Building
Rich Internet Applications
using Flex and Java

Building
Rich Internet Applications
using Flex and Java

For the latest on CreApple creations, go to www.creapple.com
The source code for this book is available to readers at www.creapple.com
To report errors, please send a note to creapple@gmail.com
For information on getting permission for reprints and excerpts, contact creapple@gmail.com

ISBN-13: 978-1466359574

ISBN-10: 1466359579

Creative Knowledge Bank _ 001
Creative Application Era: All you get in www.creapple.com

Building Rich Internet Applications using Flex and Java

creApple

In 2002, Rich Internet Applications (RIA) emerged as a revolutionary concept for web development. It was borne out of an idea and need to use Flash, originally created to enable animations in the web, as a presentation layer for web applications. Rich Internet Applications didn't just give a facelift to the user interface, however; they introduced significant improvements to both the user environment and the applications' efficiency.

In introducing Rich Internet Applications, Macromedia used its Flash MX and ColdFusion MX technologies to release the Pet Market Blueprint Sample Application. For many developers, used to HTML page-based applications, it was very much a novel experience.

In the following years, a number of technologies have developed with the RIA concept in mind, including now standardized technologies such as Java and .Net. In this book, we will use the most common RIA architecture, and learn how to rebuild a Java open-source based web application as a Rich Internet Application by looking at a sample application. This will involve taking Mybatis' Jpetstore6 Sample Application as a basis – one of the best models for Java applications. Throughout the course of this book, we will then use Flex and other technologies to turn this HTML page-based Web Application into CrePetstore, a RIA Blueprint Application. This book provides a pre-built sample of CrePetstore, which is designed with cross-platform support in mind – not only across Windows, Mac OS and Linux, but mobile devices.

While the sample application itself is already a handy reference, the detailed explanations in this book provide a more in-depth understanding of both the background and implementation of RIAs. I hope that when you are finished with this book, you will find yourself an expert in building Rich Internet Applications.

What You Can Get

In this book, we discuss both Java-based web applications and Flex-based RIAs. For the Java developer, this means you can learn how to take your existing familiarity with Java and integrate Flex and RIAs into your projects. For developers

who are more familiar with Flash or Flex, this book will guide you through the core techniques demanded by server-side applications, such as Spring, POJO (Plain Old Java Objects) and ORM (Object-relational mapping).

This book is divided into ten chapters, which are grouped according to three concepts – Model, Control and View.

Model

- **Getting Started**: Learn more about how Rich Internet Applications came into being, and the developer environment necessary for the CrePetstore project.
- **Spring Fundamentals**: Learn about Spring, one of the signature frameworks for Java web applications, and its principles, structure and implementation, together with source code.
- **Data Service**: Learn about the principles and implementation of HTTPService, Web Service and RemoteObject - the techniques used to service data from server-side to the Flex client.

Control

- **Rich Internet Applications**: Learn about the basic principles behind RIAs, and how Flex and Spring applications can be integrated to deliver them.
- **Cairngorm Framework**: Learn about Cairngorm, a Model-View-Control (MVC) framework used to structure Flex applications. More than just a framework, however, Cairngorm is generally considered a methodology and best practice that can help achieve optimal arrangements of Flex code and components.
- **Data Communication**: Analogous to the 'Data Service' chapter, this chapter examines how the data services are implemented on the Flex client side through source code examples.

View

- **Flex Fundamentals**: Looking at the sample application, learn the basic techniques behind building Flex application, using either Flex SDK or Flash Builder.
- **Event and Data**: Learn about event-based architecture and data binding, two ways to create more efficient and organized Flex applications.
- **User Interface Enhancement**: How to use style, skins, chart, navigation, transition and effect and more to improve Flex user interfaces.
- **Advanced Topics**: More advanced techniques such as localization for multiple languages and mobile platform support using Adobe AIR.

Using Sample Application

As mentioned above, this book explains various concepts and techniques by referring to examples from CrePetstore, a sample application. The sample code provided in this book comes from this application, and we recommend that you download it directly for reference at the CreApple site (http://www.creapple.com). The CrePetstore application is not simply a collection of examples, but a fully functioning application, and will serve as a useful reference in many ways. The source code for CrePetstore is open for readers to view and use. However, you should note that if the code is reused in another application, the creator's permission must be given separately, and that no responsibility can be taken for the code once reused.

Acknowledgements

Writing this book meant a series of seemingly never-ending challenges. The original intention seemed worthwhile – create a sample application, and explain it in a way that is easy to understand – but further complicated the process. After many nights of asking myself whether I will get there in the end, the journey is finally at an end. How can I express in words the joy and thanks that I feel at this moment?

To my parents, who have always remained my faithful supporters, to my loving wife and her belief and devotion, and to my son and daughter, who give me courage in hope – I want to speak words of apology before thanks. I ask for forgiveness for my selfish desire to write, which took my time and energy away from family, and as my thanks, promise a happier future.

I also cannot forget the power of knowledge, shared by countless people over the internet. Without them, this book and the sample application will not have been possible. Although I will not be able to see or thank each of them in person, I want to extend my thanks to them as well. I am delighted to be able to contribute what little I can to this pool of knowledge, as they have, through this book.

Finally, I'd like to say thank you to myself as well, for enduring this journey to fulfil a small dream, and to keep a promise to myself.

A Bird that flies higher can see farther

JONATHAN LIVINGSTON SEAGULL

A creative People can beautify the world

JONATHAN SUH

01
Getting Started

We are now about to begin our adventure with Flex. Using a sample application of an online shopping mall, the Creative Pet Store, we will learn how Flex-based Rich Internet Applications work, and how to implement them in practice. By the end of this book, you will have the necessary skills and knowledge to create your own Rich Internet Applications with confidence. This book looks to provide real examples from the 'CrePetstore' sample application in favour of technical minutiae, showing you how the Flex source code actually works. You will be able to see how your own Flex applications should look and work, bringing in examples from the sample applications for your own use.

The 'CrePetstore' sample application is an example of a fairly common solution in Rich Internet Applications (RIA): it is based on an existing server-side application called 'JPetstore', which is page-based, then only uses the RIA system for view layers. This approach has the advantage of a rich yet stable legacy system, allowing the various features of RIAs to be implemented even when the system as a whole cannot or should not be replaced. Such solutions often involve migration of model and control layers, and can be achieved with comparatively low resources and time. The 'CrePetstore' sample, therefore, is designed to provide you with the examples that match real-life solutions as closely as possible.

In this chapter, we will prepare for our journey ahead with a brief look at the background to RIAs, the 'CrePetstore' sample application, and what you need to prepare for when beginning a web application project. Let's take our first steps towards web development with Flex!

Background

In February 14, 1946, the University of Pennsylvania's laboratories witnessed the arrival of humanity's first electronic computer, the ENIAC (Electronic Numerical Integrator and Computer). It was the starting point to the information revolution that has since transformed the world. The Time magazine lauded had described it as able to calculate in two hours what a million mathematicians would need a year to complete. Developed with military applications in mind, ENIAC proved its worth by calculating missile trajectories in thirty seconds – a task that had taken between 7 and 20 hours previously. It was a revolutionary invention in every sense of the word.

Since then, over sixty years of in information technologies have accelerated development beyond comparison to their predecessor, the industrial revolution. IT has now established itself as an absolute necessity in almost any business operation.

How do Rich Internet Applications (RIA) fit into all this? We can answer that question by looking at the history of IT in terms of client-server relationships.

The Evolution of Applications

The earliest information systems had no real sense of the 'client'; all functions were handled exclusively through the central mainframe. Tasks were rarely delegated, and the sole 'client', the dummy terminal, only performed the basic role of receiving user data and displaying the data processed centrally.

The co-founder of Intel, Gordon Moore, famously commented that computer chip capacity would double every two years. Moore's Law proved largely correct, and the corresponding reduction in the cost and size of central processors led to the development of more robust client-based systems. As demand for client applications such as word processing and spreadsheets increased, the 'dummy terminal' was replaced by micro-computers. However, as data handling was decentralized, maintaining data integrity became more and more problematic. The rapid growth of the number of client systems also raised concerns over cost of investment.

The 1990's saw another fundamental shift in the landscape of business applications, with the arrival of the internet and the corresponding development of network capabilities. A page-based architecture, consisting of a series of HTML or other 'pages', was introduced through web browsers. This once again returned the control of the data to a centralized server, facilitating easier communication between server-side and client-side. However, page-based web applications often could not compete with offline client applications in terms of complexity and

functionality; their simpler and lighter user interfaces could not provide the same kind of applications that users had come to expect from their client applications. The page-based architecture also increased dependence on the network itself, as it would need to contact the server for every new page and request. These concerns have led to the development of RIAs today.

The term RIA originates from Macromedia (now part of Adobe), around 2001, and was also largely synonymous with X internet, rich web client or rich web application. Though similar attempts had been made previously, Macromedia was able to popularize RIAs through their flagship services such as Flash, Dreamweaver, and now Adobe with its Flex series.

RIAs are designed to combine the richness of client applications' user interfaces and the efficiency of internet-based applications. Its strengths can be summarized as below:

- **An Improved User Interface**: One of RIAs' basic aims was to provide functionalities that older applications – for instance, those based on plain HTML – could not provide. A common example is the drag-and-drop functionality. This can raise productivity and also meet the expectations of the end user by providing functionalities found in client applications.
- **Better Performance**: As discussed above, the weakness of many page-based web applications is that they expend a great deal of time and resource communicating to the server every time the application is refreshed or new information is retrieved. RIAs are designed to minimize server-client interaction, and are based on a 'no-page-refresh model'. Its Views interface provides a more stable and efficient way to manage client sessions and minimize network load.
- **More Efficient Development**: Adobe now provides open-source SDKs for Flex, and also supports RIA development through a variety of components and frameworks. They have also made available Flash Builder, a powerful integrated development environment that can also be used as an eclipse plug-in. All of these tools contribute to a more efficient development cycle.

In that case, let's examine our sample application and see what it is about RIA architectures that provide improved efficiency. Let us say that a user is navigating the 'CrePetstore' application, and searching for a pet animal to purchase. He/she browses through the categories of animals, selects 'Dog', and in turn, the specific product 'Bulldog'. He/she then selects 'female', and orders the chosen pet. Simply put, the idea here is that there is a client-server communication through the network, where the client side involves the user and the 'View' (the website's UI), and the server side includes the Control and Model components.

What if 'CrePetstore' were a HTML page-based application? The user would begin from a page that displays the category lists. Though the category data could be hard-coded into the page, in most cases, it would need to be retrieved from the server. When the user chooses the category 'Dog', the page would then need to request the product list from the server, and produce a new page, while the user waits. This process would repeat itself until the end of the transaction, contributing to server and network load with each click. The user also needs to wait a certain time, depending on his or her internet connection, each time for the page to be loaded. Finally, the page-based application is then required to manage its sessions carefully, ensuring consistency between its individual pages.

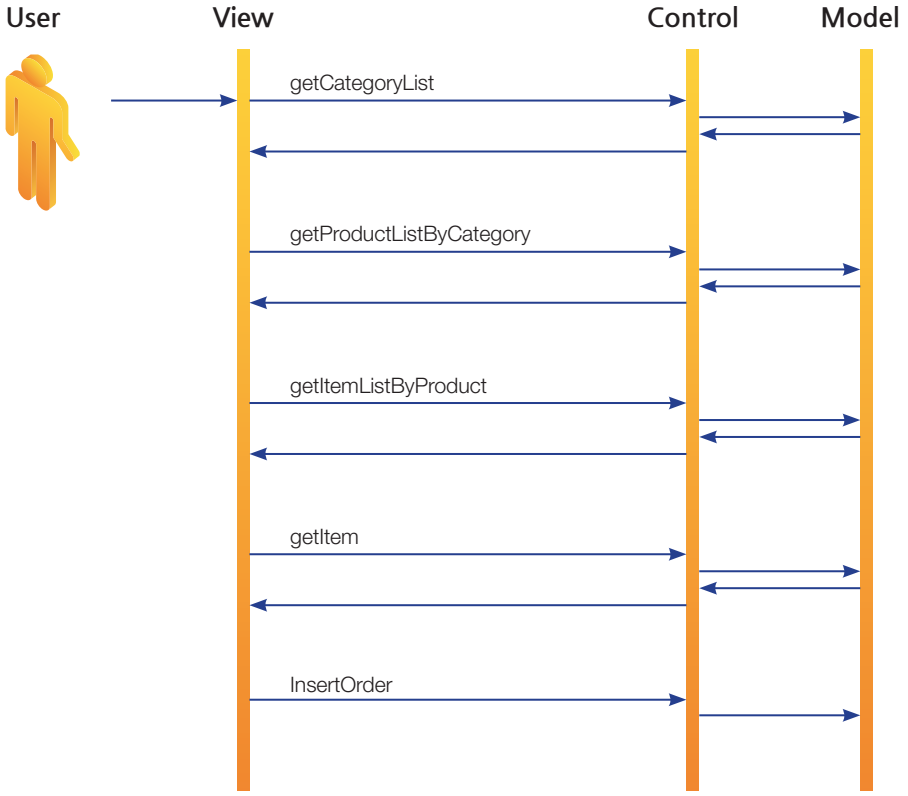

Figure 1.1 Interaction in Page–based Applications

The same transaction would work rather differently with an RIA architecture. The client-side View would operate with a great deal of independence, handling more of the data retrieval processes. When the user first accesses the application, the View is downloaded onto the machine, and brings with it commonly accessed

data such as category, product and item lists. Using this pool of data, the client-side component is able to then provide the same data without retrieving it from server-side, allowing the user to navigate the application without waiting for response from server. Furthermore, because the application is not broken up into individual pages, but integrated into a single instance with various states and actions, RIAs can bypass the complex session management page-based architecture requires.

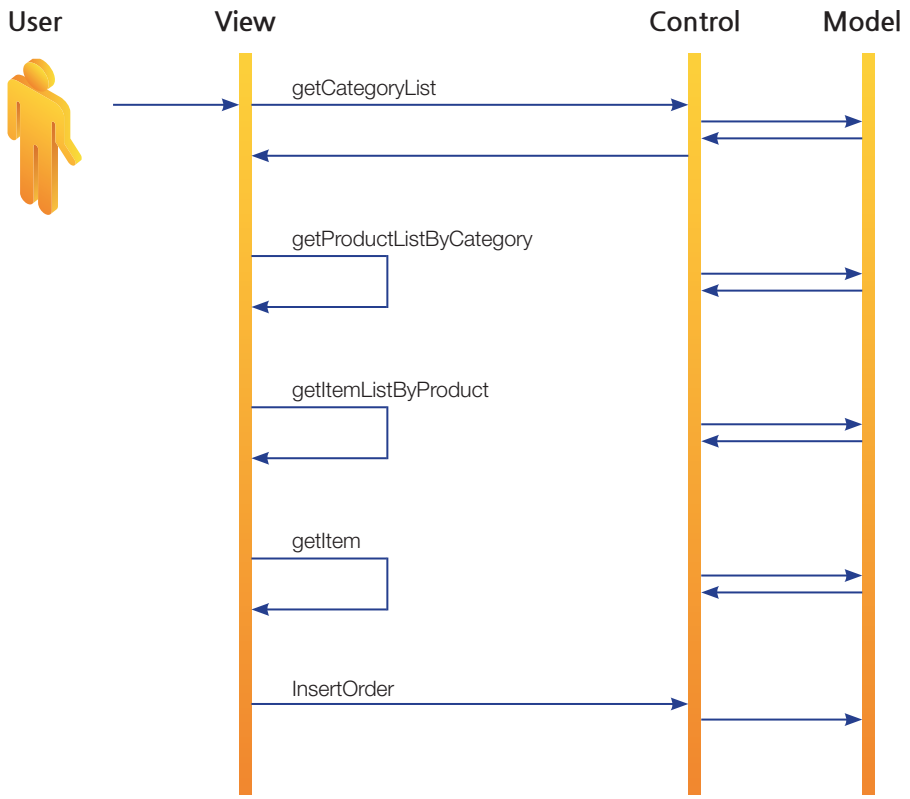

Figure 1.2 Interaction in Rich Internet Applications

The user experience also changes significantly with the use of RIAs. In many page-based online shopping sites, the implementation of a 'shopping cart' involves juggling various sessions and states, which can lead to inconvenience on the part of the user. For instance, a user might select an item to purchase, only to have the page reload to place it in the cart. The user may then need to manually return to the product search screen to continue shopping, and then return to the cart page periodically to check his or her selected items. All of these navigations involve requests to the server and detract from the user experience.

Figure 1.3 Shopping Cart UI in Page–Based Applications

Rich Internet Applications address these issues by offering a single View that can handle multiple states such as product browsing and shopping cart screens. The support for functionalities such as drag-and-drop also enhances usability. As the 'CrePetstore' example shows below, the user is able to examine all the relevant information at once, and make changes to the shopping cart dynamically without waiting for page loads. RIAs also help make the inclusion of audiovisual elements easier than before.

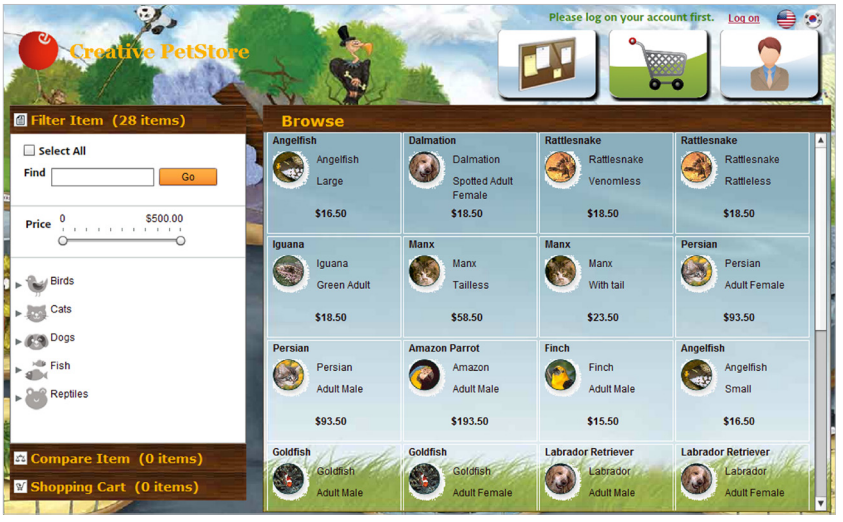

Figure 1.4 Shopping Cart UI in Rich Internet Applications

Pet Store Blueprint Application

A blueprint application is a prototype that can demonstrate the capabilities of a specific technology, and act as reference material for developers using that technology. One such blueprint application is 'JPetstore', or Java Pet Store. In 2001, Sun Microsystems created J2EE (Java 2 platform, Enterprise Edition), expanding on their highly successful Java platform. Centered on Enterprise JavaBeans (EJB) and Java Server Page (JSP), J2EE offered a well structured and stable platform, but was unfamiliar to many developers. Sun Microsystems, therefore, provided 'JPetstore' as a blueprint application for J2EE – currently found in (http://java.sun.com/developer/releases/petstore/).

Figure 1.5 J2EE–based Java Pet Store

Sun Microsystems used JPetstore to showcase many of J2EE's functionalities. In particular, the application was designed to address concerns that an EJB-based, enterprise-level service could be useful for small to medium businesses, by using the example of a pet store.

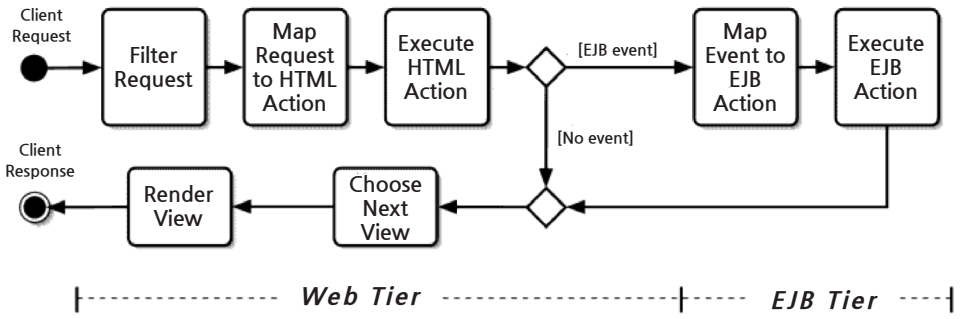

Figure 1.6 Java Pet Store's Framework Service Cycle

In November of the same year, another pet store-themed blueprint application entered the scene. Microsoft, whose .NET technology was in tight competition with Java at the time, introduced its own '.Net Pet Shop' application. With this, Microsoft aimed to not only provide reference material to .NET developers, but show to the IT world that .NET could beat Java at its own game. In fact, Microsoft advertised its own application by claiming that its codebase is 1/7 the size of JPetstore, and ten times faster.

Figure 1.7 .NET–based Pet Shop

Microsoft claimed that the two blueprint applications demonstrated .NET's superiority over its rival. However, this view was far from universally accepted. Some pointed out that Sun's JPetstore was based on EJB, designed for enterprise-level applications, and was therefore needlessly burdened by many features and functionalities a shopping mall application would never use. While Sun's intention was to prove that EJB, for all its complexities, was an approachable solution for developers, this had allowed Microsoft to give the impression that .NET was a faster and more efficient solution for web development.

Worried that Microsoft's claims would harm Java's reputation, open-source Java developers responded by releasing a blueprint application of their own. It would use open source technologies that divested many of EJB's bells and whistles, but showcased the core functionalities more efficiently. The interesting point here is that these open-source pet store applications were often more similar to .Net Pet Shop than Java Pet Store.

In any case, these open-source pet store applications have emerged as very useful references for small to medium businesses and their applications. The best known of these applications include the Spring JPetstore (http://www.springsource.org/), which was supported up to Spring Framework 2.x, and MyBatis' JPetstore (http://code.google.com/p/mybatis/).

Figure 1.8 Spring–based JPetStore

Finally, one fresh approach to web applications stands out from this battle between Java and .NET around the turn of the century. In June 2002, Macromedia released a blueprint pet store application of its own, based on its own Flash MX and ColdFusion MX lineup. Their Pet Market application showcased an RIA approach web applications, using flash to create a client application that could then interface with server-side.

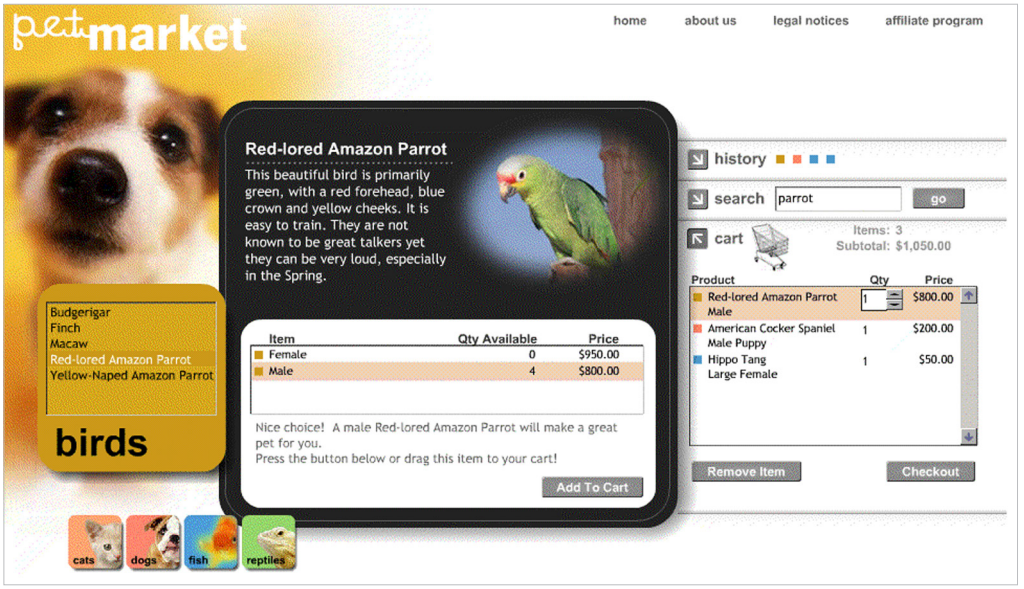

Figure 1.9 Flash–based Pet Market

Macromedia's Pet Market utilized Flash's unique capabilities to provide advanced UI functionalities such as drag-and-drop, and used a single View model rather than HTML-based pages, as the Java and .NET applications did. It was presented not as a replacement for the old HTML-based model, but a way to expand on it. However, the Pet Market also demonstrated the limitations of the time; creating the entire UI through Flash proved difficult and sometimes overly complex.

Our Creative Pet Store application is, then, a nod to these pet stores of old – a blueprint for Flex-based RIAs that we will use as reference throughout this book. It brings together the open source Java applications' flexibility and compatibility with the Flash-based Pet Market application's rich UI.

Figure 1.10 Flex–based Creative Pet Store

The Flex-based 'CrePetstore' inherits the open-source Java Pet Store's server-side code as a legacy system, integrating an RIA user interface with it. This presents a good opportunity to compare the differences between page-based applications and RIAs, and also reference material for building Flex RIAs of your own.

Development Environment

Now, we are ready to configure our own Flex-based projects. In this section, we will examine the development environment for the Creative Pet Store sample application. This is an important preparatory step, and should be considered as a key milestone before any coding begins. For Flex RIAs, we need a certain set of tools in order to write, test and deploy the application.

This book recommends certain general principles regarding Flex RIA development. The first is that the application should be based on open-source technologies, when possible, rather than specific commercial products. The second is that even when using tools such as Flash Builder and Eclipse, the application should avoid using functions specific to such IDE (Integrated development environment), so that the code continues to be compatible with other tools if such changes are required during the development cycle. The 'CrePetstore'

sample application will also demonstrate how to configure the development environment to minimize dependency on specific tools. Finally, we recommend that the project as a whole is always considered as a single unit. The book often separates the Flex from Java for convenience, but the sample application itself is managed as a single unit.

These principles stem from my own experience with Flex and web development in general, and the understanding that many developers use the same tool in different ways. Reducing dependency on specific tools or specific solutions works to create a more flexible development environment.

The success or failure of any project, therefore, depends on the choice of tools. As discussed previously, our sample application involves a mix of a Java server-side application and a Flex client-side application. The application as a whole needs to: manage its data (through databases), provide services through the web, and be deployed quickly and easily. 'CrePetstore' uses the following tools to achieve these aims:

- **JDK**: The Java Development Kit. This includes tools such as Java's virtual machine (JVM), the compiler, debugger, and Java applet, and can be found in (http://www.java.com).
- **Apache Ant**: A pure Java build tool that is simpler and easier to use than GNU Make (http://ant.apache.org).
- **Apache Tomcat**: An open Source JSP and Servlet Container provided by the Apache Foundation (http://tomcat.apache.org).
- **Flex SDK**: An open source Flex development framework that includes documentation, source and bug database (http://www.adobe.com/products/flex).
- **MySQL**: First introduced in January 1998, MySQL is an open-source, SQL-based database management system (http://www.mysql.com).
- **Eclipse**: A project aiming to provide a universal toolset for development. Eclipse is an open source IDE mostly used in Java, though the development language itself is independent (http://www.eclipse.org).
- **Flash Builder**: Flash Builder is either used as a stand-alone program or an Eclipse plug-in. It is a powerful tool that aids in Flex application design and coding (http://www.adobe.com/products/flash-builder.html).

Environment Variables

Once all the tools are installed according to the instructions from their providers, we need to adjust their settings. Some of the tools we will use – JDK, Ant, Tomcat, Flex SDK – will need to be configured for environment variables beforehand. This is necessary for the tools to know where to look for required files and programs in the development environment. In Windows, this normally involves finding the configuration file and manually setting the specific file paths. Let's say that we have decided to install all our tools within 'C:\Dev\':

Configuring Environment Variables

```
set ANT_HOME= C:\Dev\Ant
set JAVA_HOME= C:\Dev\JDK
set CATALINA_HOME= C:\Dev\Tomcat
set FLEX_HOME= C:\Dev\Flex
set PATH=%PATH%;%ANT_HOME%\bin;%JAVA_HOME%\bin;
%CATALINA_HOME%\bin; %FLEX_HOME%\bin;
```

In Windows, you can also use the control panel to set environment variables, or simply save them as a batch file. In the former case, we simply set our desired file path as the system variable, which is then applied for all users.

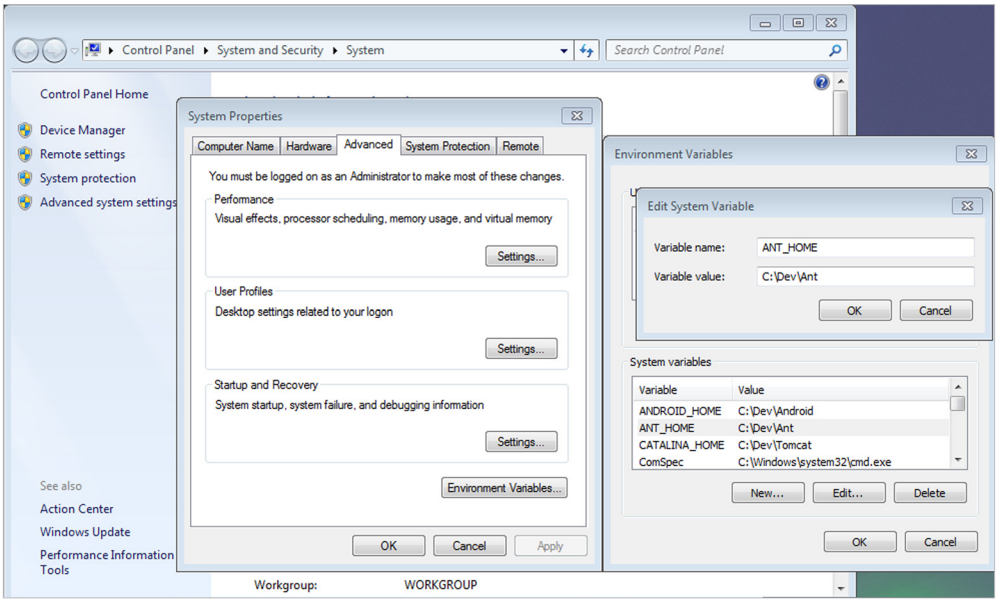

Figure 1.11 Configuring Environment Variables in Windows

The same process also needs to be performed in other, Unix-affiliated operating systems, such as Mac OS X or Linux. In their case, however, we first need to have a basic understanding of two concepts – Kernel and Shell. A Kernel can be understood as a core module that manages the resources (memory, processor, disk) used in various tasks and ensures the smooth operation of the OS. Meanwhile, a Shell is used to regulate and also connect these Kernels.

The first shell – the 'Bourne Shell' – was named after Stephen Bourne, one of the developers at the AT&T Bell Labs. This was used as the reference for all shells that followed. One of these, the Bourne Again Shell ('bash'), added new functions such as command-line editing and improved scripting support. It is now one of the key shells used across all Unix-affiliated OS. While other shells, such as 'csh', 'tcsh' and 'zsh' are also well known, we will mainly use 'bash' for setting our environment variables.

When Unix-affiliated OS such as Mac OS X are loaded, they use shells such as 'bash' to initialize their user environment. These shells can therefore be used to define commonly used aliases and other settings, so that they do not need to be set manually each time the system is booted. We can make use of the following bash files depending on the context:

- **/etc/profile**: This applies to all user accounts that use bash. It includes all the settings related to bash and sh shells, and can only be edited with administrator access.
- **~/.profile**: This applies to a specific user account, and is run when a new shell is created – for instance, when a new terminal window is opened.
- **~/.bash_logout:** This is used when logging out from a shell.

Let us say that your JDK, Ant, Tomcat and other tools have been installed in a Mac OS X system, and in '/Library'. We can set the environment variables as the following:

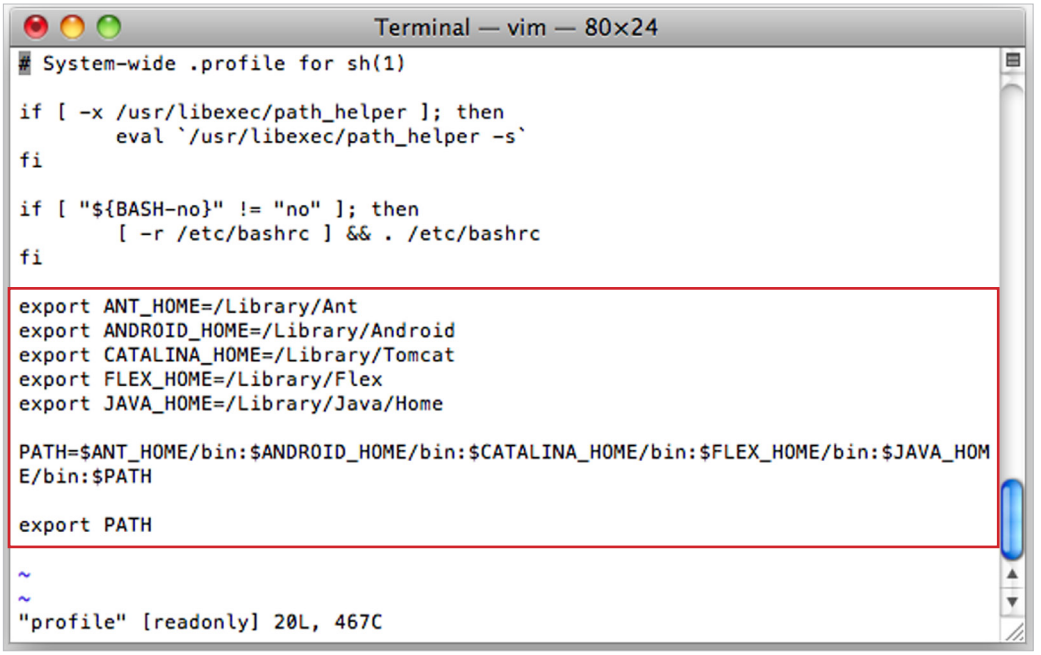

Figure 1.12 Setting Environment Variables in Mac OS X

We will see later that setting these environment variables early on allows us to avoid having to set absolute paths for each operating system or end user environment, but use such relative paths to increase compatibility.

Configuring the Sample Application

Now, it is finally time to move on to the sample application itself. Having installed the necessary tools and set our environment variables, we can install and configure 'CrePetstore'. The installation files can be found in (http://www.creapple.com). CreApple stands for 'Creative Application Era', and is a website for the distribution of applications across areas such as mobile technology and robotics.

Once the download is complete, place the 'CrePetstore.zip' file in an appropriate location, such as 'C:\Dev\workspaces', and extract the archive. Now we can use Eclipse to review the sample application.

Eclipse, as discussed previously, is a well known open-source IDE that is often celebrated for its ability to use various plug-ins to extend its capabilities. Flash Builder is also provided in such a plug-in form, so that it can be integrated to Eclipse. Other features available through plug-ins include version control systems such as Subversion or CVS. As individual developers will make their own choices about how to configure their Eclipse IDE, we will stick to explaining its basic capabilities as a code editor. Using the File > Import menu, open the import pop-up window, then selecting General > Existing Projects into Workspace.

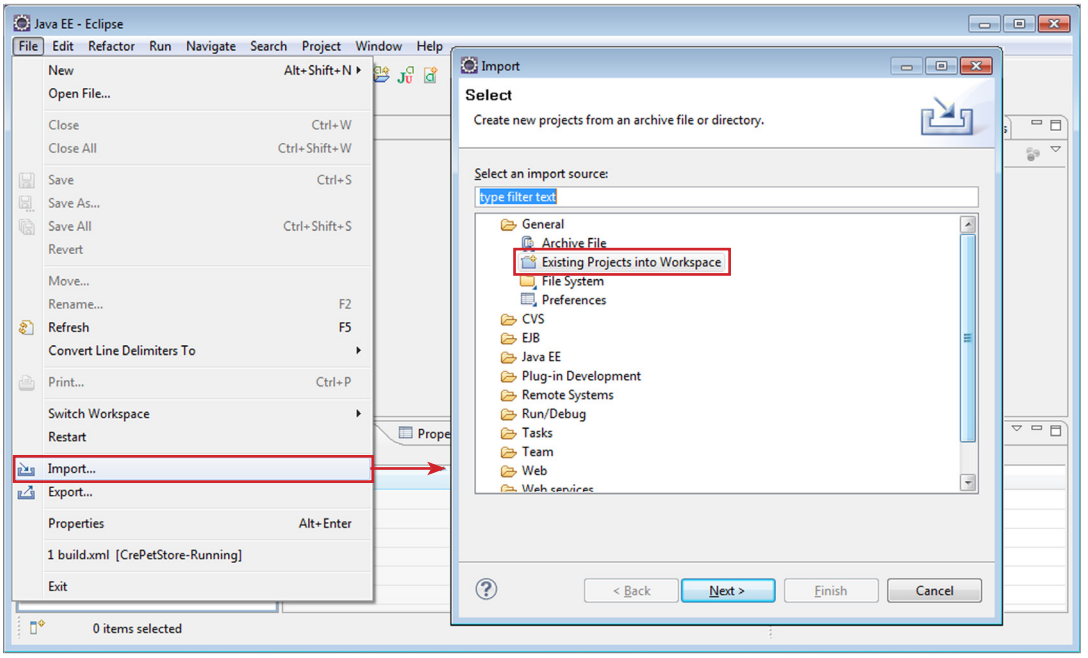

Figure 1.13 Import Existing Projects into Eclipse

Press 'Next' then 'Finish' to import the 'CrePetstore' project. This will allow us to view and edit the application within Eclipse.

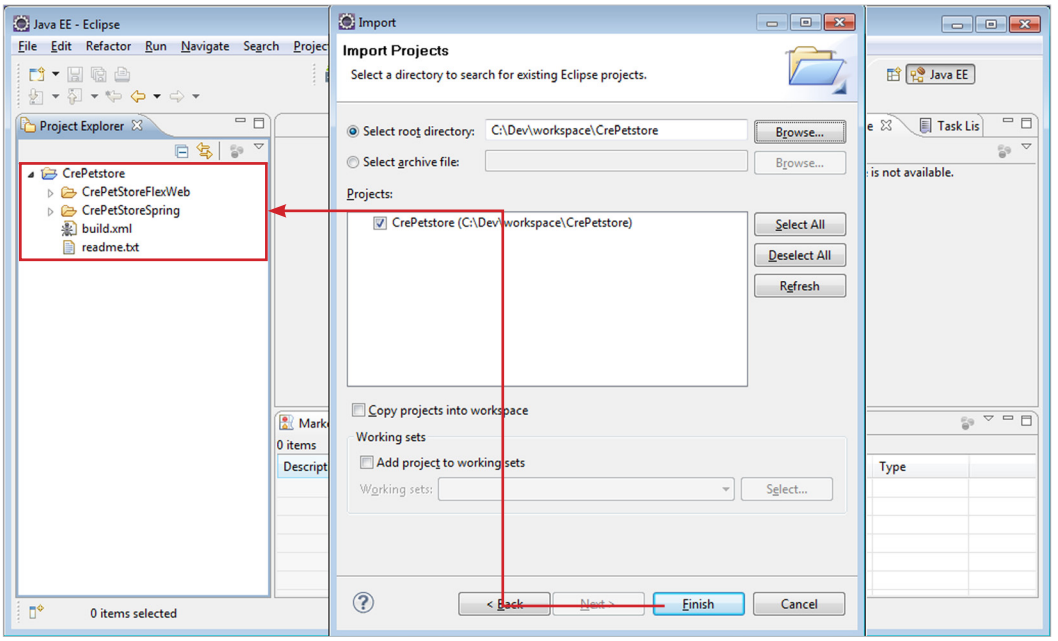

Figure 1.14 Importing the Creative Pet Store Project

Once imported into Eclipse, the project still needs to be deployed. In this example, we will use Apache Ant to do this.

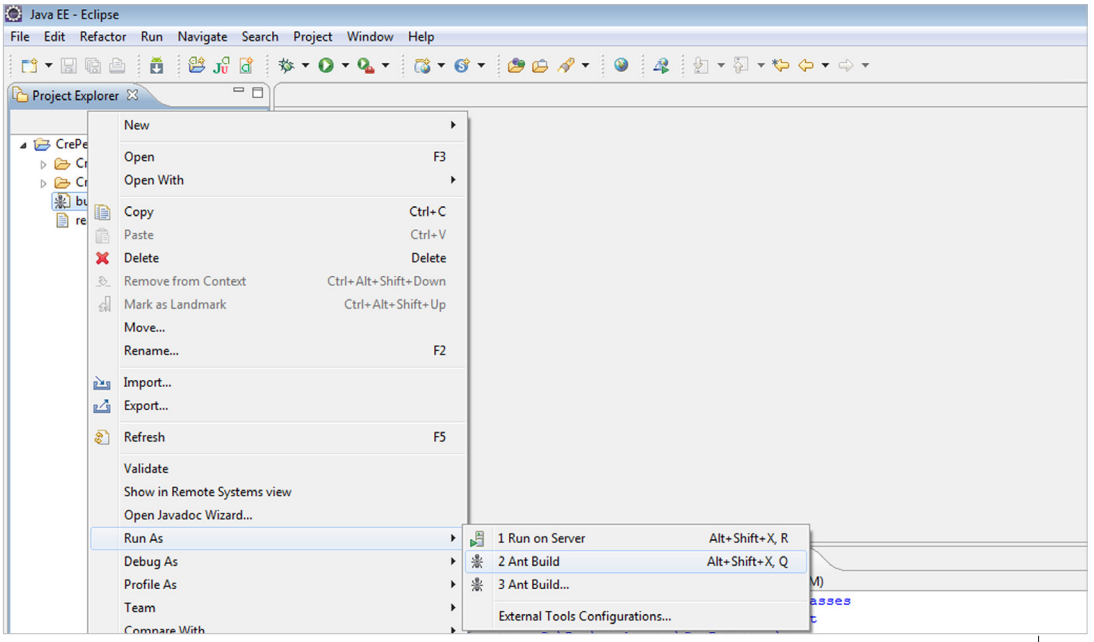

Figure 1.15 Deploying Creative Pet Store Project in Eclipse

We will return to the subject of deployment later on in the book. For now, we only need to know how to use Ant in Eclipse. Right-click on the 'build.xml' file, which is included in the 'CrePetstore' project, then use the Run As > Ant Build command to deploy the project using Ant. This process, of course, requires us to have set the environment variables correctly beforehand.

Eclipse is not always necessary for deployment: it is merely a more convenient and efficient option. We can also use Windows' command window, or Unix OS' terminal, to manually run Ant.

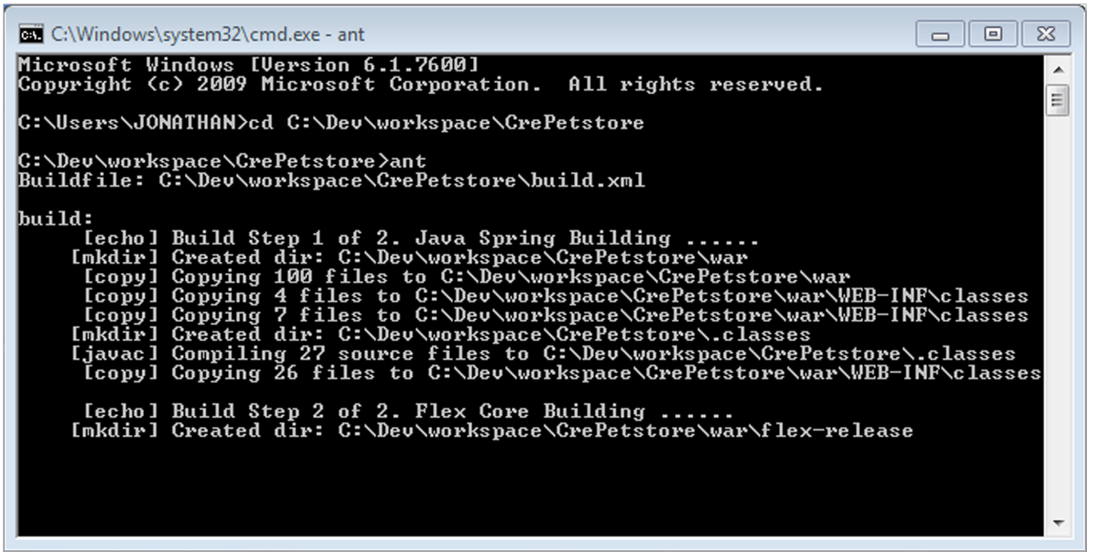

Figure 1.16 Project Deployment using Ant

An alternative method is to import the Flex project through Flash Builder. Flash Builder uses the FXP file format to import and export projects as single files. The 'CrePetstore' sample application download includes such a mobile project file, named 'CrePetstoreMobile.fxp'. In Flash Builder, use File > Import Flash Builder Project, then select the FXP file to import 'CrePetstore'.

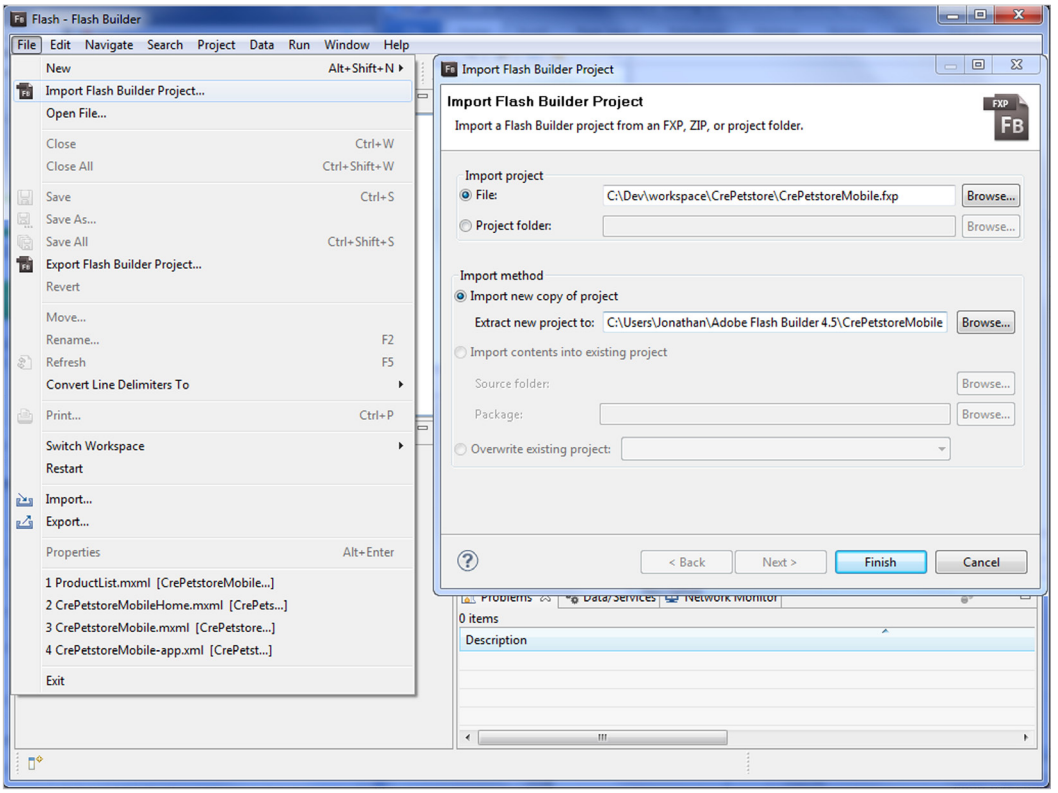

Figure 1.17 Importing Flex Projects into Flash Builder

Application Deployment

In this section, we will learn how to deploy the completed application. While the deployment process can differ across developers and situations, we recommend carrying over many of the same principles that have governed our development environment.

Our CrePetstore sample application demonstrates these principles in practice. Firstly, it does not make use of specific IDE (integrated development environment), preferring simpler and fewer tools whenever possible. Secondly, we do not focus solely on a specific OS or version, but take care to maximise end user compatibility. Finally, we will show how the compile and deploy processes can be standardised and automated for the developer's convenience.

The tool that we will showcase here for deployment is ANT – one of the most used and loved by web developers. In the following section, we will briefly examine

ANT's basic structure, and then learn how the CrePetstore application uses it to extend certain functionalities. ANT will also allow us to bypass the use of IDEs such as Eclipse through the use of batch files.

Introducing ANT

ANT is a Java-based tool that automates build-related tasks such as compiling of code. The purpose of such build tools is to automate certain 'rote' tasks that would otherwise take up a large amount of development time. If you have developed C or C++ based applications in the past, for example, you will be familiar with a powerful build tool known as 'make'. Readers with Java experience may also have used ANT before.

ANT is a TLP (Top Level Project) from the well-known Apache Software Foundation. Developed by James Duncan Davidson, it was originally subordinate to another TLP, Jakarta, before being upgraded to a TLP of its own in November 2002. ANT, like many of the other tools discussed in this book, is open-source, and can be acquired online at (http://ant.apache.org/). ANT shares many common functions with build tools such as make, gnumake, nmake and jam, but is also unique in its multi-platform compatibility. Its shell command base allows ANT code to be ported across multiple platforms with little to no changes – a direct inheritance of Java's 'Write Once, Run Anywhere' feature.

Tools such as ANT can be highly attractive – or tempting – for many developers, who quickly grow to appreciate its ability to automate laborious tasks. This has resulted in many supplementary functions being added on over time; these include SQL execution, FTP/Telnet use, .NET linkage and compatibility for an increasing number of specialised products such as Visual Age.

One of the alternatives to ANT is Maven, a tool which can standardize and manage the build process at a project level. This is in contrast to ANT itself, where each build process needs to be set up separately, and any new developers would need to understand the particular process in use before contributing. Maven offers a more standardized and structured approach to build processes by centralizing its resource management; this allows projects with multiple developers to organize its output more efficiently. Maven is also designed to manage .jar files used in the project together with other .jar files that the project may be dependent on.

In many ways, Maven recommends and facilitates best practice in development. However, it is, by nature, less flexible than ANT. In this book we will stay with ANT for our sample application; in any case, Maven is capable of retrieving tasks built in ANT, should you be interested in using it.

Let's begin by installing ANT. For most users, the latest binary edition from ANT's homepage (http://ant.apache.org/) is sufficient; however, a source edition is also available for those who would like to delve into its operations a little further.

Once installed, you can see that ANT is structured in the following way. In particular, 'bin' and 'lib' are crucial for ANT's normal operations.

- **bin**: Stores the scripts required to execute ANT.
- **lib**: Stores the critical .jar files.
- **docs**: Stores all documentation for using ANT.
- **etc**: Stores various types of .xls files, which are then used to produce .xml format reports on the result of performed tasks.

As we learned in the previous section, we now need to set the environment variables for our project. Before running ANT, the variable ANT_HOME should be set to a relative path rather than an absolute one, after which ANT_HOME should be added to the PATH variable. As ANT is Java-based, it might also be helpful to define JAVA_HOME.

Afterwards, let's try running ANT. For now, let's simply open the command window, navigate to the CrePetstore sample application's root folder, then run the command 'ant'. Depending on your system, this will set off a lengthy and rather confusing process, as ANT gets to work (see below diagram). It's our job now to learn what exactly is happening here, and how we can optimize what it does for our own project's needs.

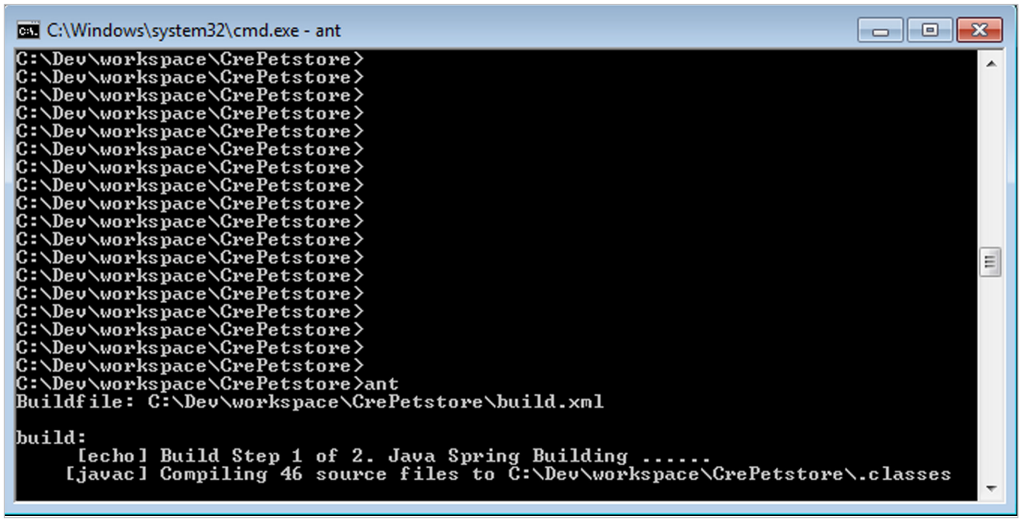

Figure 1.18 Ant Build Execution in Command Window

ANT's Operational Principles

How exactly does ANT work? We can learn the answer through a 'build.xml' file, contained in the installation package. The ANT executable simply reads this file and carries out its instructions. Let's try opening ANT through Eclipse to examine the contents of the default 'build.xml' file:

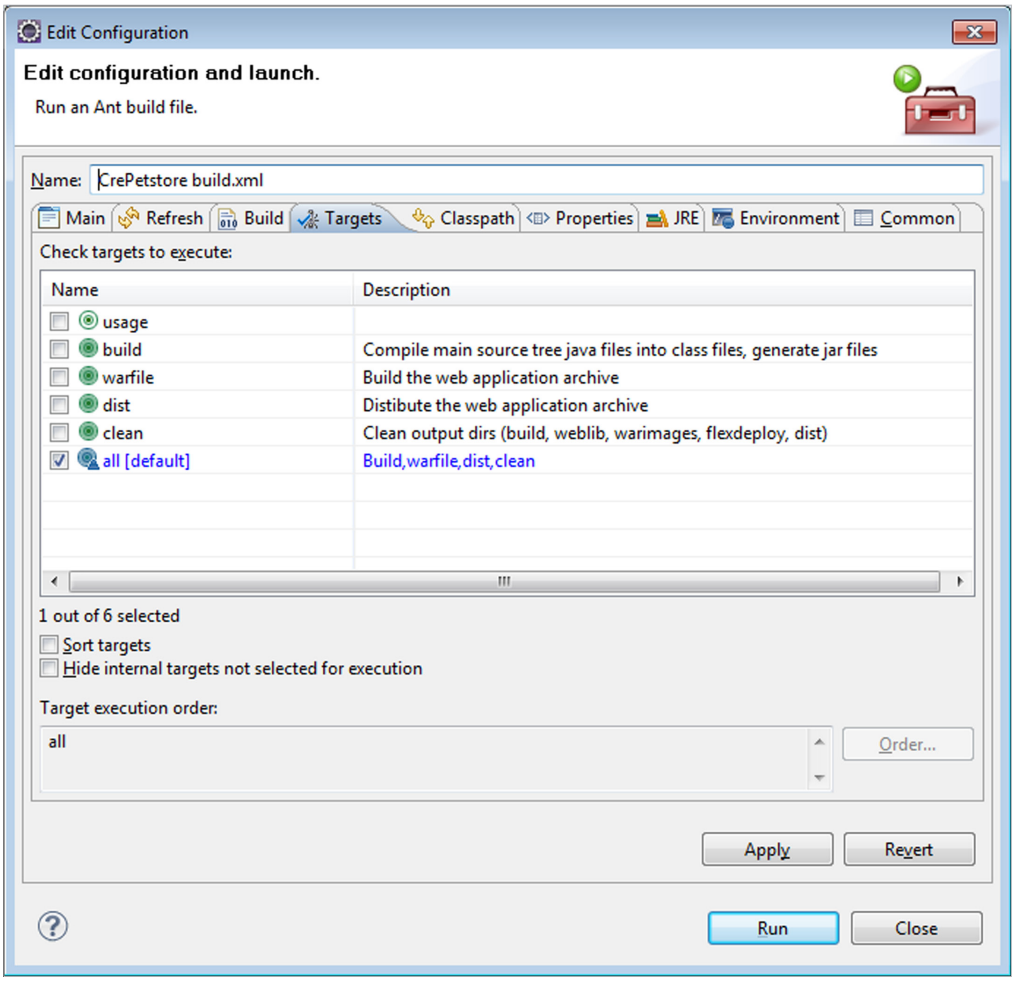

Figure 1.19 Ant Build Execution in Eclipse

As you can see, the file assigns multiple targets under a single project; these targets, in turn, consist of various tasks. It is this hierarchy of projects, targets and tasks that coordinate to create ANT as a whole. The build file's XML base allows

it to be easily modified or reused, which fits nicely with ANT's aforementioned compatibility across multiple platforms.

Let's examine the build file's elements one by one. The first is the project tag, which all build files must begin with:

```
<project name="crepetstore" basedir=". " default="all">
```

- **name**: String type; can be omitted. Displays the name of the project.
- **default**: String type. The target defined here is used when the user does not specify a target, and cannot be omitted. The 'ant' command that we tried earlier on is an example of this – because we did not specify a target, the command was simply interpreted as 'ant all'.
- **basedir**: Path type; can be omitted. The above example, '.' simply denotes the current directory. This is the base directory on which other pathnames are defined.

Next is the 'target'. As discussed previously, a single project can have multiple targets, which can be organized in terms of dependency.

```
<target name="all" depends="build,warfile,dist,clean"
        description="Build,warfile,dist,clean"/>
```

- **name**: String type; can be omitted. Displays the name of the project.
- **depends**: String type; can be omitted. Denotes the target(s) that the current target is dependent on (and must be run first).
- **if**: String type; can be omitted. Denotes the conditions that must be fulfilled in order for this target to run.
- **Unless**: String type; can be omitted. Denotes the conditions that must not be fulfilled – the opposite of 'if'.
- **Description**: String type, can be omitted. Describes the target.

Before moving on to tasks, we also need to touch on several other aspects of targets – property, path and fileset. These can increase the readability of the project, and in turn, the code's reusability. Making full use of these elements can optimize your code for future extensions or modifications.

As shown below, the property is defined at the beginning of the code, and can then be reused within the project.

```
<property environment="env"/>
<property name="JAVA_HOME" value="${env.JAVA_HOME}"/>
……
<property name="war.dir" value="war"/>
<property name="war-WEB-INF.dir" value="${war.dir}/WEB-INF"/>
```

In this case, the property 'war.dir' denotes the directory of the same name, and can be reused as '${war.dir}'. This means that even when the directories have been moved, only the original property path needs to be modified to reflect the change.

Paths commonly used together with properties are called 'Path-like Structures'. The path first defines the id value, then uses a subordinate element, 'pathelement', to define the specific file or directory. In the below example, we set the id value as 'master-classpath', then add on two previously defined properties - ${build.dir} and ${spring-src.dir}. Once defined in this way, the id value can be referenced anywhere in the project using refid.

```
<path id="master-classpath">
    <pathelement location="${build.dir}"/>
    <pathelement location="${spring-src.dir}"/>
        <fileset dir="${spring-devlib.dir}">
            <include name="**" />
        </fileset>
</path>
……
<javac destdir="${build.dir}" ……>
    <src path="${spring-src.dir}"/>
    <classpath refid="master-classpath"/>
</javac>
```

Next is the fileset. As the name suggests, fileset is a type that represents the combination of file and directory, and forms the basis of directory-based tasks together with types such as patternset. Tasks often make use of filesets to select or filter out specific directories or files. In the below example, the first fileset is

configured to include all files (**) under the ${database.dir} directory, itself a subdirectory of ${spring-src-resources.dir}. We could also use the <exclude> tags to filter out specific directories and files within that selection.

```
<fileset dir="${spring-src-resources.dir}">
    <include name="${database.dir}/**"/>
    ……
</fileset>
```

Finally, let's examine the tasks themselves. While ANT is a very powerful tool that can be packaged with many supplementary functions, we will limit ourselves here to ANT's more basic and standard functions; ANT itself is well documented in its own home page, should you need further reference.

The first task we will look at is 'delete', which can be used not only to delete the selected file or category, but also an entire fileset. Empty directories are not deleted unless otherwise specified through <delete includeEmptyDirs="true">. Another task, 'mkdir', performs the inverse function – to create new directories.

```
<delete file="${catalina.dir}/${deploy.name}.war"/>
<delete dir="${catalina.dir}/${deploy.name}"/>
……
<mkdir dir="${war.dir}"/>
```

One task that we need to understand very clearly is 'javacTask'. Javac is a core task for any developer that uses ANT to create Java-based web applications. Let's use the example below to learn how it works. The first thing to note is that 'src path' denotes javac's source location – in this case, ${spring-src.dir}, or in plain, /src/main/java. In turn, 'destdir' is the destination where the class file is to be created. In our example, we have set it to ${build.dir}. So far so good; but here is when we run into something new – 'deprecation'. JavacTask sets this option to 'off' by default, but we have turned it on for our example. Deprecation works by checking whether the code contains anything that has been rendered obsolete or unsupported by later updates in the JDK, and then notifies the user in the command window. Since we have also set the debug property to true, the debug information will be displayed as well. Finally, we define the 'classpath', and could also have used the include and exclude functions to customize our selection of files and directories. JavacTask is one of the more customizable tasks in ANT, but is not wholly different from many other tasks available. Check the ANT manual (http://ant.apache.org/manual/index.html) for more detail on ANT tasks.

```
<javac destdir="${build.dir}" source="1.5" target="1.5" debug="true"
    deprecation="false" optimize="false"
    failonerror="true"includeantruntime="false">
        <src path="${spring-src.dir}"/>
        <classpath refid="master-classpath"/>
</javac>
```

We will now review how ANT can be used to compile Flex application projects. We begin by ensuring compatibility between Flex and ANT through the use of 'taskdef' tags. There are three Flex ANT tasks available for use:

```
<taskdef resource="flexTasks.tasks"
        classpath="${FLEX_HOME}/ant/lib/flexTasks.jar"/>
```

- **mxmlc**: Used to compile Flex applications, modules, resource modules and CSS .swf files.
- **compc**: Used to compile .swc files and Runtime Shared Libraries (RSLs).
- **html-wrapper**: Creates a HTML wrapper and supporting files for your Flex application. This command also supports a number of settings, such as with/without deep linking support, with/without express install, and with/without player detection. It can also be used to set the height, width, background color and other details for the application.

The below code demonstrates how the html-wrapper tag is used to create the Flex application's HTML wrapper, setting some key attributes in the process:

```
<html-wrapper
    title="Creative PetStore Sample Application"
    file="${flexapp.name}.html"
    height="100%"
    width="100%"
    application="app"
    swf="${flexapp.name}"
    history="true"
    express-install="true"
    version-detection="true"
    output="${flexdeploy.dir}"/>
```

- **application**: The name of the .swf object used in the HTML wrapper. This name can be referred to when using JavaScript or External Interface APIs.
- **swf**: Sets the name of the .swf file that the HTML wrapper embeds. In this case, we use ${flexapp.name} to create 'Main.html', previously defined as a property.
- **template**: Sets the template to output type. There are three available: 'client-side-detection', 'express-installation' and 'no-player-detection'. The default is 'express-installation'.

Another Flex-related task is 'mxmlc' – used to create the application and related modules. Again, let's review the key attributes in the task code:

```
<mxmlc file="${flexweb-src.dir}/${flexapp.name}.mxml"
    output="${flexdeploy.dir}/${flexapp.name}.swf"
    default-background-color="${flexapp.background}"
    actionscript-file-encoding="UTF-8"
    keep-generated-actionscript="false"
    incremental="true">
    <load-config filename="${FLEX_HOME}/frameworks/flex-config.xml"/>
    <source-path path-element="${FLEX_HOME}/frameworks"/>
    <!-- List of SWC files or directories that contain SWC files. -->
    <compiler.library-path dir="${FLEX_HOME}/frameworks" append="true">
    <include name="libs" />
    </compiler.library-path>
    <compiler.library-path dir="${basedir}" append="true">
    <include name="${flexweb-devlib.dir}" />
    </compiler.library-path>
</mxmlc>
```

- **file**: Defines the name of the Flex application. Again, we use the previously defined ${flexapp.name} to use 'Main.mxml'.
- **output**: Defines the name of the compiled Flex application.
- **actionscript-file-encoding**: Sets the type of actionscript encoding. The sample application uses 'UTF-8' for maximum compatibility across languages.

- **incremental**: Sets the type of the compiled cache file. If the application is recompiled, the task can check for the cache file and compile only the modified areas.
- **keep-generated-actionscript**: Whether the automatically generated action-script – used to create the mxml code - is retained.
- **load-config**: The filepath of 'flex-config.xml', the Flex application's configuration file. The default option is the file contained in the SDK.
- **compiler.library-path**: Includes the specified libraries in the Flex application. Note that this will increase the application's size.

All of these various functions in ANT that we have learned about can often be automated further through the use of macros. While we will cover this issue in greater detail in the 'Localization' section, let's use the example of building locale modules for each supported locale.

```
<!-- Flex Module Building. -->
<parallel threadCount="${threads.maximum}">
    <compileLocale locale="en_US" />
    <compileLocale locale="ko_KR" />
</parallel>
……
<macrodef name="compileLocale"
    description="Compiles the Resource package for the given locale">
    <attribute name="locale" default="en_US" />
    <sequential>
    <mxmlc output="${flexdeploy.dir}/@{locale}_ResourceModule.swf">
        <locale>@{locale}</locale>
        <source-path path-element="${flexweb-src.dir}/locale/{locale}" />
        <include-resource-bundles>
            localeproperties</include-resource-bundles>
        <source-path path-element="${FLEX_HOME}/frameworks" />
    </mxmlc>
    </sequential>
</macrodef>
```

This macro, defined as 'compileLocale', contains the code necessary to generate locale modules – in this case, for 'en_US' (American English) and 'ko_KR' (Korean). Since the actual process of generating locale modules is the same across various locales, the macro offers a more efficient way of generating multiple locales.

One last thing we need to cover is what to do once the application is compiled. We need to take the newly created deployment file, and automate its distribution to the Tomcat web application server. To do this, we need to first retrieve the location of Tomcat from the OS' environment variables. The next step is to run the 'warfile' Task, which generates the .war file for distribution. Finally, the file is sent to Tomcat through the 'dist' Task. This task automatically deletes the previous .war file and folder before sending the new file.

```
<property environment="env"/>
<property name="JAVA_HOME" value="${env.JAVA_HOME}"/>
<property name="CATALINA_HOME" value="${env.CATALINA_HOME}"/>
<property name="FLEX_HOME" value="${env.FLEX_HOME}"/>
<property name="catalina.base" value="${CATALINA_HOME}/conf"/>
<property name="catalina.dir" value="${CATALINA_HOME}/webapps"/>
……….
<target name="dist" depends="warfile"
    description="Distibute the web application archive">
    <delete file="${catalina.dir}/${deploy.name}.war"/>
    <delete dir="${catalina.dir}/${deploy.name}"/>
    <copy file="${dist.dir}/${deploy.name}.war"
          tofile="${catalina.dir}/${deploy.name}.war"/>
</target>
```

[Symbols]

[A]

[B]

[Q]

[R]

[S]

[T]

[U]

[V]

[W]

[X]

[Y]