ingres Germany GmbH
Ohmstrasse 12
63225 Langen
Germany
Phone: +49 (0) 6103.9881.0

Ingres France
7C Place Du Dôme
Immeuble Elysées La Défense
92056 Paris La Défense Cedex
France
Phone: +33 (0) 1.72.75.74.54

Ingres Australia
Level 8, Suite 1
616 St. Kilda Road
Melbourne, Victoria, 3168
Australia
Phone: +61 3 8530.1700

For more information, contact ingres@ingres.com

INGRES

WP-309A