Quick answers to common problems

Mule ESB Cookbook

Over 40 recipes to effectively build your enterprise solutions from the ground up using Mule ESB

ATTUNE INFOCOM
Tune into Enterprise Open Source

Dr. Zakir Laliwala
Azaz Desai
Abdul Samad
Uchit Vyas

[PACKT] PUBLISHING
open source*
community experience distilled

Mule ESB Cookbook

Over 40 recipes to effectively build your enterprise solutions from the ground up using Mule ESB

Dr. Zakir Laliwala

Abdul Samad

Azaz Desai

Uchit Vyas

[PACKT] open source
community experience distilled
PUBLISHING

BIRMINGHAM - MUMBAI

Mule ESB Cookbook

First published: August 2013

Production Reference: 1160813

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78216-440-1

www.packtpub.com

Cover Image by Rakesh Shejwal (shejwal.rakesh@gmail.com)

Credits

Authors

Dr. Zakir Laliwala

Abdul Samad

Azaz Desai

Uchit Vyas

Reviewers

Lieven Heuninck

Maurizio Turatti

Acquisition Editor

Kartikey Pandey

Lead Technical Editor

Dayan Hyames

Technical Editors

Jalasha D'costa

Amit Ramadas

Copy Editors

Gladson Monteiro

Sayanee Mukherjee

Aditya Nair

Adithi Shetty

Laxmi Subramanian

Project Coordinator

Kranti Berde

Proofreader

Linda Morris

Indexers

Hemangini Bari

Tejal Soni

Graphics

Sheetal Aute

Production Coordinator

Arvindkumar Gupta

Cover Work

Arvindkumar Gupta

About the Authors

Dr. Zakir Laliwala is an entrepreneur, open source specialist, and hands-on CTO of Attune Infocom. Attune Infocom provides enterprise-level open source solutions, and services for SOA, BPM, ESB, Portal, Cloud computing, and ECM. At Attune Infocom, he is responsible for solutions and services delivery and product development. He is exploring new enterprise-level open source solutions, and defining architecture, roadmap, and best practices. He has provided consulting and training on various open source technologies to corporates around the world on Mule ESB, Activiti BPM, JBoss' jBPM and Drools, Liferay Portal, Alfresco ECM, Jboss SOA, and Cloud computing.

Dr. Zakir pursued Ph.D. in Information and Communication Technology from Dhirubhai Ambani Institute of Information and Communication Technology. He was the Adjunct Faculty at Dhirubhai Ambani Institute of Information and Communication Technology (DA-IICT) and CEPT University, where he now teaches students pursuing Master's.

He has published many research papers in IEEE and ACM international conferences on web services, SOA, Grid computing, and Semantic Web. He also serves as a reviewer at various international conferences and journals. He has contributed chapters on open source technologies and writes books on open source technologies.

Abdul Samad has more than seven years' hands-on experience in leading and implementing Java, J2EE, Portal, and ECM open source solutions. He has successfully migrated IBM WebSphere portal to Liferay Portal for a client based in the U.K. He has delivered successful training with his experience and expertise on Liferay Portal and jBPM to Sambaash, AT&T, Cognizant, Urja Technologies, and Protea Technologies. He was part of an enterprise-level, open source portal application implementation for media and publication houses, portal customization projects, and led a team of developers to achieve the client's requirements on time.

He has expertise in implementing J2EE technologies (JSP, Servlet, MVC Frameworks, BPM, ESB, and Portlet frameworks) to develop enterprise web applications. He has worked with various frameworks such as Mule ESB, jBPM, Liferay, Alfresco, and Oracle WebLogic portal on his journey.

Azaz Desai has more than three years' experience in Mule ESB, JBPM, and Liferay technologies. He is an Oracle Certified Java Programmer (OCJP). He is responsible for implementing, deploying, integrating, and optimizing services and business processes using ESB and BPM tools. He is a lead writer of *Mule ESB Cookbook*, *Packt Publishing*, as well playing a vital role of trainer on ESB to global clients at Attune Infocom.

He is very enthusiastic and active in understanding client-specific requirements on web service integration. He has done various integration of web services, such as Mule ESB with Liferay, Alfresco, jBPM, and Drools. He was a part of a key project on Mule ESB integration as a messaging system. He has worked on various web service standards and frameworks, such as CXF, AXIS, SOAP, and REST.

Uchit Vyas a B.Tech. graduate in Computer Science with a research interest in ESB and Cloud, is a certified Cloud Architect (AWS), Cisco (CCNA), VMware (VSP), and Red Hat Linux (RHCE) professional. He has an energetic strength to work on multiple platforms at a time and the ability to integrate open source technologies. He works as a Sr. Consultant and looks after AWS – Cloud, Mule ESB, Alfresco, Liferay and deploying Portal, and ECM system. He was previously working with TCS as an Assistant System Engineer.

With over three years' hands-on experience on open source technologies, he manages to guide the team and deliver projects and training sessions meeting client expectations. He has provided more than 13 training sessions on Cloud computing, Alfresco, and Liferay in a couple of months. In the last few years, he has moved over 80 percent of Attune Infocom business processes to the Cloud by implementing agile SDLC methodology on Amazon, Rackspace, and private Clouds such as Eucalyptus and OpenStack. His skills are not limited to designing and managing Cloud environment/infrastructure, server architecture. He is also active in Shell scripting, autodeployment, supporting hundreds of Linux and Windows physical and virtual servers hosting databases, and applications with continuous delivery using Jenkins/CruiseControl with Puppet/Chef scripting.

About the Reviewers

Lieven Heuninck holds an MSc. degree. He has an up-to-date knowledge on technology and is very capable in designing technical solutions using state of the art technological components. He knows how to combine his technical skills with a good practical knowledge of the various functional processes present in today's organizations. He is passionate about Enterprise Architecture, Service-Oriented Architecture, and all the tools that can bring these concepts into reality. He is so passionate about it that he co-started a company called Apogado (`http://www.apogado.com`) to provide dedicated services in his areas of expertise. Apogado currently conducts various missions for large organizations in private and public sectors. When Lieven is not working, he enjoys sailing.

Maurizio Turatti is a software integration architect with more than 15 years' professional experience in many SOA and enterprise integration projects in Europe and the Middle East. He is now working as an Open Source Team Leader. He has worked formerly at SeeBeyond, Sun Microsystems, and Alfresco. He loves to research and experiment with leading edge open source technologies. You can contact him through his CamelCase blog: `http://blog.maurizioturatti.com/`.

He is also the author of the book *Instant Apache Maven Starter*, *Packt Publishing*.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

Mule ESB is a lightweight Java-based enterprise service bus (ESB) and integration platform that allows developers to connect applications together quickly and easily, enabling them to efficiently exchange data. You can therefore use Mule ESB to allow different applications to communicate with each other via a transit system to carry data between applications within your enterprise or across the Internet. It is also useful if you use more than one type of communication protocol while integrating three or more applications/services.

Mule ESB Cookbook takes readers through the practical approach of using Mule ESB 3.3. This book solves numerous issues faced by developers working on Mule ESB in real time and provides use cases on how to integrate Mule with other technologies. It also focuses on development and delivery using Mule ESB through integrating, migrating, and upgrading advanced technological tools.

This book gives the reader a strong overview of the Mule framework using practical and easy-to-follow examples. It has three sections: problems, approaches, and solutions. The key aim of this book is to show you how to allow different applications to communicate with each other by creating a transit system to carry data between applications within your enterprise or across the Internet. Mule ESB enables easy integration of existing systems, regardless of the different technologies that the applications use, including JMS, web services, JDBC, HTTP, and more.

Mule ESB Cookbook will teach you everything to communicate between applications that are built on different platforms, as well as how to migrate them into your application across multiple platforms or on the Cloud.

What this book covers

This book contains recipes related to deployment, scripting, and the API discussing core concepts of standard components, performance tuning, and Cloud integration through practical task-oriented recipes. This book will provide you practical knowledge of the Mule ESB architecture and its configuration. Core concepts and components required to understand how Mule ESB works are also explained.

Chapter 1, *Getting Started with Mule ESB*, discusses Mule core concepts and terminology. It also provides an environment setup for Mule ESB and Mule Studio. By the end of this chapter, you will be familiar with Mule IDE integration with Eclipse, and how to create a Hello World project and flow in Mule Studio. At the end of this chapter, you will learn how to configure Mule elements and deploy applications on the Mule server.

Chapter 2, *Working with Components and Patterns*, describes what a component is and its types, such as Echo, Logger, REST, SOAP, HTTP, and Java. You will also know how to configure a component, how to use it in a workflow, and what patterns are in Mule ESB.

Chapter 3, *Using Message Property, Processors, and Sources*, helps you understand what message sources, processors, and properties are. By the end of this chapter, you will be able to use processors in a workflow, and use message processors to control the message flow, and message property scopes.

Chapter 4, *Endpoints*, explains what an Endpoint is. Endpoints send and receive data and are responsible for connecting to external resources and delivering messages.The two types of Endpoints available in Mule Studio are: Inbound Endpoint and Outbound Endpoint. Inbound Endpoint is used for receive messages and Outbound Endpoint is used for sending messages.

Chapter 5, *Transformers*, explains what a transformer is. By the end of this chapter, you will be able to configure the JSON-to-Object and Object-to-XML transformers and DataMapper.

Chapter 6, *Configuring Filters*, explains what a filter is and how to configure the Logic filter. By the end of this chapter, you will be able to create a custom filter and configure the Message filter.

Chapter 7, *Handling Exceptions and Testing*, explains what an exception is. By the end of this chapter, you will be able to configure the Catch Exception Strategies, Rollback Exception Strategies, and JUnit testing.

Chapter 8, *Introducing Web Services*, explains what a web service is. By the end of this chapter, you will be able to create a JAX-WS web service and integrate external web services.

Chapter 9, *Understanding Flows, Routers, and Services*, explains what a Router is, and how to configure the Router and the Splitter Flow Control.

Chapter 10, *Configuring Cloud Connectors*, explains what a cloud connector is and how to integrate Twitter and Dropbox connectors.

What you need for this book

You will need the following software to be installed before running the code examples:

- Mule ESB requires JDK 6 or a later version. JDK 6 can be downloaded from the following site: `http://www.oracle.com/technetwork/java/javase/downloads/jdk6downloads-1902814.html`.
- Mule Studio is a powerful, user-friendly Eclipse-based tool. Mule Studio is an Eclipse-based tool that has three main components: a package tree, a palette, and a canvas. Mule ESB easily creates flows and edits and tests them in a few minutes. It is based on drag-and-drop elements and supports two-way editing. Mule Studio can be downloaded from the following site: `http://www.mulesoft.org/all-mule-studio-downloads`.
- You will also require PostgreSQL 9.2. PostgreSQL is a web hosting database that is used to store website information such as user information. PostgreSQL is a powerful, open source object-relational database system. It runs on all major operating systems, including Linux, Unix, and Windows. It is fully ACID compliant, and has full support for foreign keys, joins, views, triggers, and stored procedures. PostgreSQL can be downloaded from the following site: `http://www.piostgresql.org/download`.
- You will need Selenium IDE. It is an integrated development environment for Selenium scripts. It is implemented as a Firefox extension and allows you to record, edit, and debug tests. Selenium IDE includes the entire Selenium Core, allowing you to easily and quickly record and play back tests in the actual environment that they will run on. Selenium IDE is not only a recording tool, but it is also a complete IDE. You can choose to use its recording capability, or you may edit your scripts by hand. Selenium IDE can be downloaded from the following site: `http://docs.seleniumhq.org/download/`.

Who this book is for

This book provides solutions for developers who are working on Mule ESB and for integrators or migrators who are integrating and migrating Mule with other technologies. It focuses on development and delivery using Mule ESB through integrating, migrating, and upgrading advanced technological tools.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "Enter the project name, `Logic Filter`."

A block of code is set as follows:

```
package com.org;

public class User {

  private String name;
  private String lname;
  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
  public String getLname() {
    return lname;
  }
  public void setLname(String lname) {
    this.lname = lname;
  }

}
```

Any command-line input or output is written as follows:

```
java -jar selenium-server-standalone-2.31.0.jar
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Click on **Next** and then on **Finish**."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Getting Started with Mule ESB

In this chapter, we will cover the following topics:

- Understanding Mule concepts and terminologies
- Setting up the Mule IDE
- Installing Mule Studio
- Configuring Mule components
- Deploying your first Hello World application on the Mule server

Introduction

Mule ESB is a lightweight Java programming language. Through ESB, you can integrate or communicate with multiple applications. Mule ESB enables easy integration of existing systems, regardless of the different technologies that the applications use, including JMS, web services, JDBC, and HTTP.

Understanding Mule concepts and terminologies

Enterprise Service Bus (**ESB**) is an application that gives access to other applications and services. Its main task is to be the messaging and integration backbone of an enterprise.

An ESB is a distributed middleware system to integrate different applications. All these applications communicate through the ESB. It consists of a set of service containers that integrate various types of applications. The containers are interconnected with a reliable messaging bus.

Getting ready

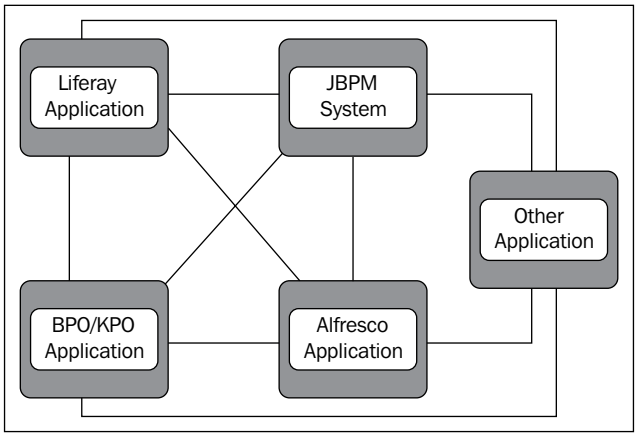
An ESB is used for integration using a service-oriented approach. Its main features are as follows:

- Polling JMS
- Message transformation and routing services
- Tomcat hot deployment
- Web service security

We often use the abbreviation, **VETRO**, to summarize the ESB functionality:

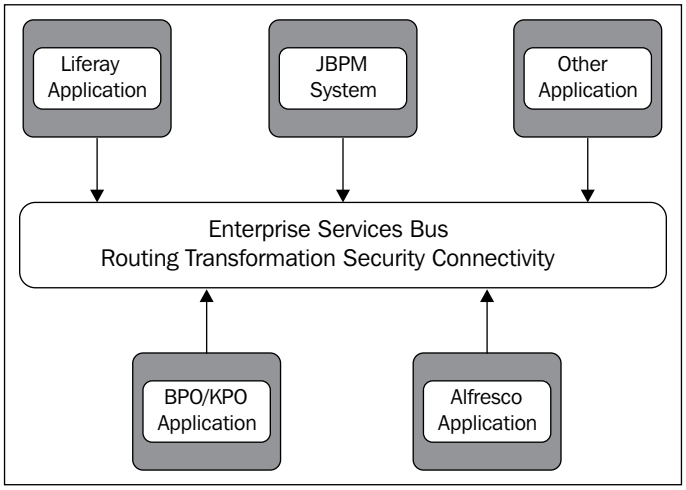
- **V** – validate the schema validation
- **E** – enrich
- **T** – transform
- **R** – route (either itinerary or content based)
- **O** – operate (perform operations; they run at the backend)

Before introducing any ESB, developers and integrators must connect different applications in a point-to-point fashion.



How to do it...

After the introduction of an ESB, you just need to connect each application to the ESB so that every application can communicate with each other through the ESB. You can easily connect multiple applications through the ESB, as shown in the following diagram:



Need for the ESB

You can integrate different applications using ESB. Each application can communicate through ESB:

- To integrate more than two or three services and/or applications
- To integrate more applications, services, or technologies in the future
- To use different communication protocols
- To publish services for composition and consumption
- For message transformation and routing

What is Mule ESB?

Mule ESB is a lightweight Java-based enterprise service bus and integration platform that allows developers and integrators to connect applications together quickly and easily, enabling them to exchange data. There are two editions of Mule ESB: Community and Enterprise. Mule ESB Enterprise is the enterprise-class version of Mule ESB, with additional features and capabilities that are ideal for clustering and performance tuning, DataMapper, and the SAP connector. Mule ESB Community and Enterprise editions are built on a common code base, so it is easy to upgrade from Mule ESB Community to Mule ESB Enterprise.

Mule ESB enables easy integration of existing systems, regardless of the different technologies that the applications use, including JMS, web services, JDBC, and HTTP. The key advantage of an ESB is that it allows different applications to communicate with each other by acting as a transit system for carrying data between applications within your enterprise or across the Internet. Mule ESB includes powerful capabilities that include the following:

- **Service creation and hosting**: It exposes and hosts reusable services using Mule ESB as a lightweight service container
- **Service mediation**: It shields services from message formats and protocols, separate business logic from messaging, and enables location-independent service calls
- **Message routing**: It routes, filters, aggregates, and re-sequences messages based on content and rules
- **Data transformation**: It exchanges data across varying formats and transport protocols

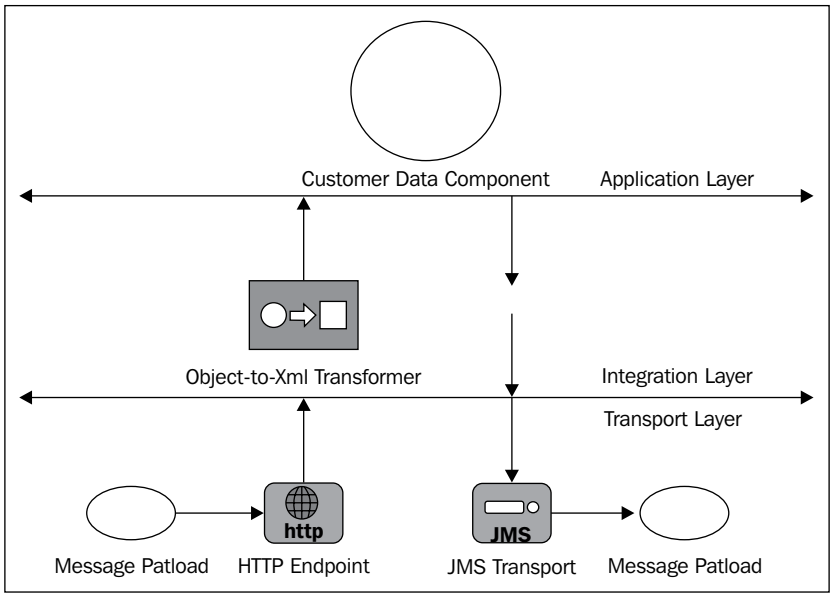
Mule ESB is lightweight but highly scalable, allowing you to start small and connect more applications over time. Mule provides a Java-based messaging framework. Mule manages all the interactions between applications and components transparently. Mule provides transformation, routing, filtering, Endpoint, and so on.

How it works...

When you examine how a message flows through Mule ESB, you can see that there are three layers in the architecture, which are listed as follows:

- Application Layer
- Integration Layer
- Transport Layer

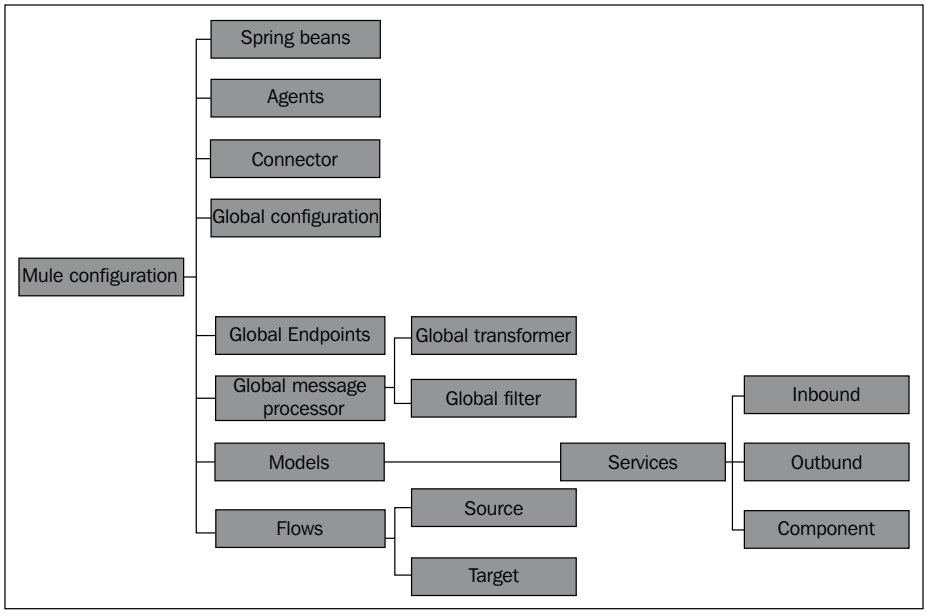
Likewise, there are three general types of tasks you can perform to configure and customize your Mule deployment. Refer to the following diagram:



The following list talks about Mule and its configuration:

- **Service component development**: This involves developing or re-using the existing POJOs, which is a class with attributes and it generates the get and set methods, Cloud connectors, or Spring Beans that contain the business logic and will consume, process, or enrich messages.
- **Service orchestration**: This involves configuring message processors, routers, transformers, and filters that provide the service mediation and orchestration capabilities required to allow composition of loosely coupled services using a Mule flow. New orchestration elements can be created also and dropped into your deployment.

- **Integration**: A key requirement of service mediation is decoupling services from the underlying protocols. Mule provides transport methods to allow dispatching and receiving messages on different protocol connectors. These connectors are configured in the Mule configuration file and can be referenced from the orchestration layer. Mule supports many existing transport methods and all the popular communication protocols, but you may also develop a custom transport method if you need to extend Mule to support a particular legacy or proprietary system.



- **Spring beans**: You can construct service components from Spring beans and define these Spring components through a configuration file. If you don't have this file, you will need to define it manually in the Mule configuration file.
- **Agents**: An agent is a service that is created in Mule Studio. When you start the server, an agent is created. When you stop the server, this agent will be destroyed.
- **Connectors**: The Connector is a software component.
- **Global configuration**: Global configuration is used to set the global properties and settings.
- **Global Endpoints**: Global Endpoints can be used in the **Global Elements** tab. We can use the global properties' element as many times in a flow as we want. For that, we must pass the global properties' reference name.
- **Global message processor**: A global message processor observes a message or modifies either a message or the message flow; examples include transformers and filters.

- **Transformers**: A transformer converts data from one format to another. You can define them globally and use them in multiple flows.
- **Filters**: Filters decide which Mule messages should be processed. Filters specify the conditions that must be met for a message to be routed to a service or continue progressing through a flow. There are several standard filters that come with Mule ESB, which you can use, or you can create your own filters.
- **Models**: It is a logical grouping of services, which are created in Mule Studio. You can start and stop all the services inside a particular model.
- **Services**: You can define one or more services that wrap your components (business logic) and configure Routers, Endpoints, transformers, and filters specifically for that service. Services are connected using Endpoints.
- **Endpoints**: Services are connected using Endpoints. It is an object on which the services will receive (inbound) and send (outbound) messages.
- **Flow**: Flow is used for a message processor to define a message flow between a source and a target.

Setting up the Mule IDE

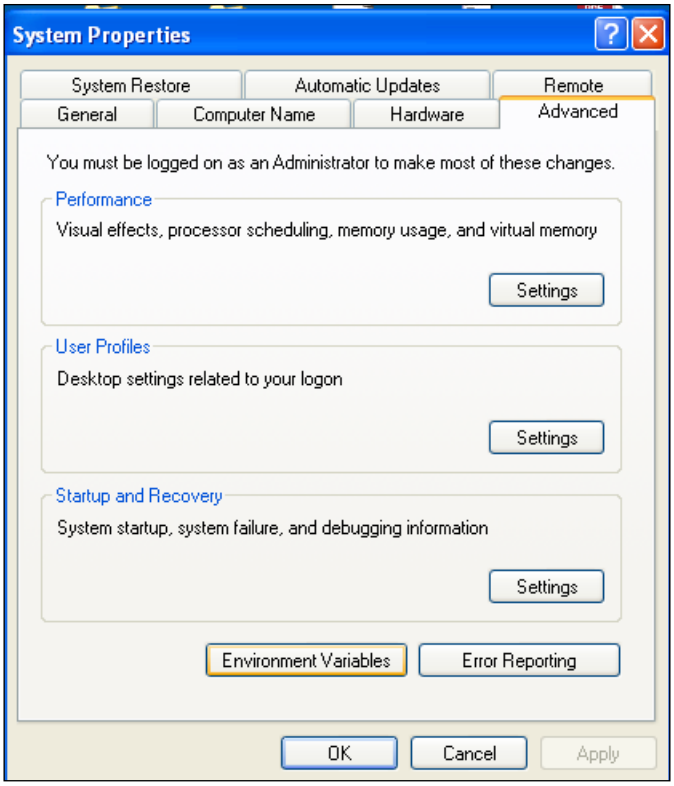
The developers who were using Mule ESB over other technologies such as Liferay Portal, Alfresco ECM, or Activiti BPM can use Mule IDE in Eclipse without configuring the standalone Mule Studio in the existing environment. In recent times, MuleSoft (`http://www.mulesoft.org/`) only provides Mule Studio from Version 3.3 onwards, but not Mule IDE. If you are using the older version of Mule ESB, you can get Mule IDE separately from `http://dist.muleforge.org/mule-ide/releases/`.

Getting ready

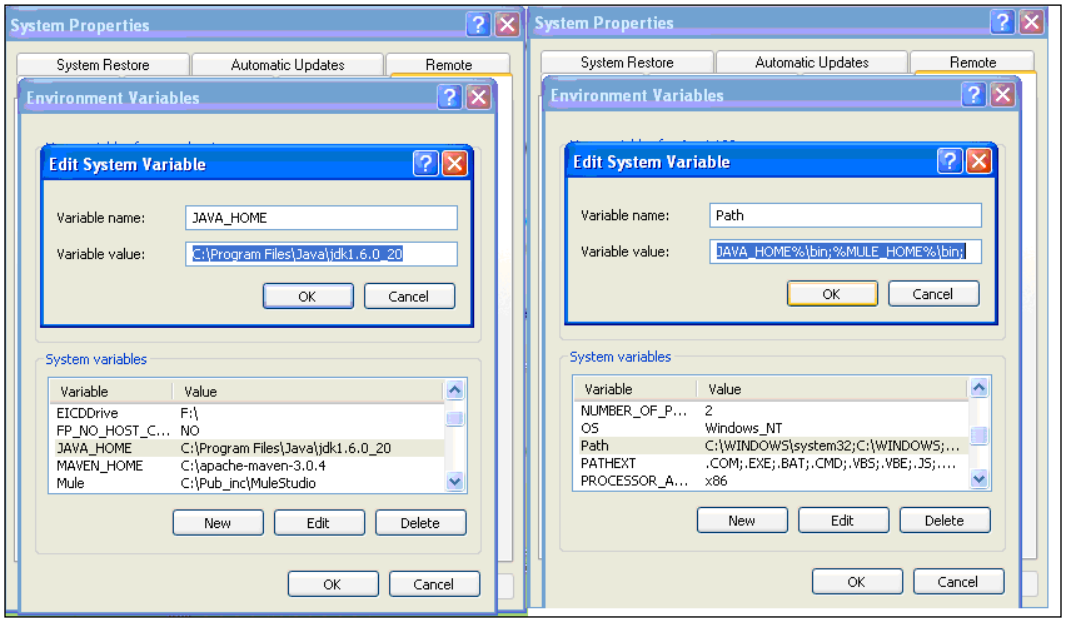
To set Mule IDE, we need Java to be installed on the machine and its execution path should be set in an environment variable. We will now see how to set up Java on our machine.

1. Firstly, download JDK 1.6 or a higher version from the following URL: `http://www.oracle.com/technetwork/java/javase/downloads/jdk6downloads-1902814.html`.
2. In your Windows system, go to **Start** | **Control Panel** | **System** | **Advanced**.

3. Click on **Environment Variables** under **System Variables**, find **Path**, and click on it.



4. In the **Edit** window, modify the path by adding the location of the class to its value. If you do not have the item **Path**, you may select the option of adding a new variable and adding `Path` as the name and the location of the class as its value.



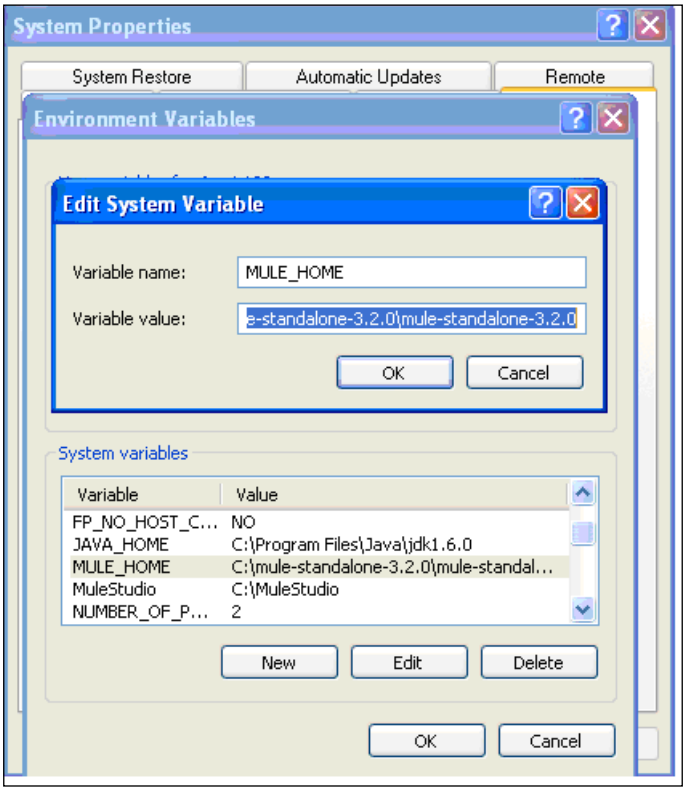
5. Close the window, reopen the command prompt window, and run your Java code.

How to do it...

If you go with Eclipse, you have to download Mule IDE Standalone 3.3.

1. Download Mule ESB 3.3 Community edition from the following URL: `http://www.mulesoft.org/extensions/mule-ide`. Unzip the downloaded file and set `MULE_HOME` as the environment variable.

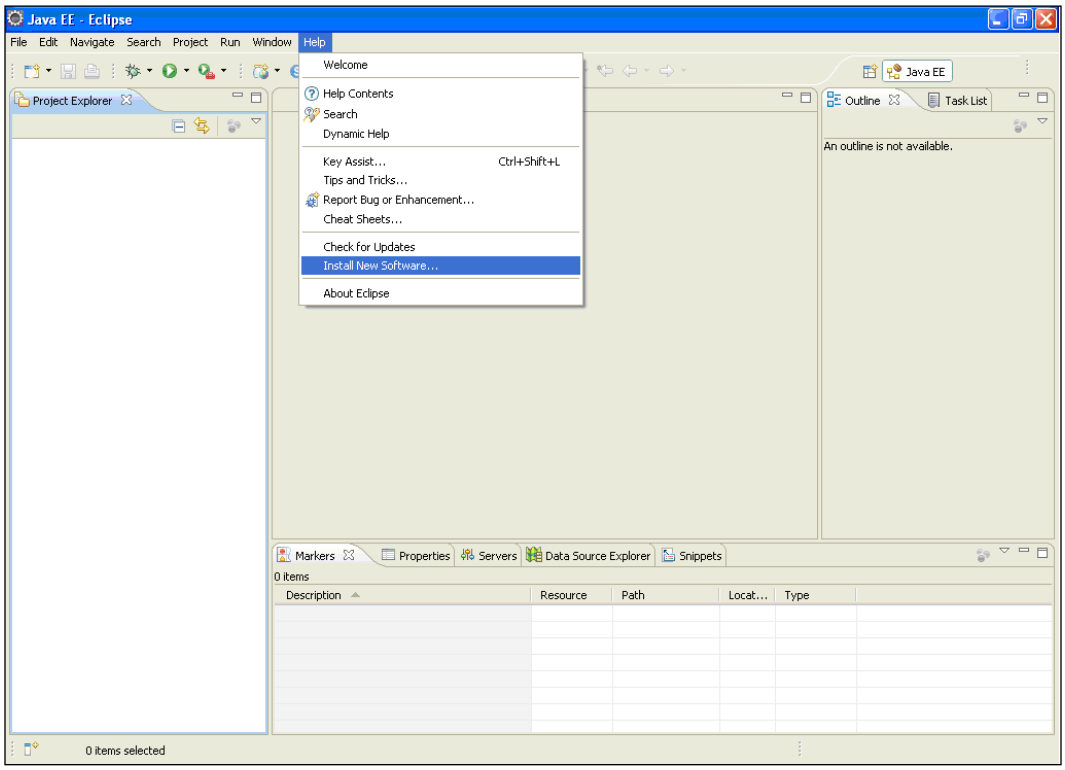
2. Download the latest version of Eclipse from `http://www.eclipse.org/downloads/`.



After installing Eclipse, you now have to integrate Mule IDE in the Eclipse. If you are using Eclipse Version 3.4 (Galileo), perform the following steps to install Mule IDE. If you are not using Version 3.4 (Galileo), the URL for downloading will be different.

1. Open Eclipse IDE.

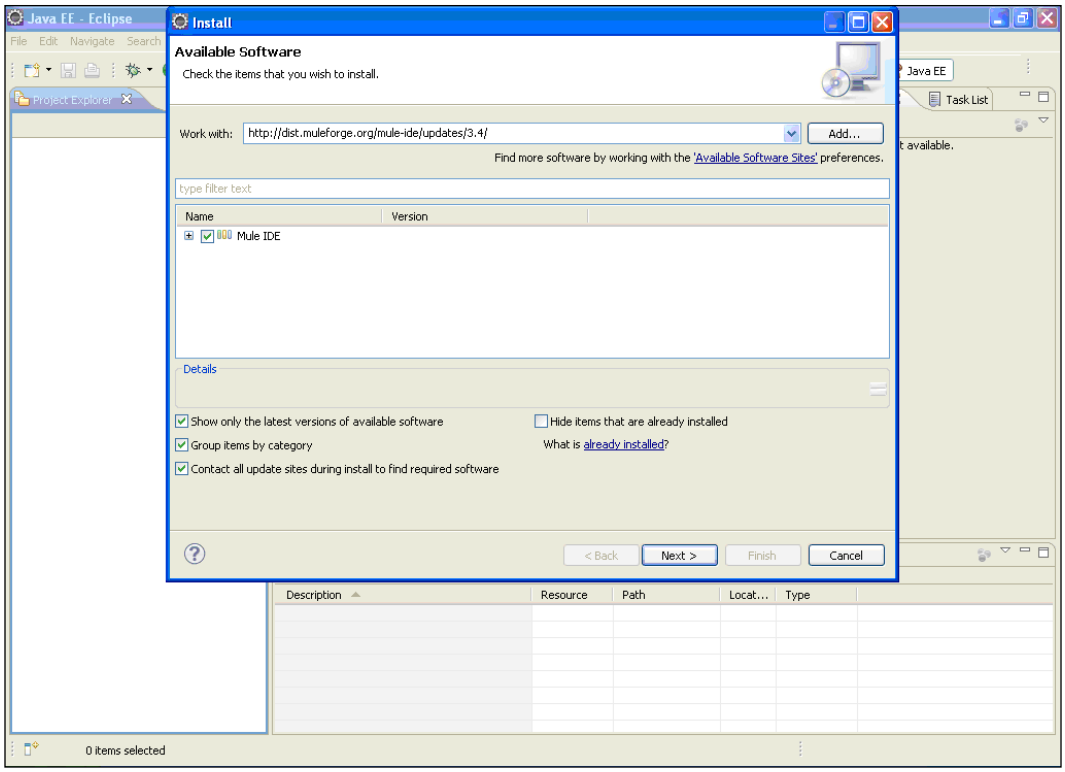
2. Go to **Help** | **Install New Software...**.



3. Write the URL in the **Work with:** textbox: `http://dist.muleforge.org/mule-ide/updates/3.4/` and press *Enter*.
4. Select the **Mule IDE** checkbox.
5. Click on the **Next** button.
6. Read and accept the license agreement terms.

7. Click on the **Finish** button.

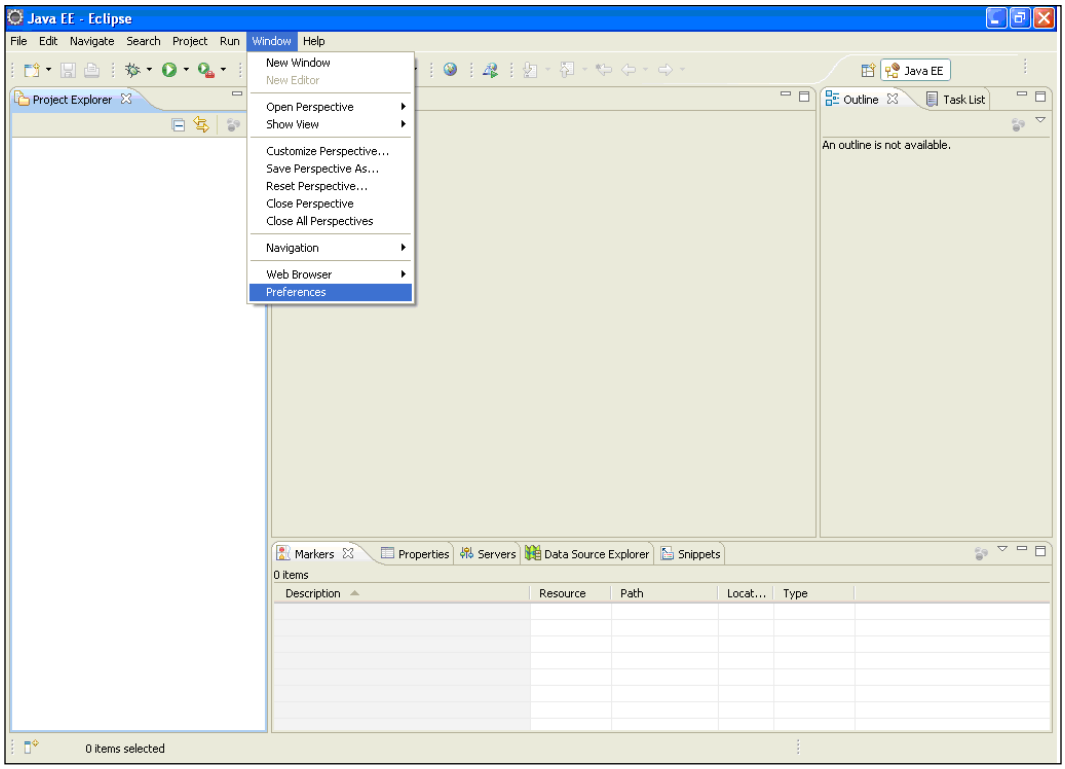
   This will take some time. When it prompts for a restart, shut it down and restart Eclipse.



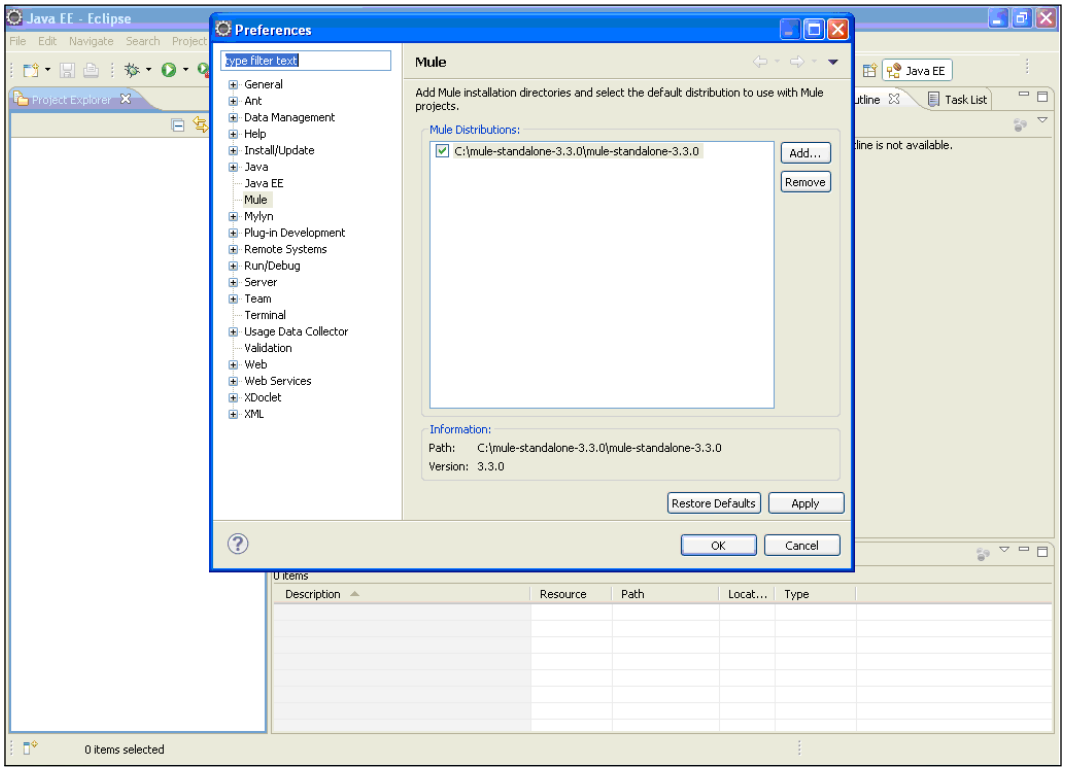
Mule configuration

After installing Mule IDE, you will now have to configure Mule in Eclipse. Perform the following steps:

1. Open Eclipse IDE.
2. Go to **Window** | **Preferences**.



3. Select **Mule**, add the distribution folder mule as `standalone 3.3`; click on the **Apply** button and then on the **OK** button. This way you can configure Mule with Eclipse.



Installing Mule Studio

Mule Studio is a powerful, user-friendly Eclipse-based tool. Mule Studio has three main components: a package tree, a palette, and a canvas. Mule ESB easily creates flows as well as edits and tests them in a few minutes. Mule Studio is currently in public beta. It is based on drag-and-drop elements and supports two-way editing.

Getting ready

To install Mule Studio, download Mule Studio from `http://www.mulesoft.org/download-mule-esb-community-edition`.

How to do it...

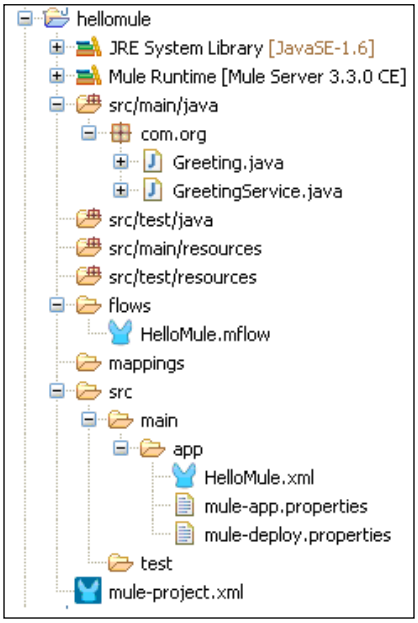
Unzip the `Mule Studio` folder. Set the environment variable for Mule Studio. While starting with Mule Studio, the `config.xml` file will be created automatically by Mule Studio.

The three main components of Mule Studio are as follows:

- A package tree
- A palette
- A canvas

A package tree

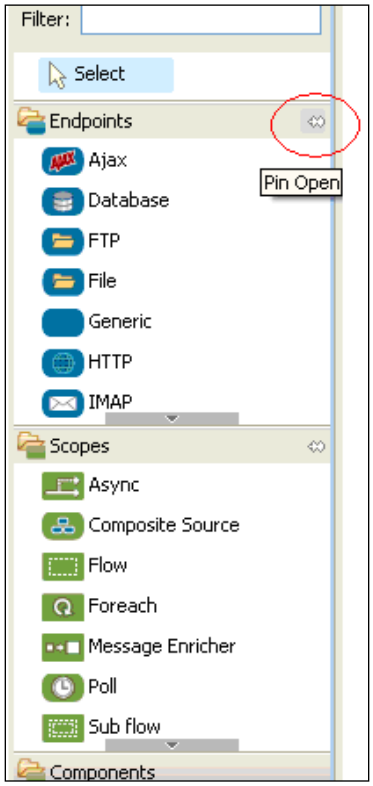
A **package tree** contains the entire structure of your project. In the following screenshot, you can see the package explorer tree. In this package explorer tree, under `src/main/java`, you can store the custom Java class. You can create a graphical flow from `src/main/resources`.



In the `app` folder you can store the `mule-deploy.properties` file. The folders `src`, `main`, and `app` contain the flow of XML files. The folders `src`, `main`, and `test` contain flow-related test files. The `Mule-project.xml` file contains the project's metadata. You can edit the name, description, and server runtime version used for a specific project. JRE System Library contains the Java runtime libraries. Mule Runtime contains the Mule runtime libraries.

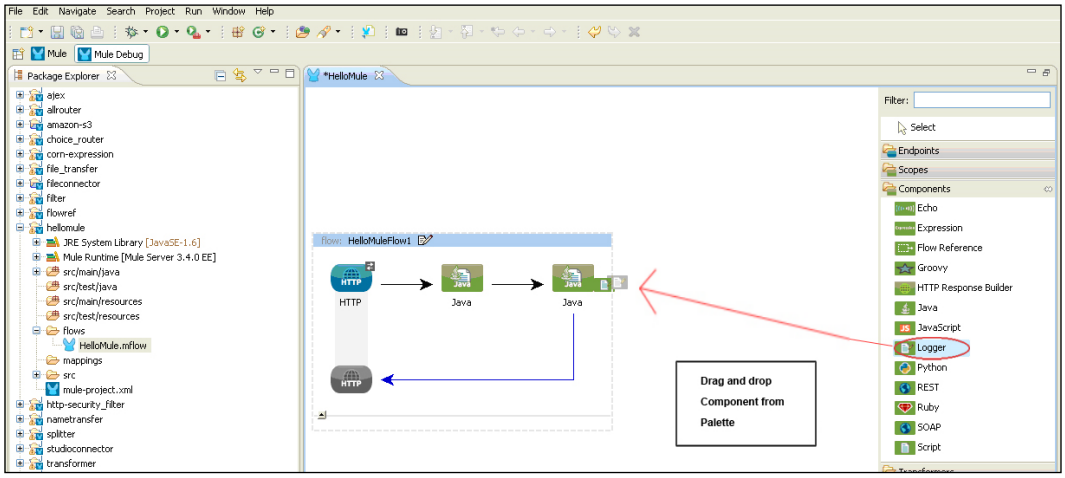
A palette

The second component is **palette**. The palette is the source for accessing Endpoints, components, transformers, and Cloud connectors. You can drag them from the palette and drop them onto the canvas in order to create flows. The palette typically displays buttons indicating the different types of Mule elements. You can view the content of each button by clicking on them. If you do not want to expand elements, click on the button again to hide the content.



A canvas

The third component is canvas; canvas is a graphical editor. In canvas you can create flows. The canvas provides a space that facilitates the arrangement of Studio components into Mule flows. In the canvas area you can configure each and every component, and you can add or remove components on the canvas.



Configuring Mule components

A simple POJO component will be invoked by Mule when a message is received. You can create your own custom component.

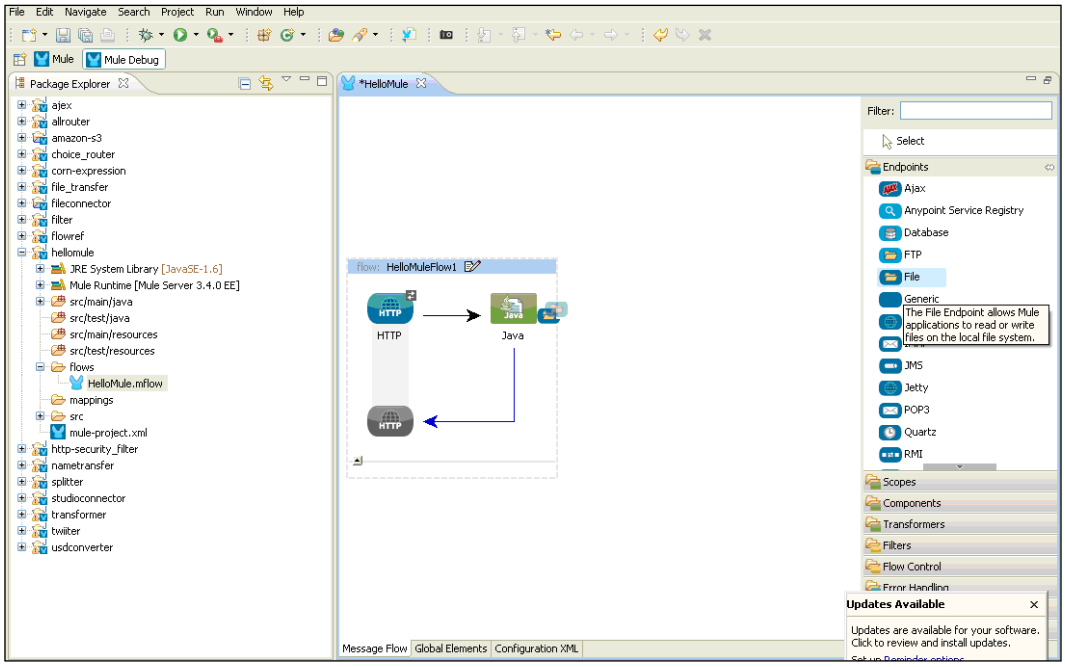
Getting ready

There are three types of components in Mule Studio:

- Simple component
- Java component
- Other components

How to do it...

Service components contain the business logic. Drag-and-drop a **Component** from the palette onto the canvas and configure the component. Double-click on the component.



How it works...

The following are the palette components present in Mule.

Palette components

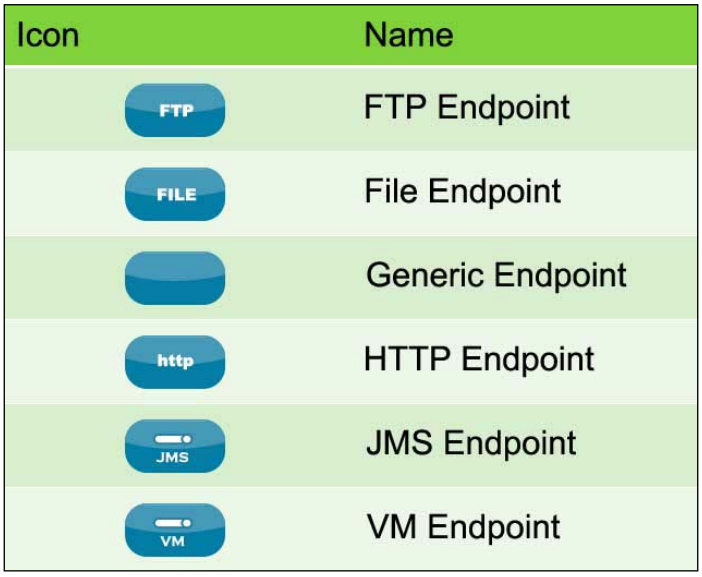
There are different palette components available in Mule Studio, where each palette component has different uses. We will see that in detail here:

- Endpoints
- Components
- Transformers
- Filters
- Flow control

- Routers
- Scopes
- Cloud connectors

Endpoints

Generally, Endpoints send and receive data, and are responsible for connecting to external resources and delivering messages. Endpoints can be Inbound or Outbound. An Inbound Endpoint receives messages via its associated transport. Each transport implements its own Inbound Endpoint element. An Outbound Endpoint sends messages via its associated transport. Each transport implements its own Outbound Endpoint element.



- **FTP Endpoint**: This Endpoint reads files from the FTP server. This Endpoint carries all the information for an FTP connection. The host and port values are required. The FTP Endpoint implements a file transport channel so that your Mule application can exchange files with an external FTP server. You can configure FTP as an Inbound Endpoint (which receives files) or Outbound Endpoint (which writes files to the FTP server). This is only used in the Enterprise edition.
- **File Endpoint**: This Endpoint reads a file from the filesystem. The File Endpoint implements a transport channel so that your Mule application can exchange files with a filesystem. You can implement the File Endpoint as an Inbound Endpoint (a message source), or as an Outbound Endpoint. This Endpoint implements only a one-way exchange pattern. The File Endpoint is used for transferring the file from one directory to another.

- **Generic Endpoint**: This Endpoint is used as a dynamic way to configure an Endpoint using Mule expressions and specifying paths. The Generic Endpoint allows for a wide array of configuration options by defining a particular transport to be used as the Endpoint.
- **HTTP Endpoint**: This Endpoint is used to process HTTP requests or responses. Mule uses HTTP Endpoints to send and receive requests over the HTTP transport protocol, or HTTPS over the SSL protocol. Configured as either Inbound (also known as message sources) or Outbound, HTTP Endpoints use one of the two patterns: request-response and/or one-way.
- **JMS Endpoint**: This Endpoint is used to send or receive messages from a JMS queue. The JMS Endpoint's two-way exchange patterns use: request-response and/or one-way.
- **VM Endpoint**: This Endpoint is used for an in-memory queue that allows you to integrate different flows or services in the same Mule configuration. The VM Endpoint's two-way exchange patterns use request-response and/or one-way.

Components

The Studio building blocks are known as components and fall into three categories: general, script, and web service.

General components execute whenever a message is received. The logic embedded into general components cannot be modified. Components such as Logger, Flow Reference, and Echo fall into this category.

Script components do not contain prepackaged logic; instead they allow the developer to specify the logic (in the form of a custom script or a Java class) to add into the component. Script components also allow you to:

- Configure interceptors
- Add Spring beans
- Change the value or reference of a specific property within the associated class

The Java component allows you to reference a Java class. The other script components support the Groovy, JavaScript, Python, and Ruby scripting engines.

Web service components, as the name implies, enable Mule to use SOAP and RESTful protocols to communicate with external web services. The SOAP and RESTful components use CXF and Jersey services to convert messages from Java to XML. Web service components also allow the developer to select or define the logic to be invoked by the component. If using the RESTful component, you only need to select a Java class and add a script to the component. On the other hand, SOAP configuration requires you to define attributes and select the operation method used to publish a SOAP web service.

As an example of how the SOAP component can be used, a SOAP message could be sent to a web-service-enabled website, such as a used car price database, with the parameters needed for a search. The site would then return an XML-formatted document with the resulting data; for example, prices, models, and features. The data returned is then integrated directly into a third-party website or application.

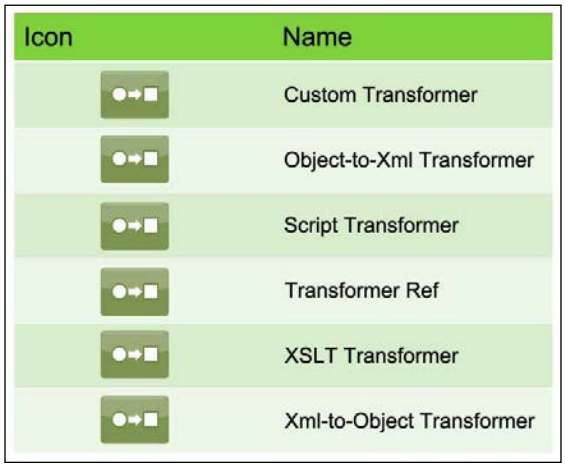
The following components are present in Mule:

- **Echo**: This component is used to echo a message payload to the console.
- **Logger**: This component is used to perform logging using an expression that determines what should be logged depending on the logging level. Use Logger to log messages, such as error messages or exceptions.
- **REST**: This component is used to make a REST service available via Jersey. REST is the formalized architecture of HTTP based on the concepts of resources, links, and a uniform interface. It uses the HTTP protocol. We can create a web service using the REST component.
- **SOAP**: This component is used to make a web service available via CXF. You can create a CXF web service in Studio by configuring a SOAP component in your Mule flow to perform any of the following CXF web service operations:
  - Publish a simple service
  - Publish a JAX-WS service
  - Proxy a published service
  - Consume a service using a simple client
  - Consume a service using JAX-WS client
  - Proxy to a service

  Using Mule's SOAP component, you can also enable WS-security, specify data bindings, and add interceptors to your CXF web service.

Transformers

Transformers convert message payloads to formats expected by their destinations. Mule ESB provides many standard transformers, which you configure using predefined elements and attributes in your Mule XML configuration file. You can also configure custom transformers using the `<custom-transformer>` element. You can configure a transformer locally or globally.



The following transformers are present in Mule:

- **Custom Transformer**: Transformers in Mule are used to convert messages from one format to another or to manipulate the message information such as headers and attachments. Mule ESB also provides several standard transformers, including XML transformers. You can create your own custom Java class using the extended `AbstractTransformer` interface. Two ways to create a Custom Transformer are:
  - Use a transformer annotation on a method
  - Create a custom Java class
- **Object-to-Xml transformer**: This transformer is used to convert a Java object to an XML representation using XStream. You configure this transformer using the `<object-to-xml-transformer>` element.
- **Script Transformer**: This transformer is used to transform the payload using a script. This defines script components to be used as transformers. The Script transformer lets you select the particular scripting engine from a pull-down list. The predefined script transformers, namely, Groovy, JavaScript, Python, and Ruby, have the scripting engines already set.
- **Transformer Ref**: This transformer is used to reference a global transformer.

- **XSLT Transformer**: This transformer is used to transform XML using XSLT.
- **Xml-to-Object Transformer**: This transformer is used to convert XML to Java bean graphs using XStream. You configure this transformer using the `<xml-to-object-transformer>` element.

Filters

Filters specify conditions that must be met for a message to be routed to a service or continued progressing through a flow. There are several standard filters that come with Mule ESB, which you can use, or you can create your own filters. You can create a global filter and then reference it from your services and flows. You can define a filter locally or globally.

Icon	Name
	Custom Filter
	Exception Filter
	Expression Filter
	Message Property Filter
	Filter Ref
	RegEX Filter
	Wildcard Filter
	Payload Filter

The following filters are present in Mule:

- **Custom Filter**: This filter is used as a user-implemented filter. The standard filters handle most filtering requirements, but you can create your own custom filter. To create a custom filter, you have to implement the `Filter` interface.
- **Exception Filter**: This is a filter that matches an exception type.
- **Expression Filter**: This filter evaluates a range of expressions providing different types of evaluators such as XPath, JXPath, and OGNL and also a custom evaluator.
- **Message Property Filter**: This filter allows you to add logic to your routers based on the value of one or more properties of a message. This filter can be very powerful because the message properties are exposed, allowing you to reference any transport-specific or user-defined properties.

- **Filter Reference**: This filter is used to reference to a globally defined filter.
- **Regular Expression Filter**: This filter is used on a filter that applies a regular expression pattern to the message payload.
- **Wildcard Filter**: This is a filter that matches string messages against wildcards.
- **Payload Filter**: This is a filter that checks the class type of the payload object inside a message.

Routers

Flow Controls/Routers route messages to various destinations in a Mule flow. Some Flow Controls incorporate logic to analyze and possibly transform messages before routing takes place.

Icon	Name	Description
	All	Router that sends message to all routes.
	Choice	Router that routes messages based on expressions.

The following Routers are present in Mule:

- **All**: This Router can be used to send the same message to multiple targets. It sends messages to all routes.
- **Choice**: The Router sends a message to the first message processor that matches. It routes messages based on expressions.

Cloud Connectors

A Cloud Connector easily integrates your Mule application with third-party web APIs. A Cloud Connector is built using the Cloud connect toolset, which resides in Mule Studio by default.

Icon	Name
	Salesforce
	Twitter

The following Cloud Connectors are present in Mule:

- **Salesforce**: This connector provides an easy way to integrate with the Salesforce API. This allows users to create flows which can query, create, and update information in Salesforce.
- **Twitter**: This connector provides an easy way to integrate with the Twitter API using Mule flows.

Deploying your first Hello World application on the Mule server

By creating a simple Hello World application, you will know how to create a flow and deploy the flow using the Mule server.

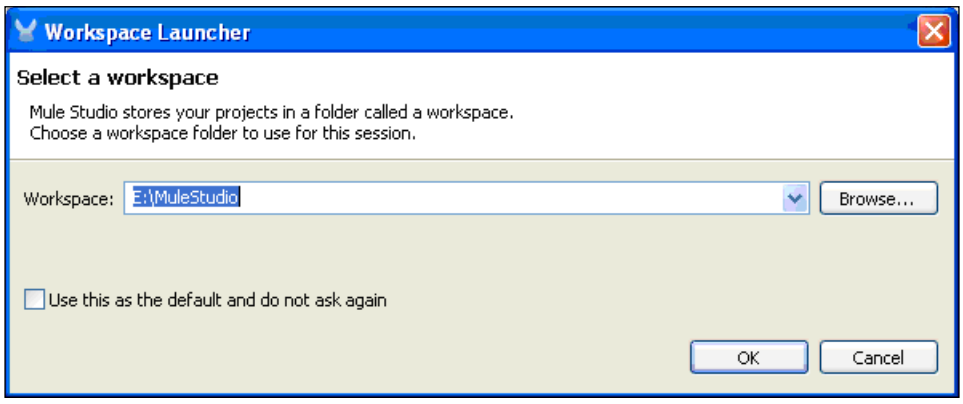
Getting ready

Using the steps for application deployment, given in this recipe, you will learn the execution of the flow, how that flow execution will occur, and what will be the output of the application code.

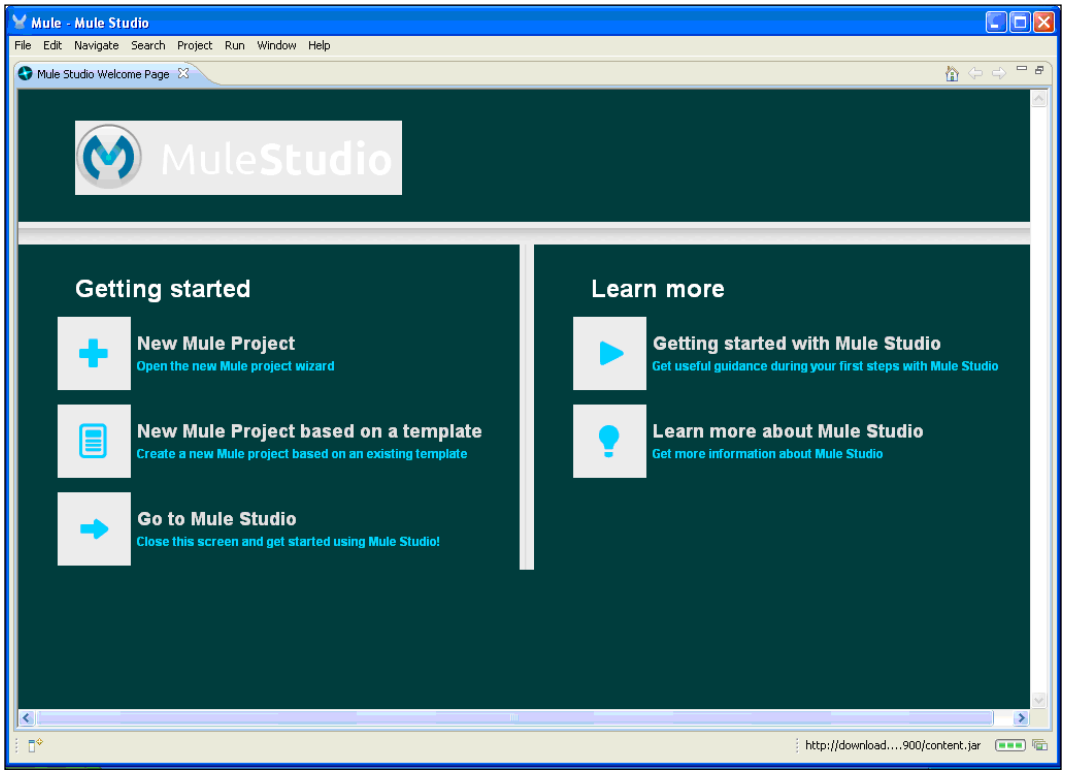
How to do it...

In this example you will see how to create and deploy the first "hello world" using Mule Studio.

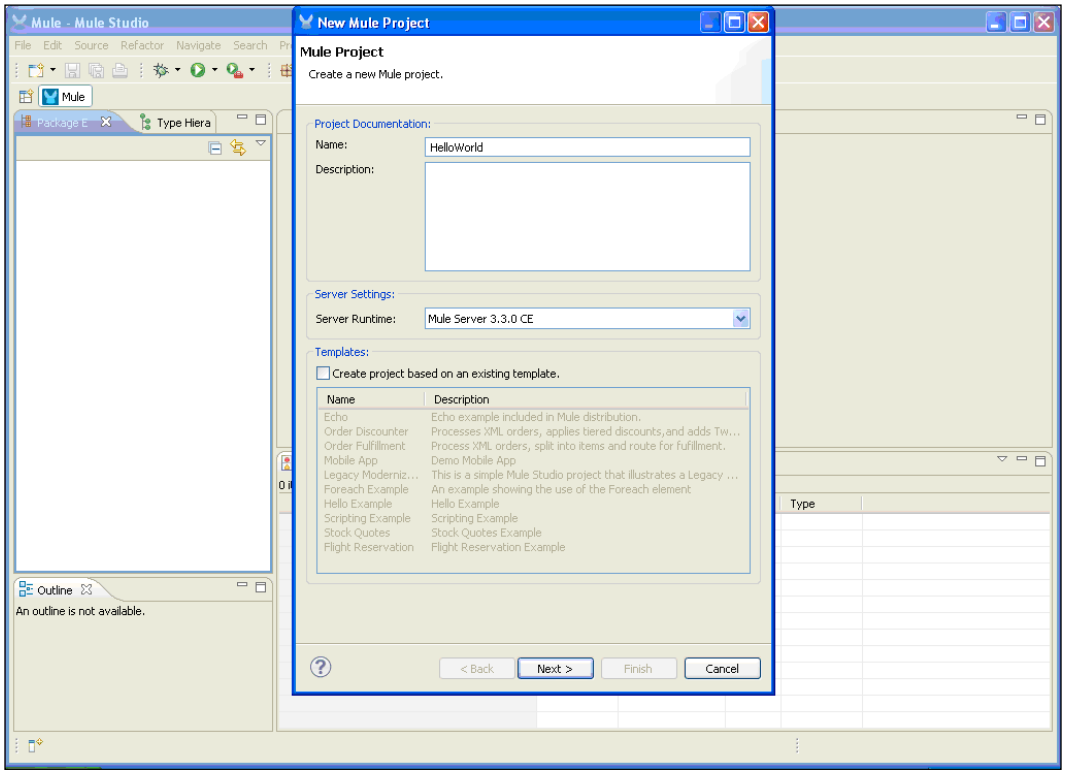
1. Open Mule Studio and enter the name for the workspace name, as shown in the following screenshot:



2. Create a project in Mule Studio. You will see the **Mule Studio Welcome Page** window. If you click on **New Mule Project Based on a template**, you will see an existing example. By clicking on **Go to Mule Studio**, it will start redirecting away from the **Mule Studio Welcome Page** window. You can even create a project from a menu bar by going to **File** | **New** | **Mule Project**.

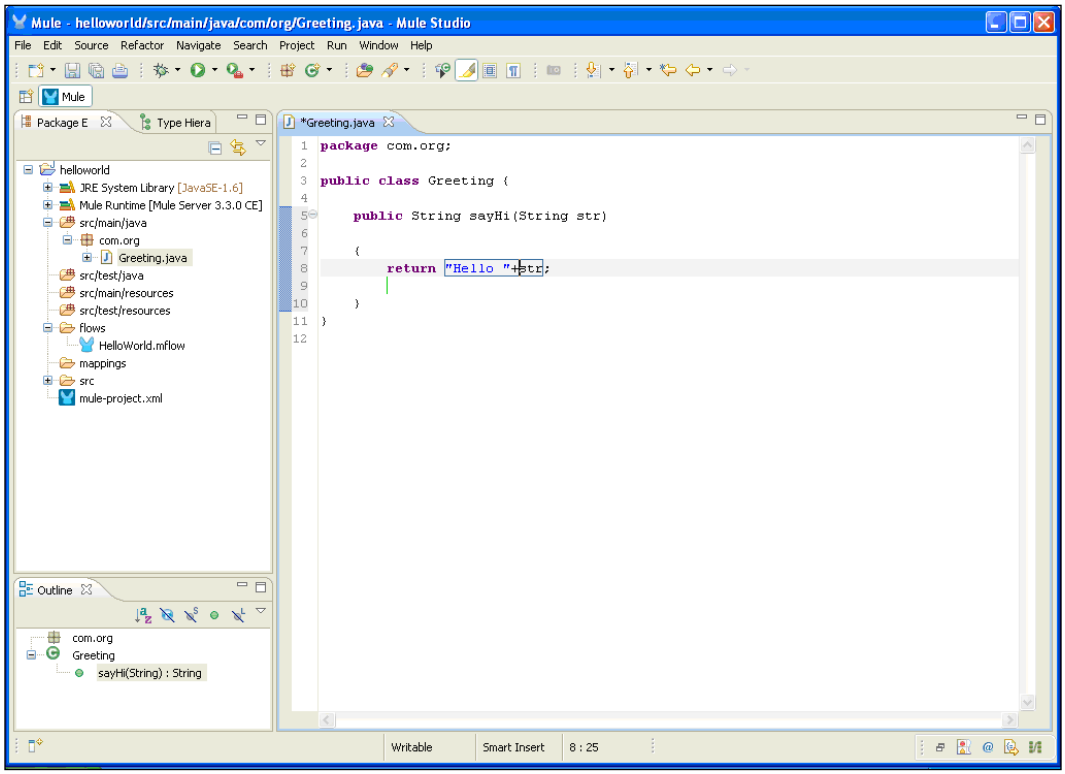


3. Enter the project name called `HelloWorld`, click on **Next**, and enter the `.mflow` name as the filename. Then, click on the **Finish** button.



4. Go to `src/main/java`, right-click on it, and go to **New** | **Class**. Create a class called `Greeting` under the package `com.org`; here we have created the `sayHi` method and its return type is set to `String`.

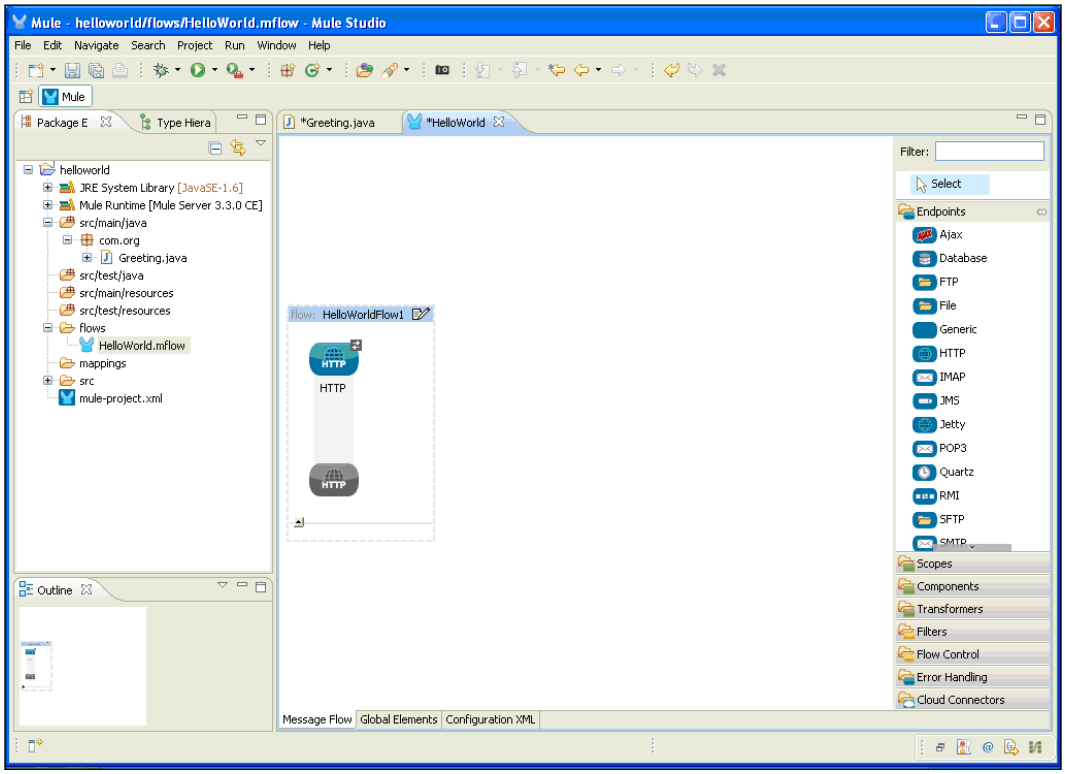
```
public String sayHi(String str)
  {
    return "Hello "+str;
  }
```



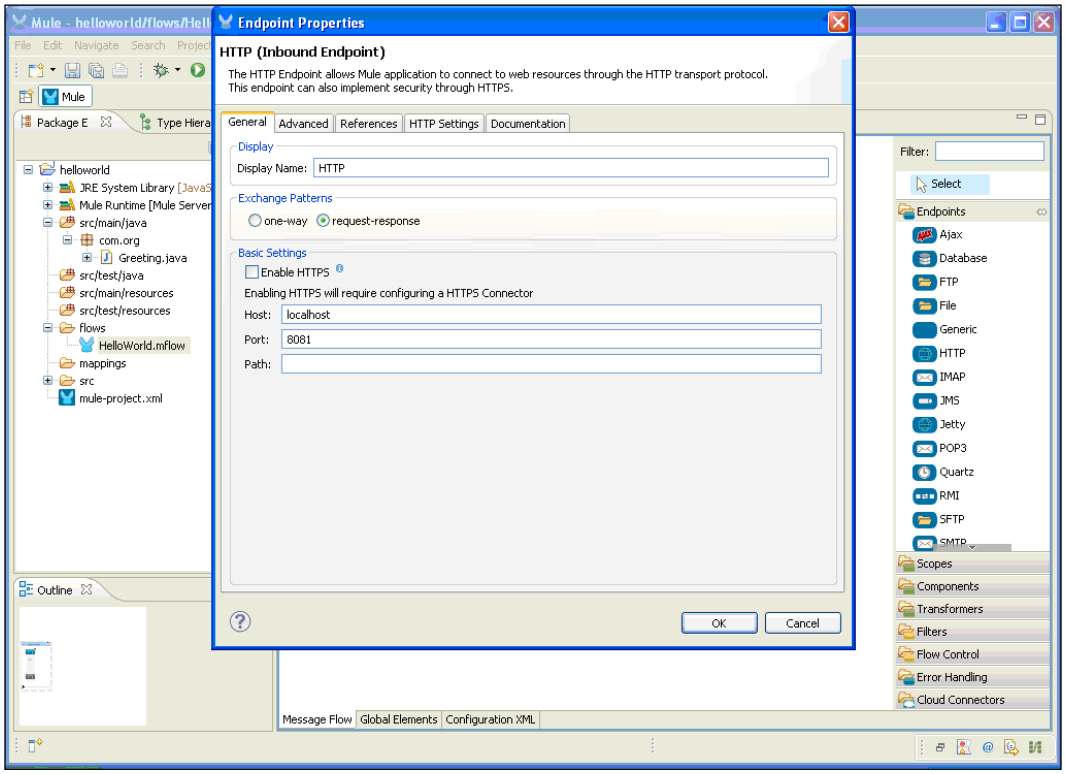
Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

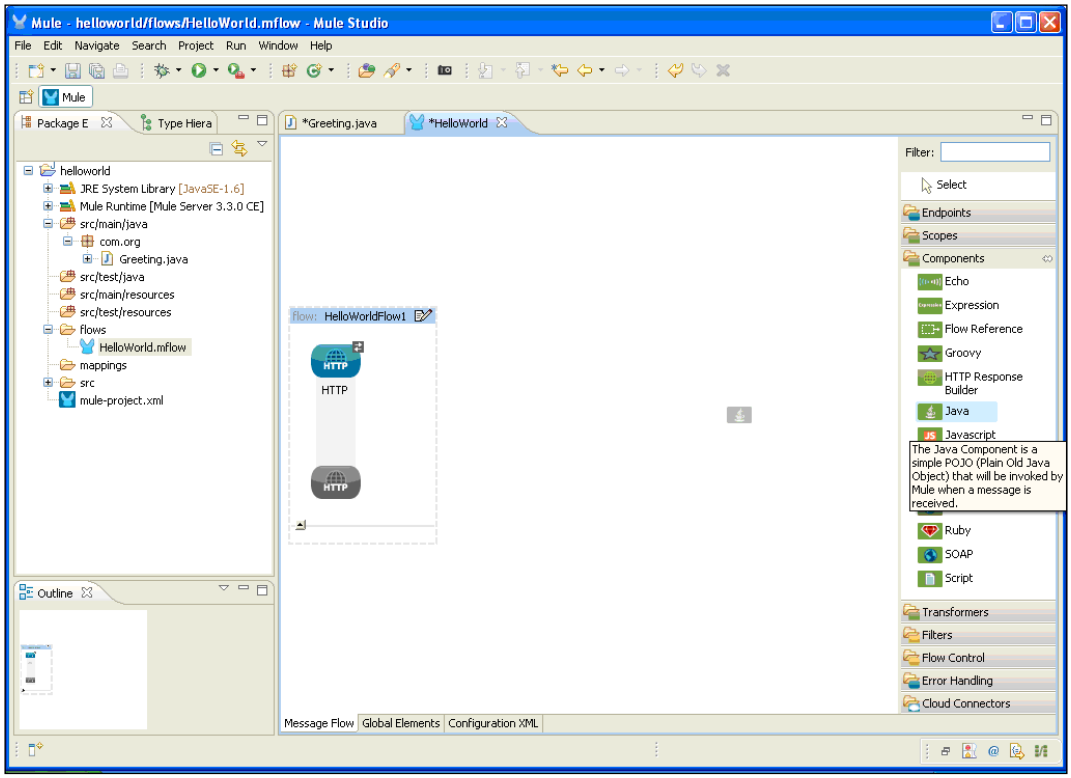
5. Go to the `Greeting.Mflow` file. In the following screenshot, the central part is called the canvas where we can put graphical elements and on the right-hand side you can see the group of elements; this area is called a palette. You have to drag-and-drop the **HTTP** Endpoint from the palette onto the canvas area.



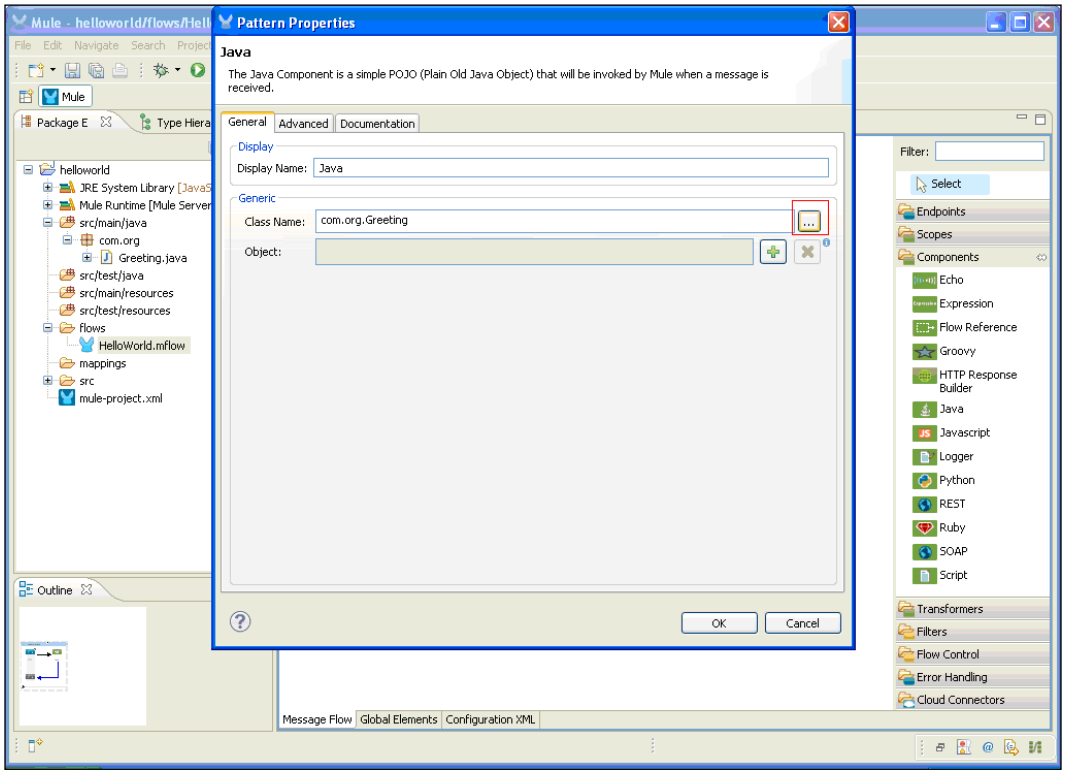
6. Double-click on the **HTTP** Endpoint to configure it. You will see the hostname. If you want to change the hostname, you can change it. In this example, we use `localhost`. If you want to change the port number, you can change that as well. By default, port number **8081** will be taken by the Mule server.



7. Drag-and-drop the **Java** component from the palette onto the canvas area. In the **Java** component, we will store the custom Java class. Configure the **Java** component. In this example, we configure the `Greeting` class that we created before.



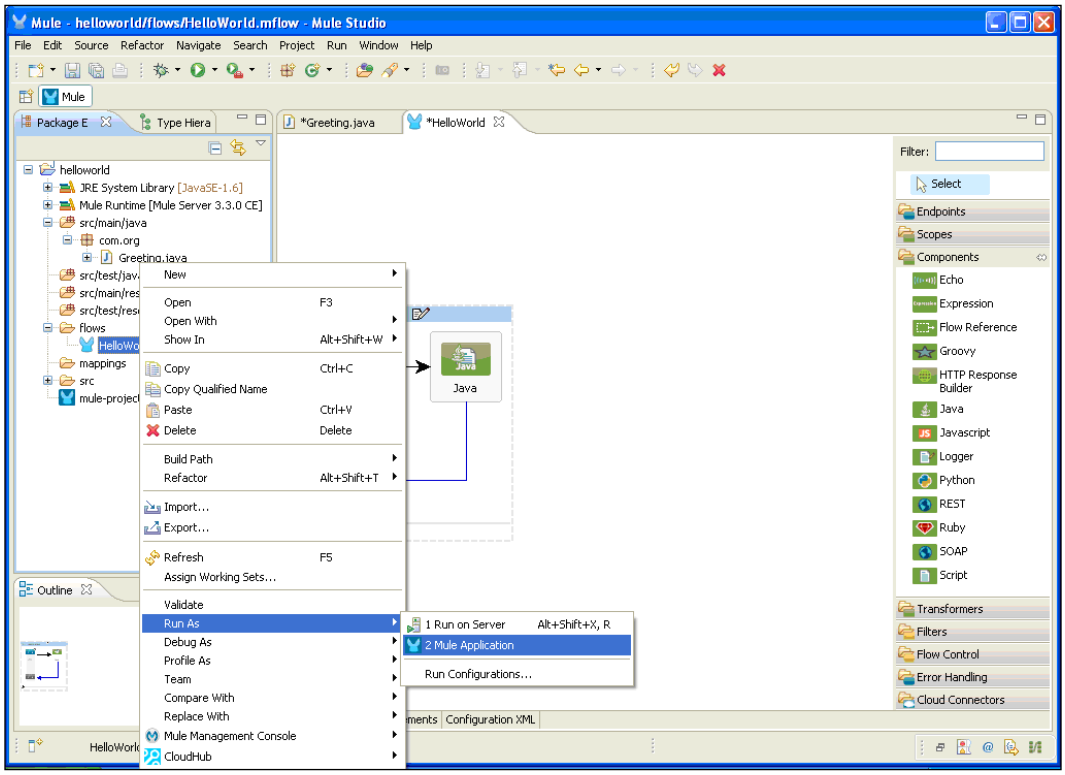
8. Double-click on the **Java** component and configure it. Click on the Java icon and configure the `Greeting` class.



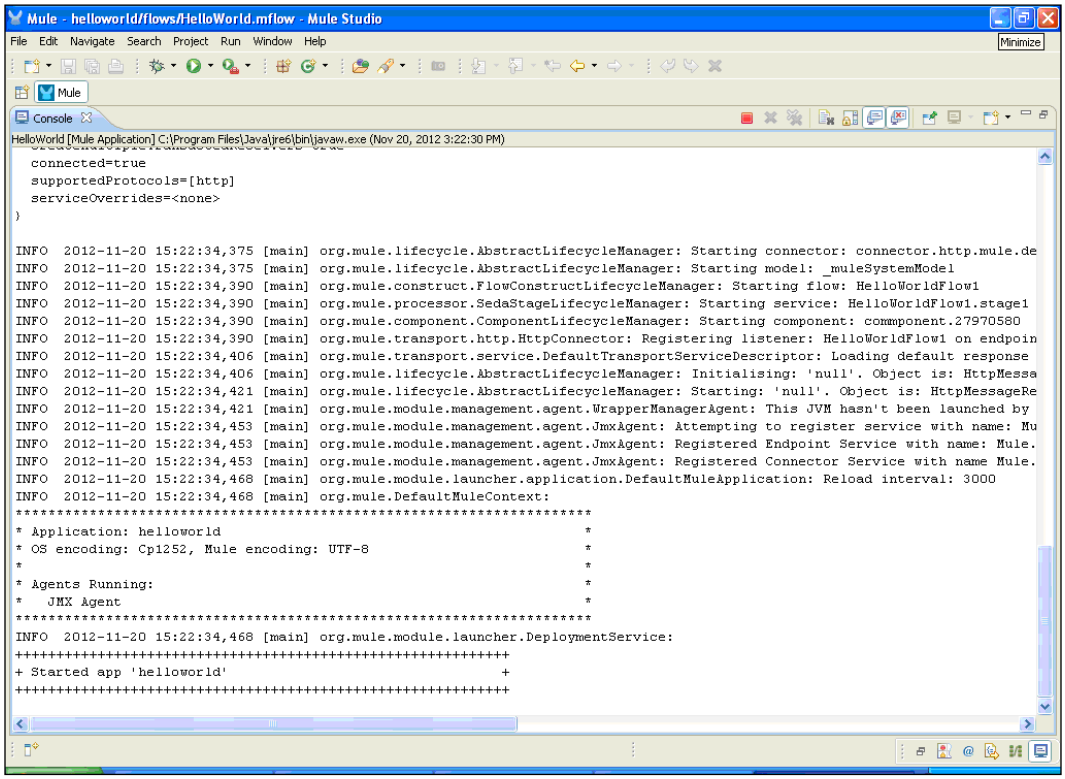
How it works...

In this section, you will see how to deploy the application and how it will run on the browser.

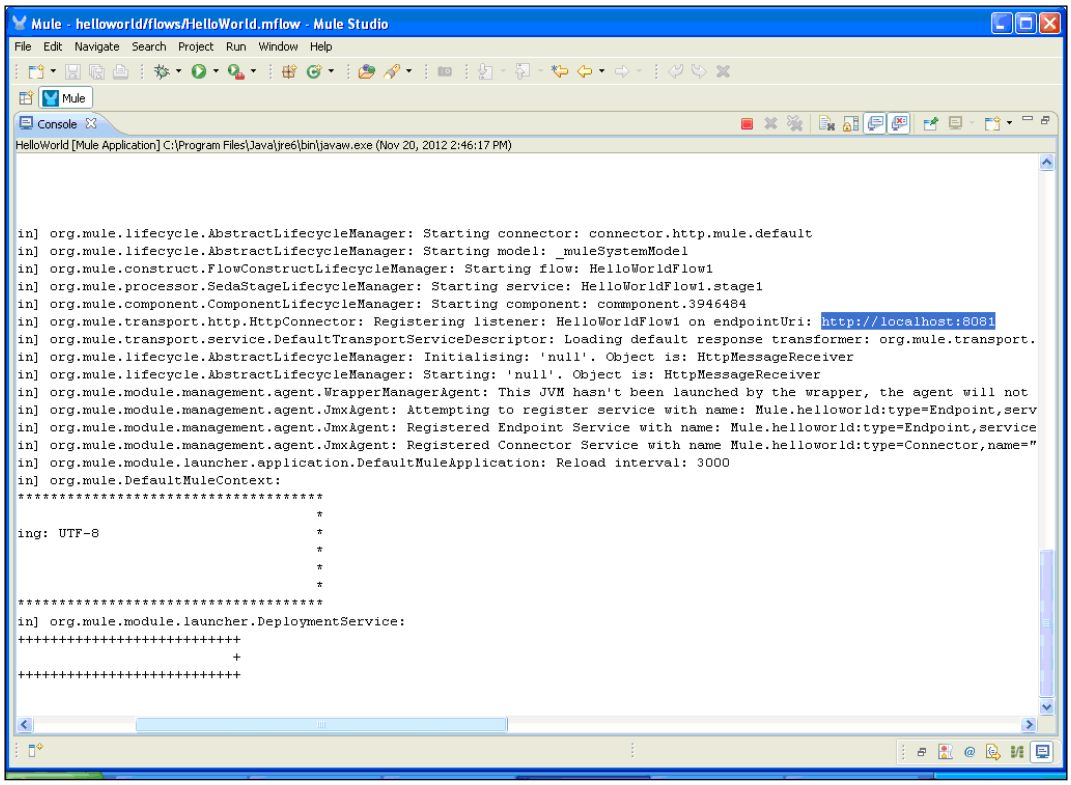
1. If you haven't saved your application code, do save it. To deploy the application code in the Mule server, go to **Run As** | **Mule Application**; the Mule server will deploy your application.



2. If your application code is successfully deployed, you will see the following output screen:



3. Copy the URL `http://localhost:8081` and paste it on your browser.



4. Paste the URL on your browser, and you will see the following output:

2
Working with Components and Patterns

In this chapter, we will cover:

- Configuring the component
- Using the Echo component to display the message payload
- Using a Flow Reference component to synchronously execute another flow
- Publishing a RESTful web service using the REST component
- Publishing a SOAP-based web service using the SOAP component

Introduction

Mule has the ability of routing, filtering, transforming, and processing with components. Each of those abilities are assigned a good number of fine-grained processors. The configuration file of a Mule application that combines those elements can end up being large. The different types of configuration patterns provided by Mule are simple service pattern, bridge, validator, HTTP proxy, and WS proxy.

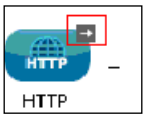
Configuring the component

In this recipe, you will see how to configure a component in Mule Studio. We will have a look at how to use different components throughout this chapter.

Getting ready

Mule uses HTTP Endpoints to send and receive requests over the HTTP protocol. Configured as either **Inbound** (also known as **message sources**) or **Outbound**, HTTP Endpoints use one of these two message exchange patterns: **request-response** or **one-way**.

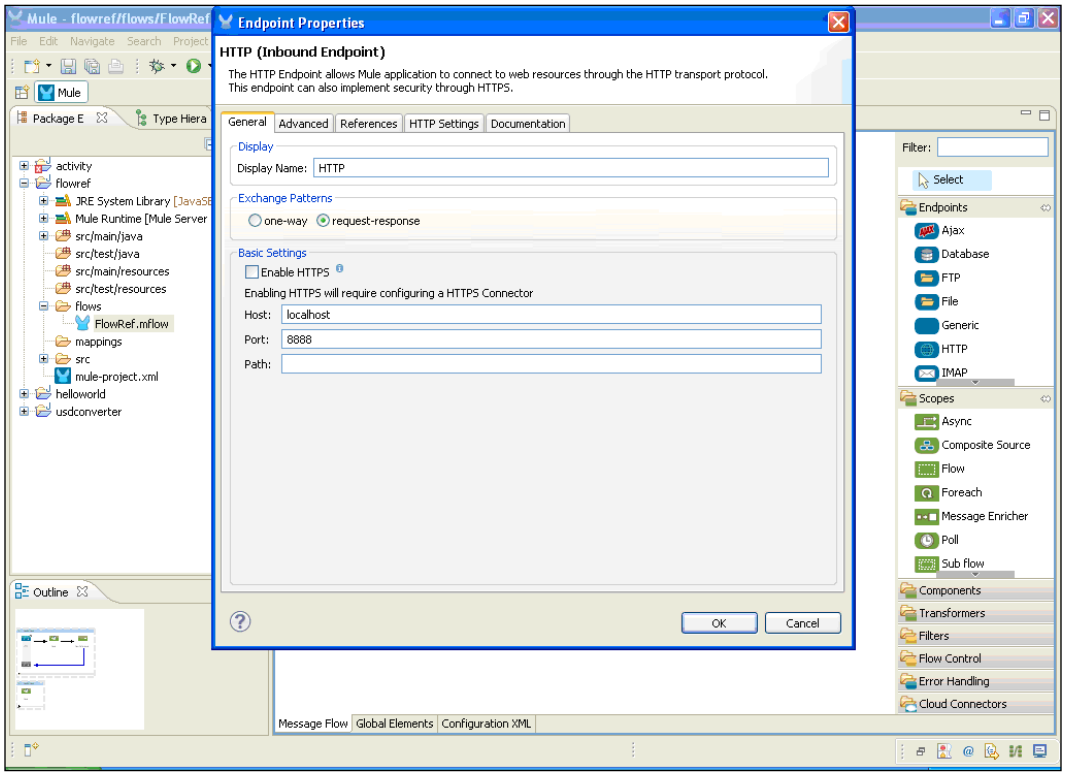
The arrows in the preceding screenshot indicate the request-response type of message exchange.



The arrow in this screenshot indicates the one-way type of message exchange.

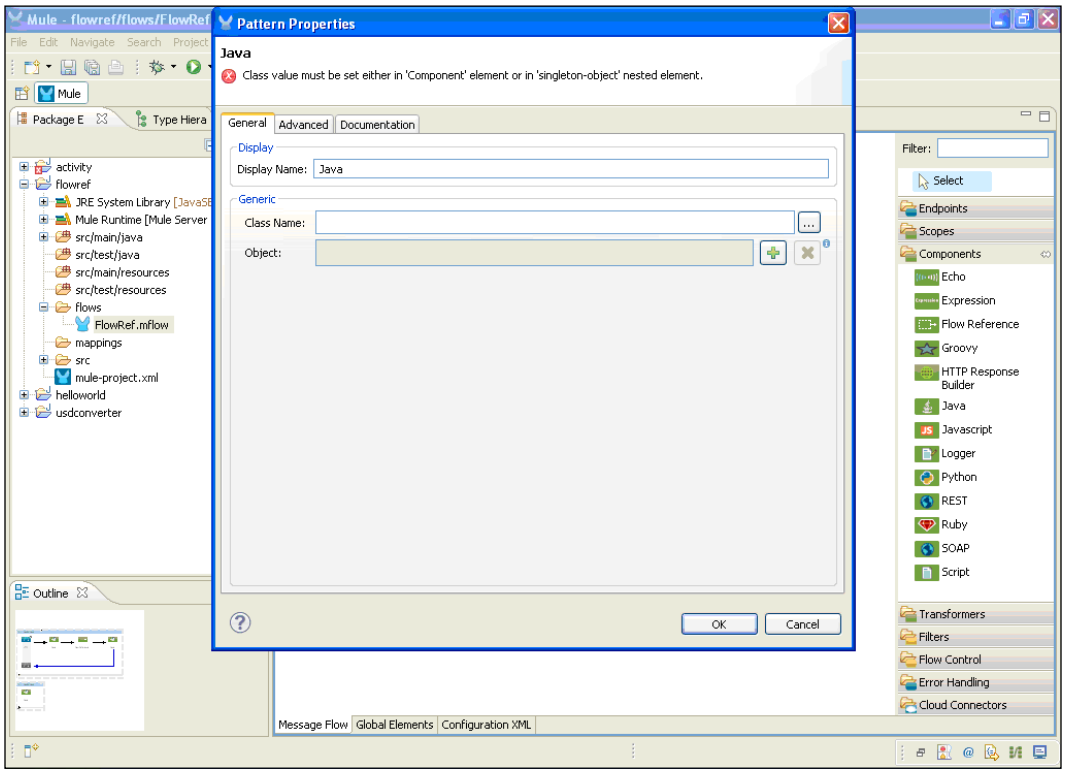
How to do it...

Double-click on the **HTTP** Endpoint to configure it. You will see a screen similar to the following screenshot on your window. You have to enter the **Host** and **Port** values. By default, the port number is **8081**; you can change the values of the **Host** and **Port** fields. However, these two fields are mandatory.



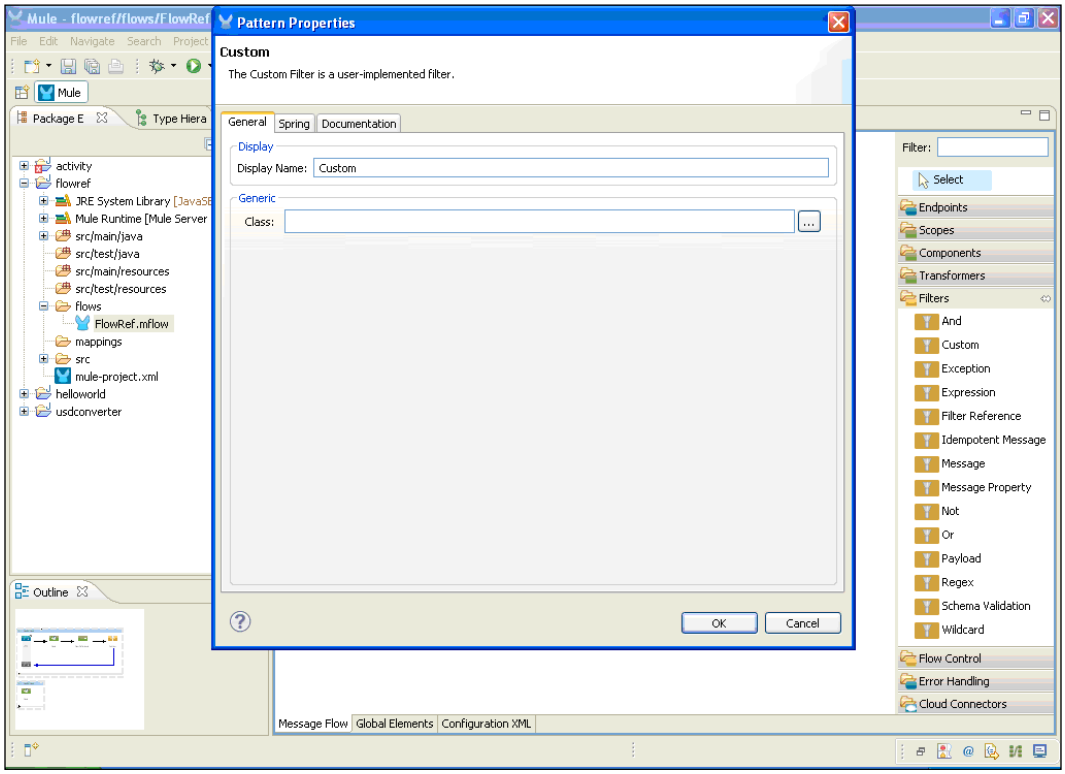
The Java component

Double-click on the **Java** component to configure it. You can import the class you had created.



Custom filters

Filters specify conditions that must be met for a message to be routed to a service. If the condition is met, the message will go to another component. You can also create your own filter. To create a filter, implement the `Filter` interface, which has a single method. You can import a custom filter class using the extended `Filter` interface.



How it works...

Components generally execute whenever a message is received; the logic embedded into components cannot be modified. Components such as **Logger**, **Echo**, **Java**, **Flow Ref**, and **Expression** all come under this category.

In the process of scripting a component, you have to develop your own business logic by writing a script, or you can import a script file written in scripting languages such as Ruby, Python, and Groovy. The Java component allows you to reference a Java class.

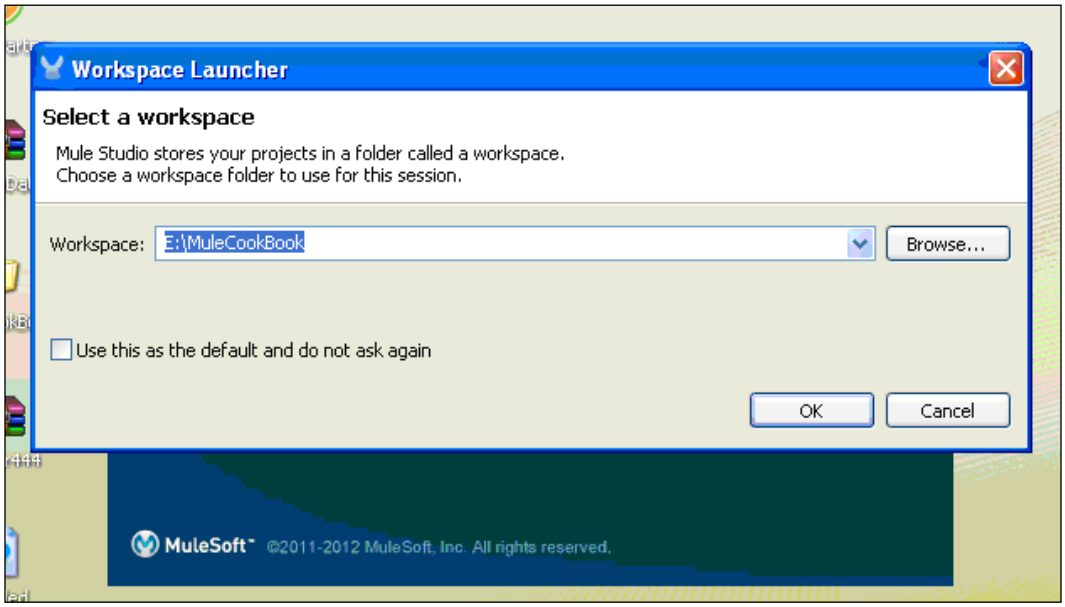
Using the Echo component to display the message payload

The **Echo** component is used to display the message payload. The Echo component is used for displaying message payloads, which receives the end user HTTP request and returns the payload message to the HTTP response, which is then sent to the end user. In this recipe, you will see how to use and configure the Echo component in Mule Studio.

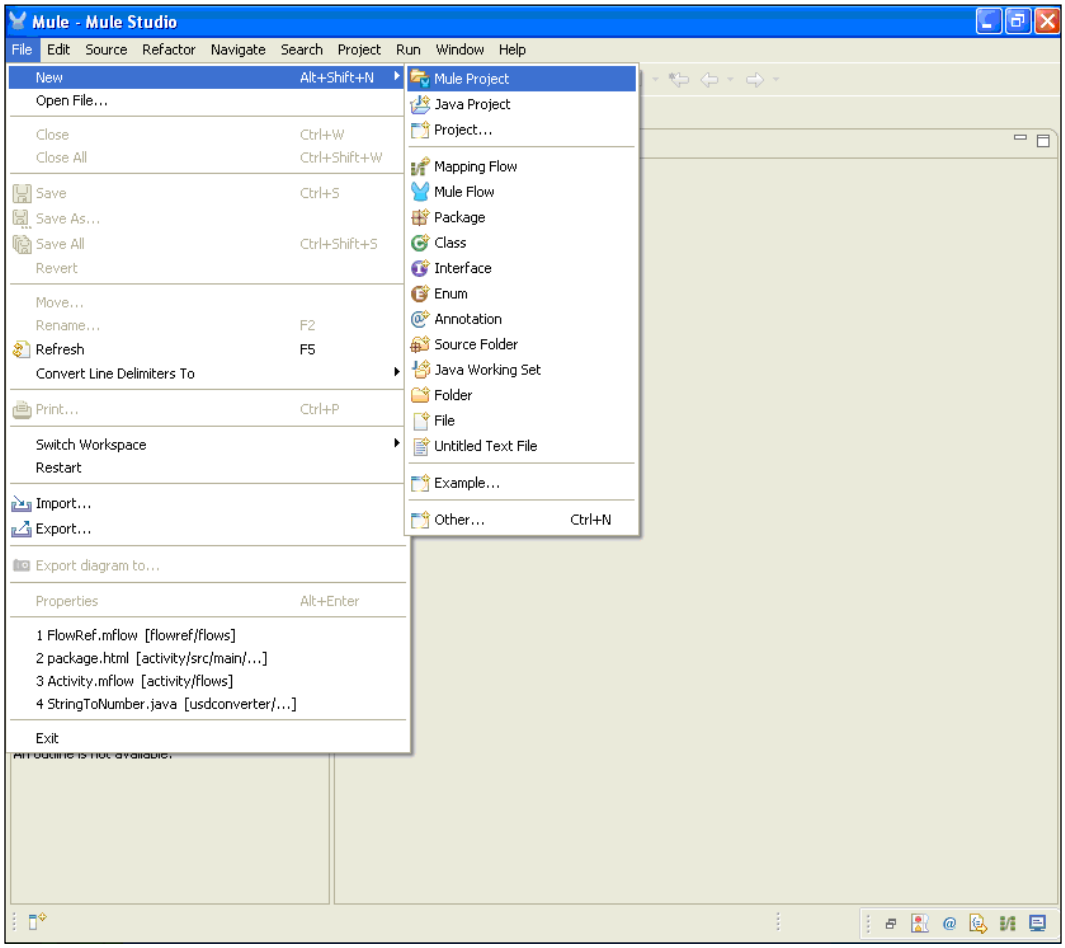
Getting ready

In this example, we'll use the following components: HTTP, Logger, and Echo.

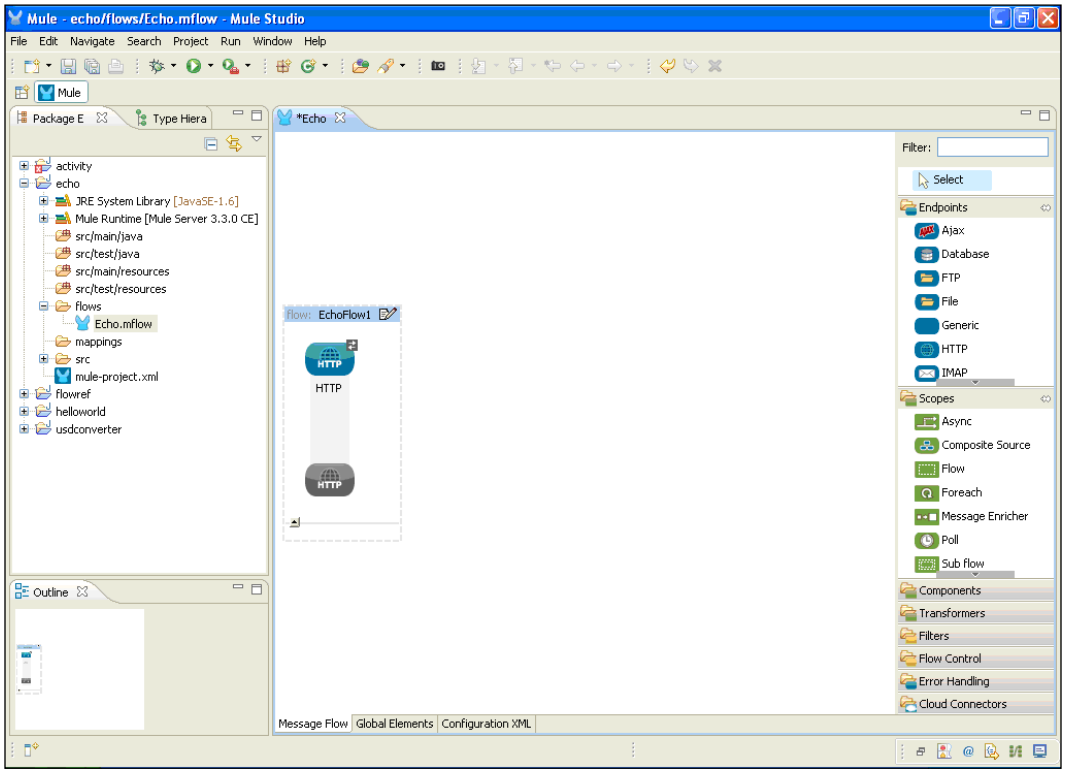
1. Open Mule Studio and enter the workspace name as shown in the following screenshot:



2. To create a new project, go to **File** | **New** | **Mule Project**. Enter the project name, `Echo`, and click on **Next** and then on **Finish**. Your new project is created. You can now start the implementation.



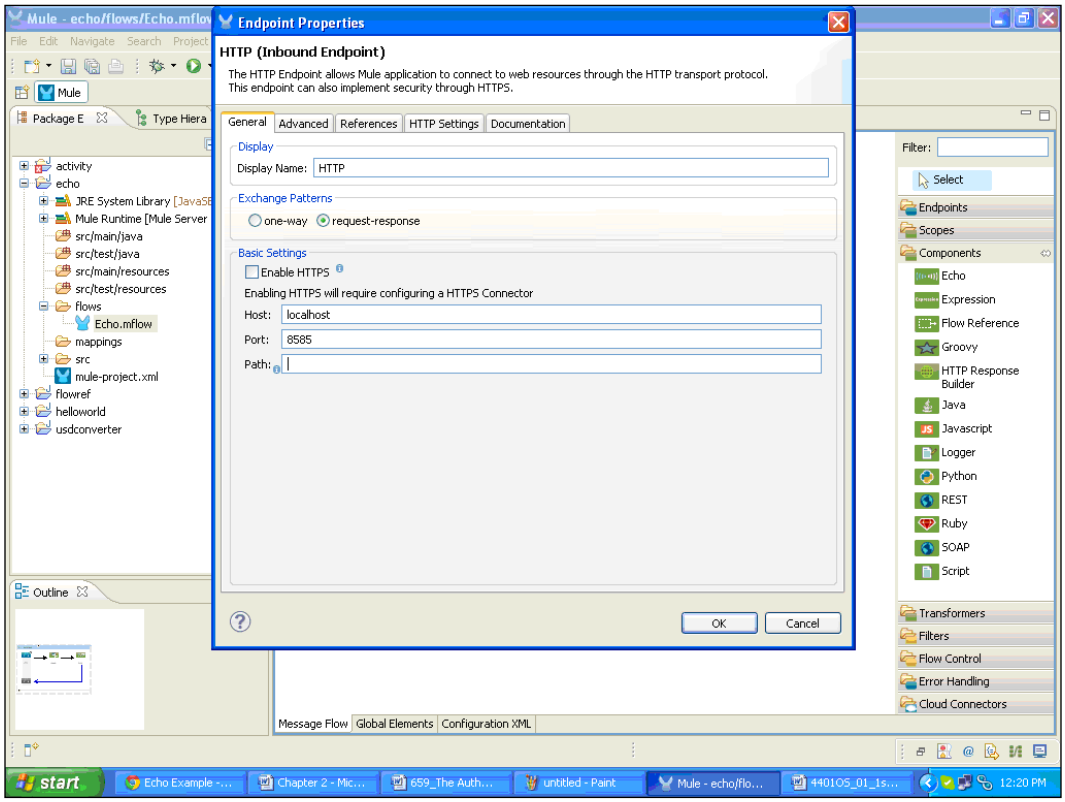
3. To create a flow, go to the `Echo.mflow` file, drag the **HTTP** Endpoint onto the canvas, and configure it by double-clicking on it.



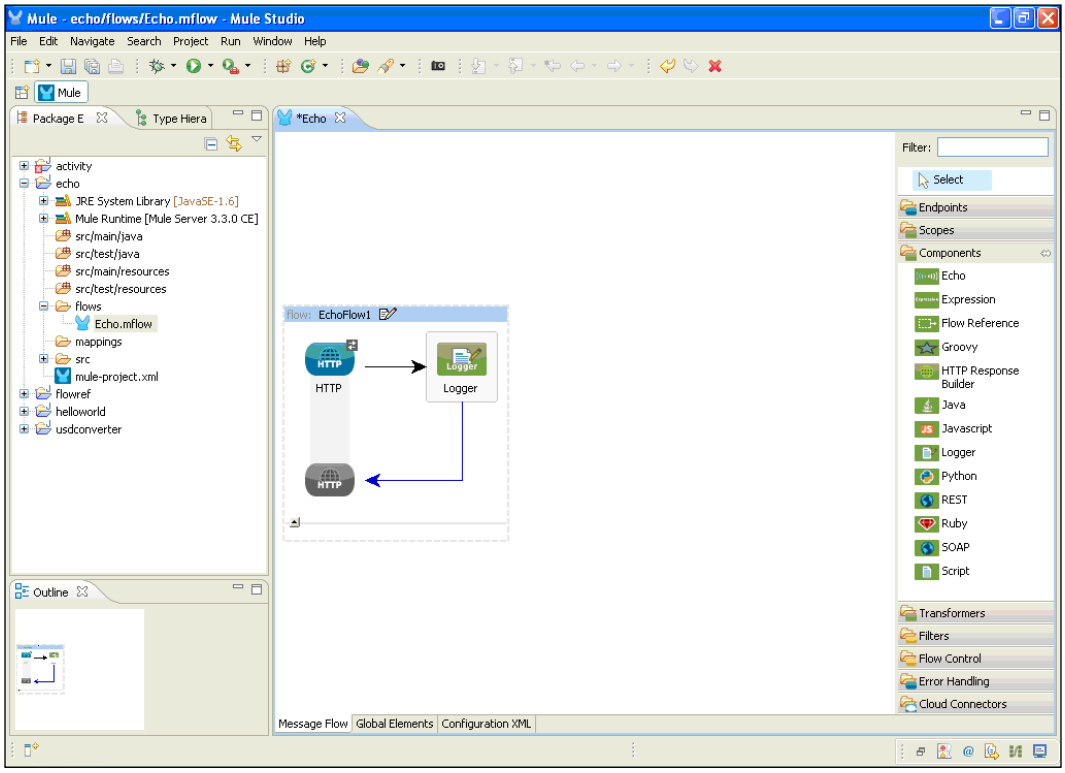
How to do it...

In this section, we will see how we can use the Logger and Echo components in a flow.

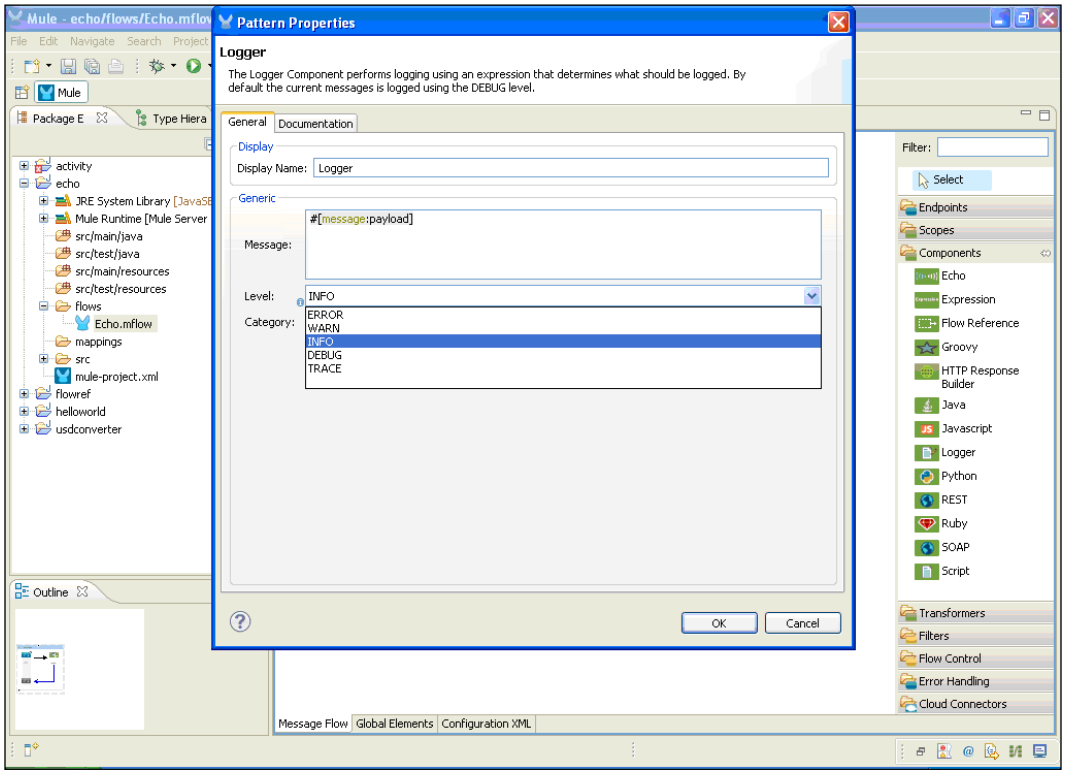
1. Double-click on the **HTTP** Endpoint to configure it. You can change the hostname and port number. Here, we have used port number `8585`.



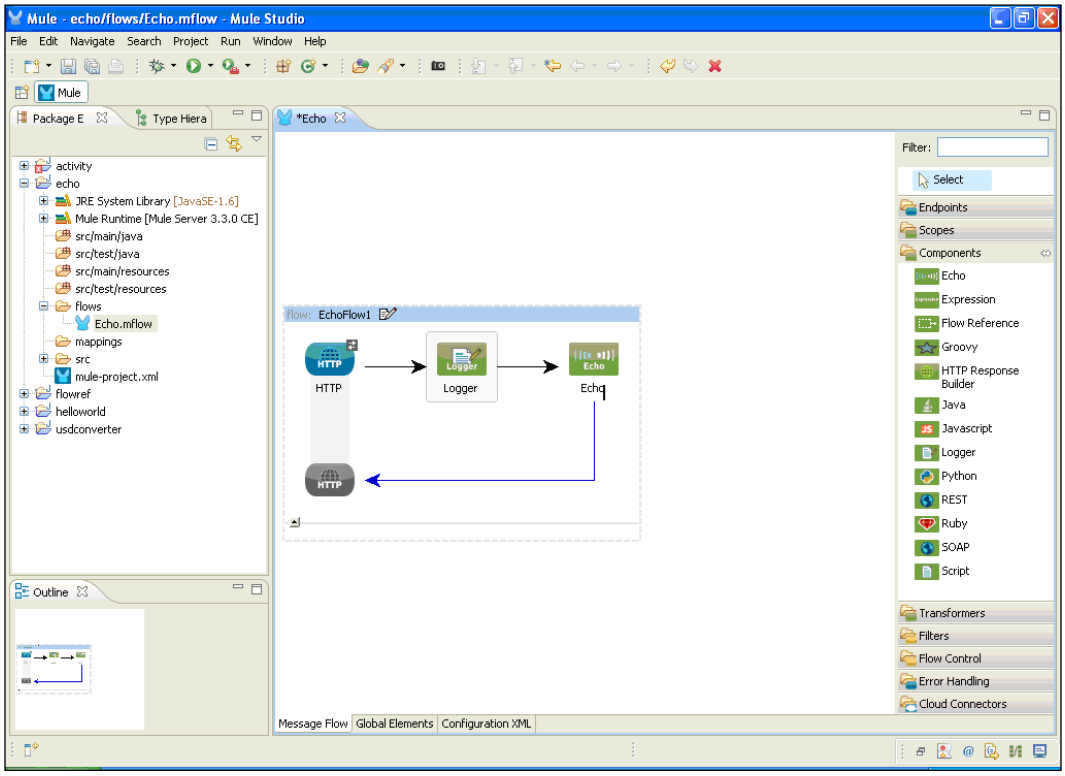
2. To display messages on the console, drag the **Logger** component onto the canvas and configure it. The Logger component uses an expression to determine what information in the message should be displayed on the console. **Mule Expression Language** (**MEL**) is the primary language used for formulating such expressions throughout the Mule ESB.



3. Double-click on the **Logger** component to configure it. You can see different types of levels. The Logger component level is used for displaying error messages or exceptions. We have selected the **INFO** level here. In the message box, you should use the expression `#[message:payload]`. This expression is used for displaying messages on the console.



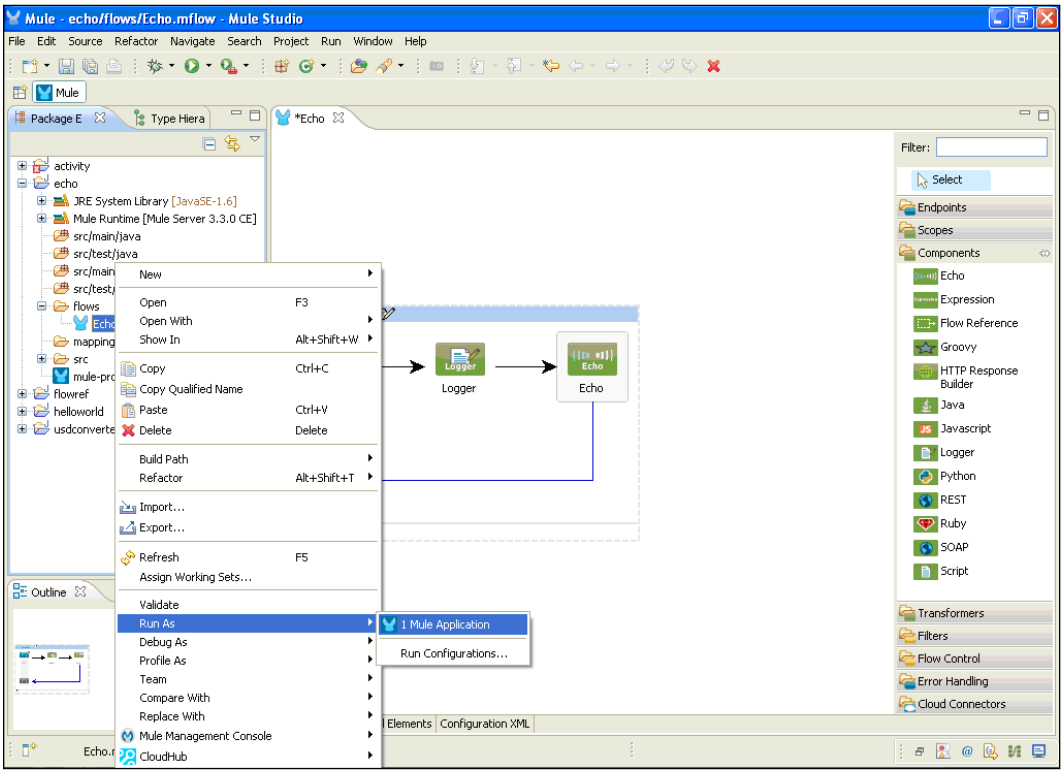
4. Drag the **Echo** component onto the canvas; there is no need to configure it. Messages sent to an **Echo** component simply return the message payload as the response to an end user.



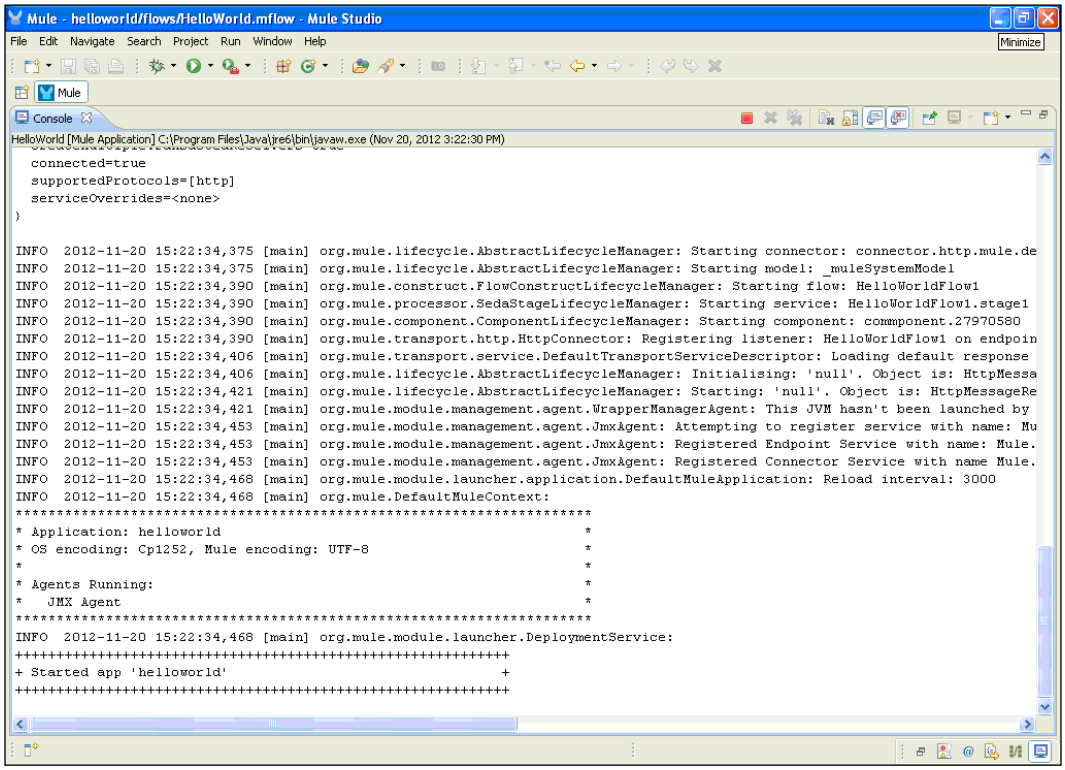
How it works...

In this section, you will see how to deploy the application and how to run the application on the browser.

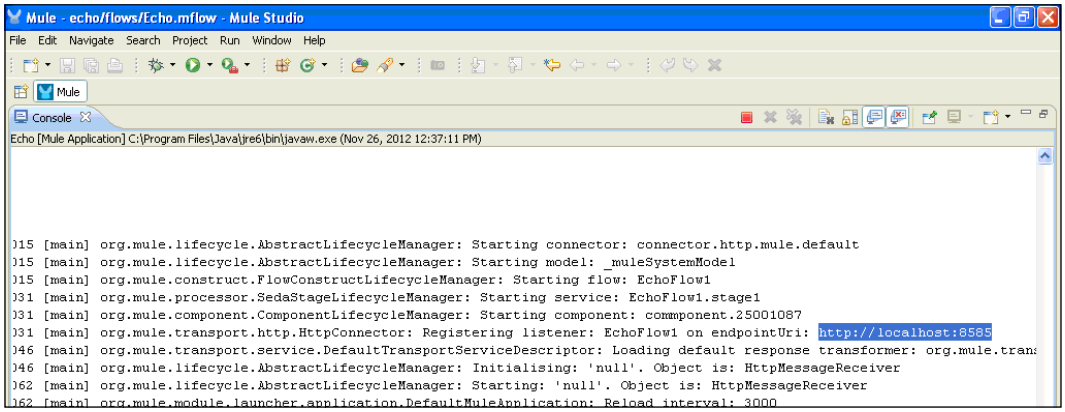
1. Now we are ready for the deployment. If you haven't saved your application code, do save it. After saving your project, right-click on the `Echo.mflow` file and go to **Run As | Mule Application**.



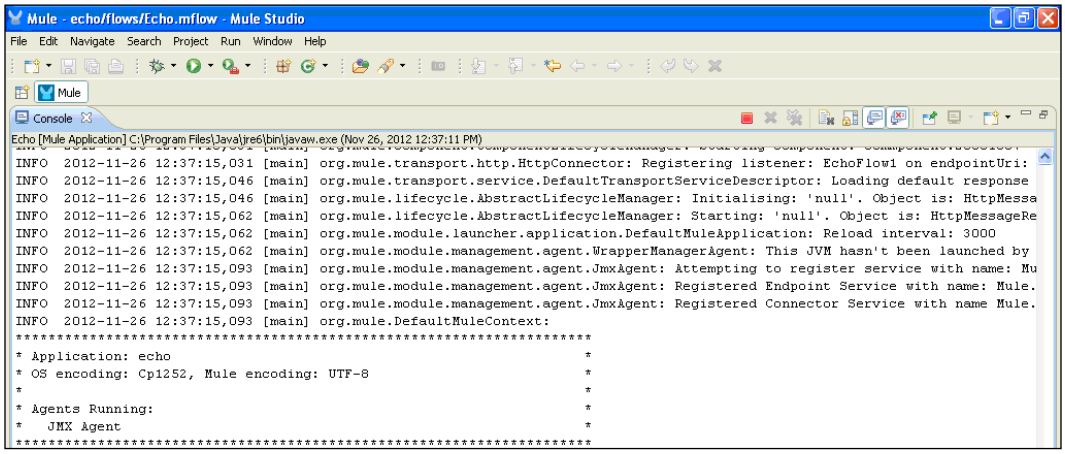
2. If your application code is successfully deployed, you will see the following message on the console: `Started app 'helloworld'`.



3. Copy the URL `http://localhost:8585` and paste it in your browser.



4. To see the output on the console, paste the URL in your browser and type in `/EchoExample`. When a user types `http://localhost:8585/EchoExample` in the browser, Mule returns a message in the browser that reads `/EchoExample`, as shown in the following screenshot:



Using the command prompt

To run a Mule application, enter the following command on the command prompt:

```
mule [-config <your-config.xml>]
```

Here, `<your-config.xml>` is the Mule configuration file you want to use. If you don't specify the configuration file, Mule looks for `mule-config.xml`, which is a generic name that does not exist in the default configuration file. If you have only one configuration file, you can name it `mule-config.xml` so that you can run Mule with it just by typing in mule. To stop Mule, press *Ctrl* + *C*.

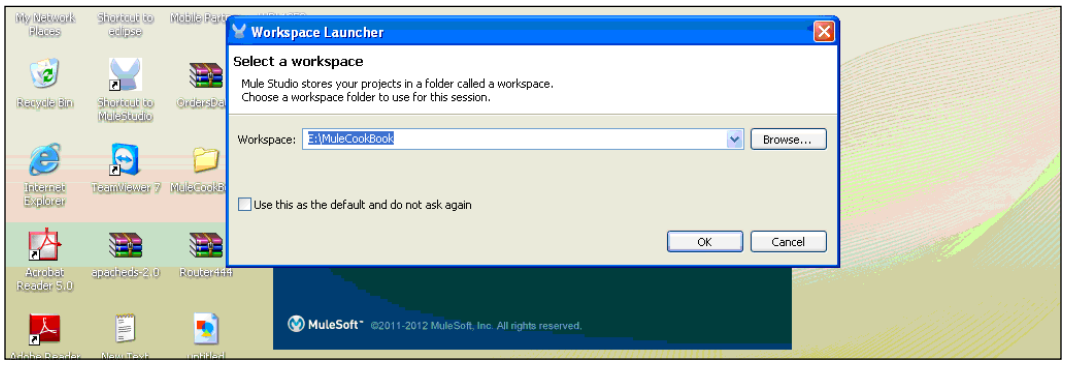
Using a Flow Reference component to synchronously execute another flow

Flow Reference is used to synchronously execute another flow that is external to the current flow. If a message reaches the Flow Reference component, Mule invokes the external flow referenced by it. Once the referenced flow completes, the control passes back to the initiating flow only after the external process is completed.

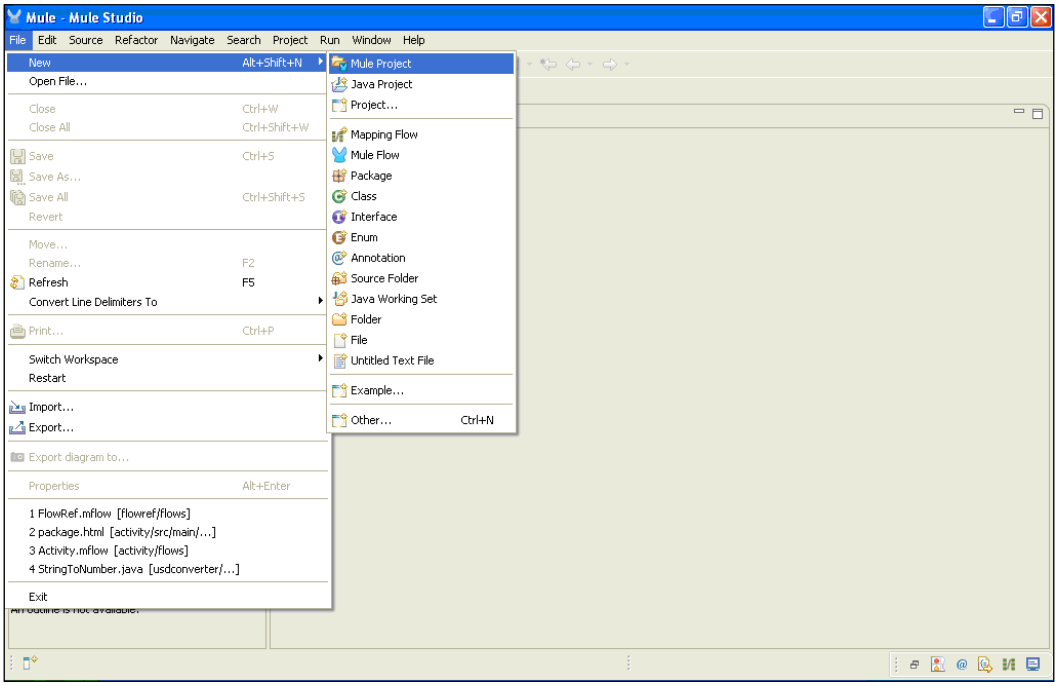
Getting ready

To demonstrate this example, we'll use the following four components: HTTP, Logger, Java, and Flow Ref.

1. Open Mule Studio and enter the workspace name as shown in the following screenshot:



2. To create a new project, go to **File** | **New** | **Mule Project**. Enter the project name, `FlowRef`, and click on **Next** and then on **Finish**. Your new project has been created now; so we are ready to start the implementation.



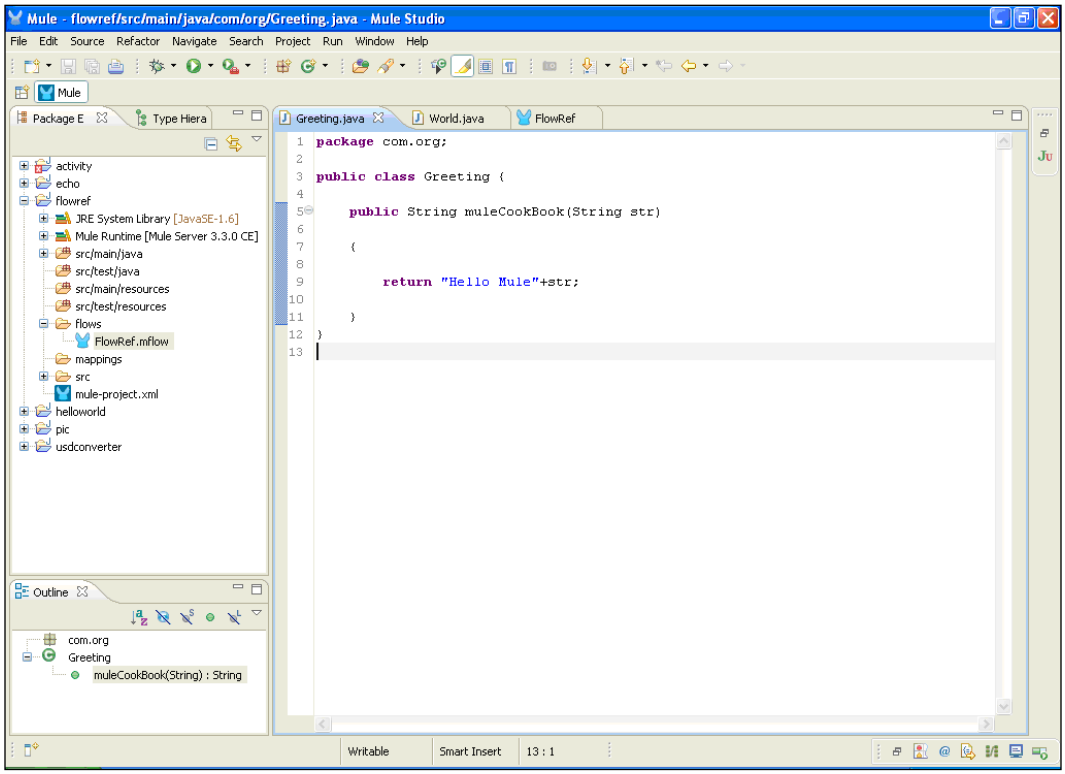
How to do it...

In this section, you will see how to configure the Java component and the Flow Ref component. Here you are creating a class, and the output will be displayed on the browser through this class.

1. To create a class, go to `src/main/java`, right-click on it, and go to **New** | **Class**. Create a class named `Greeting` under the package `com.org`; here, we create the `muleCookBook` method and its return type is set to `String`:

```
public String muleCookBook(String str)

  {
    return "HelloMule"+str;
  }
```

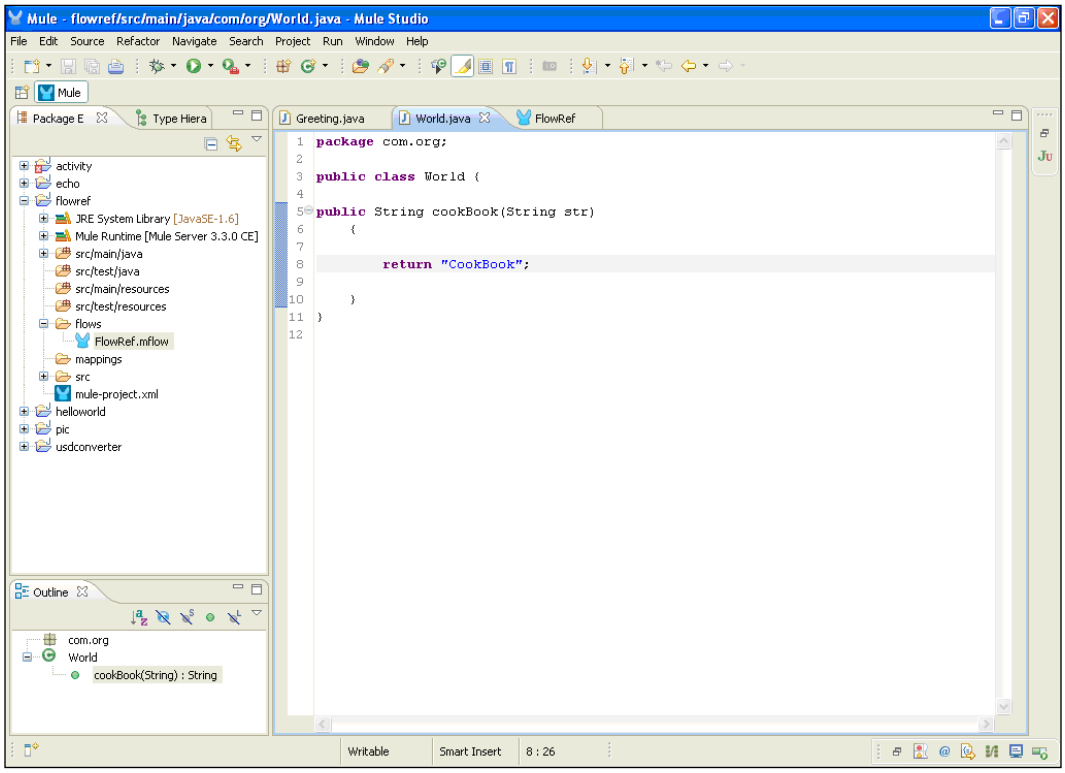
   You can see the creation of this method in the following screenshot:



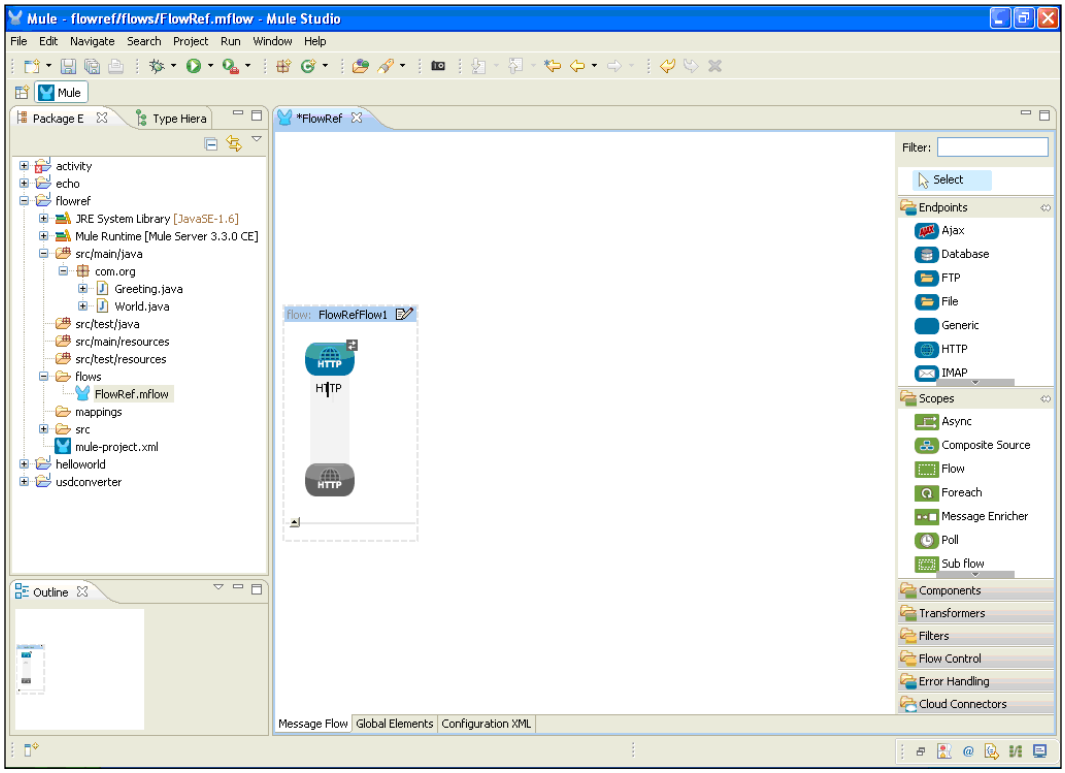
2. To create a class, right-click on the package. Create one more class, `World`, under the same package. Here, we create the method `cookbook` and its return type is set to `String`:

```
public String cookBook(String str)
  {
    return "CookBook";
  }
```

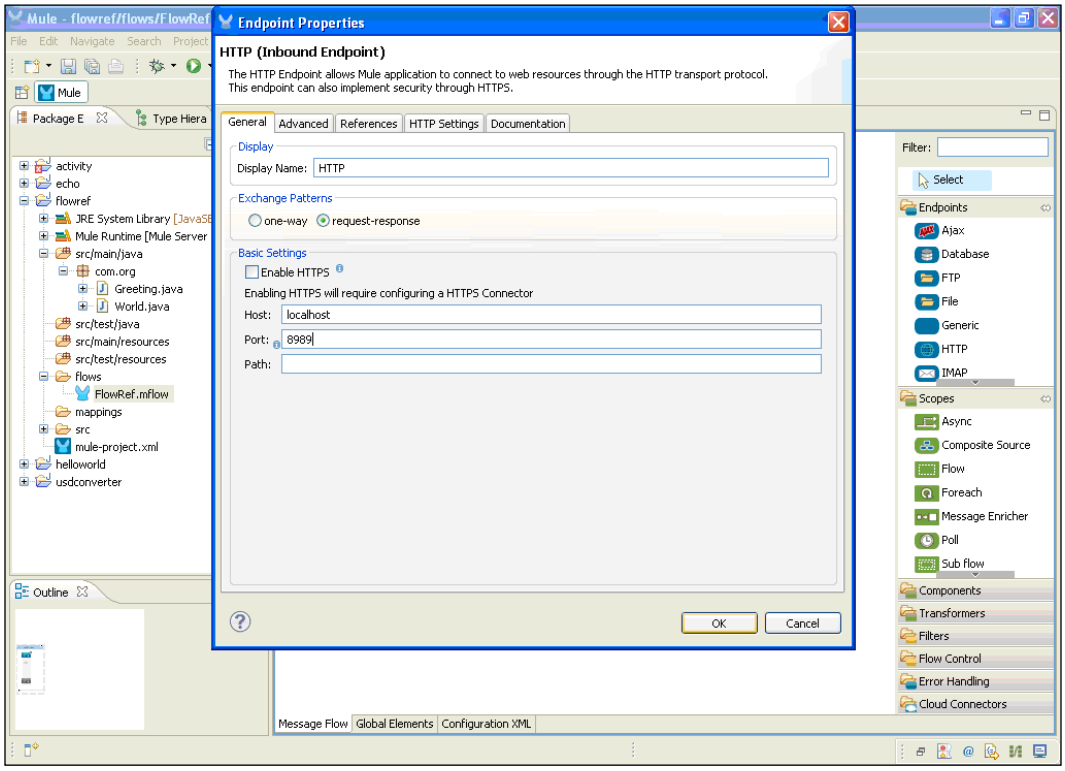
You can see the creation of this method in the following screenshot:



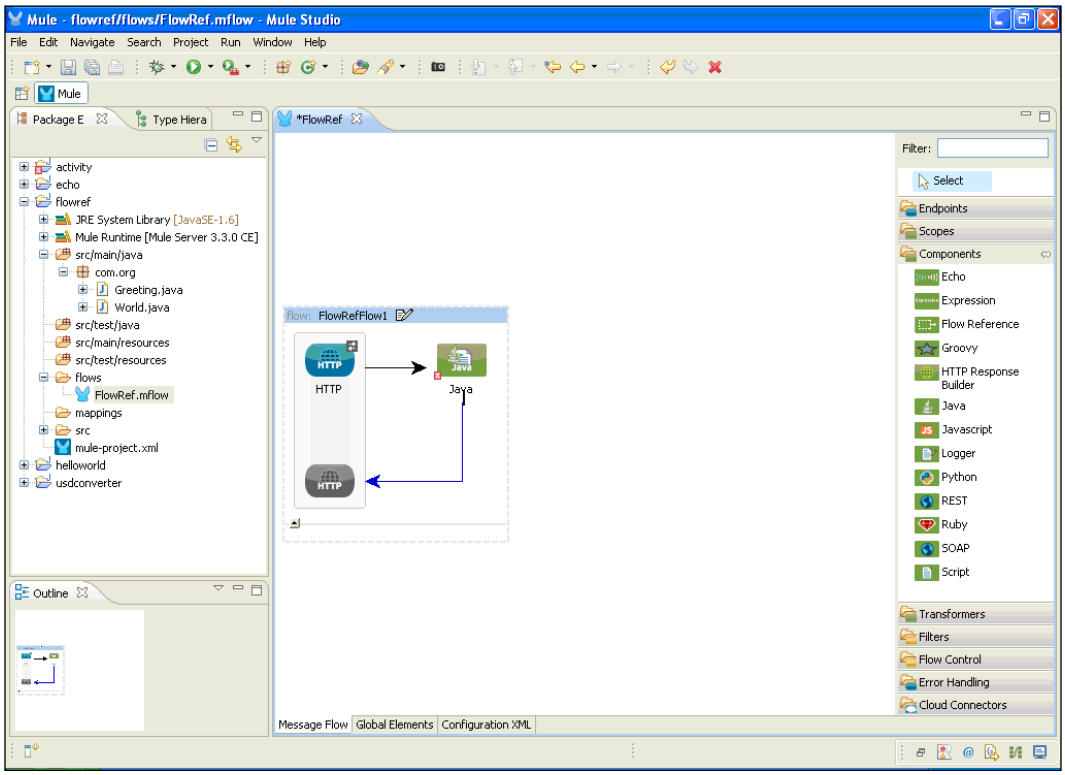
3. To create a flow, go to the `FlowRef.Mflow` file. Drag the **HTTP** Endpoint onto the canvas and configure it.



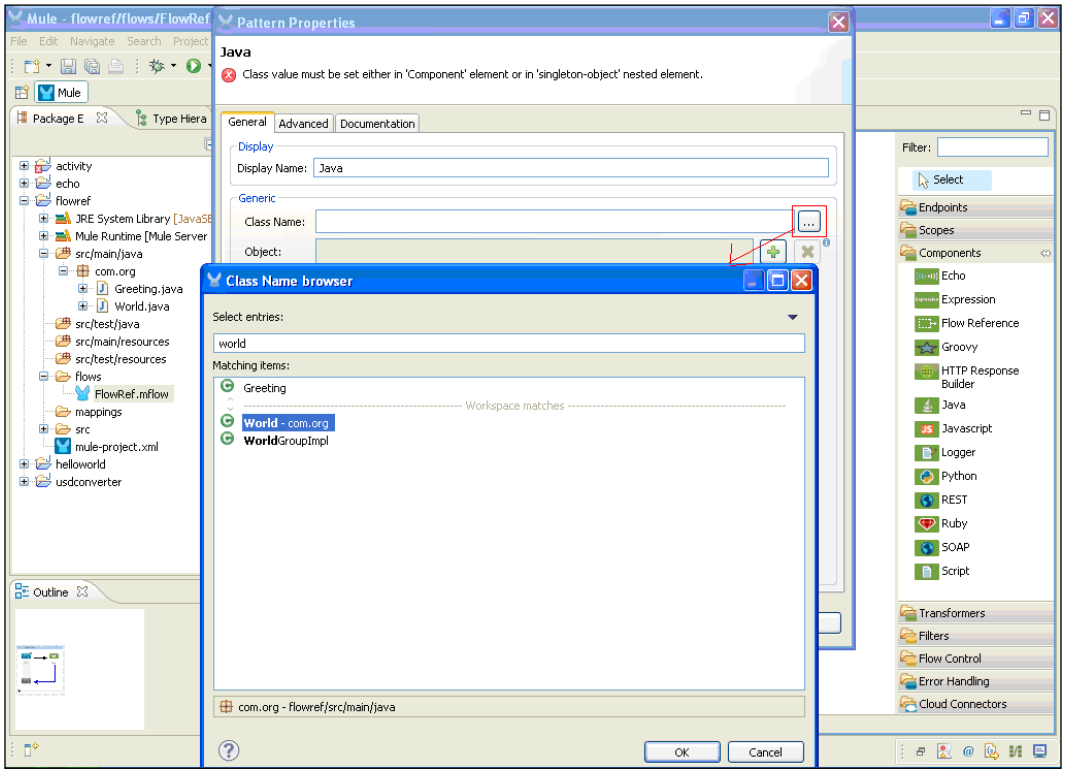
4. Double-click on the **HTTP** Endpoint to configure it. You can change the hostname and port number. We have used the port number `8989` here. Click on the **OK** button. By default, the **request-response** method is selected, as shown in the following screenshot:



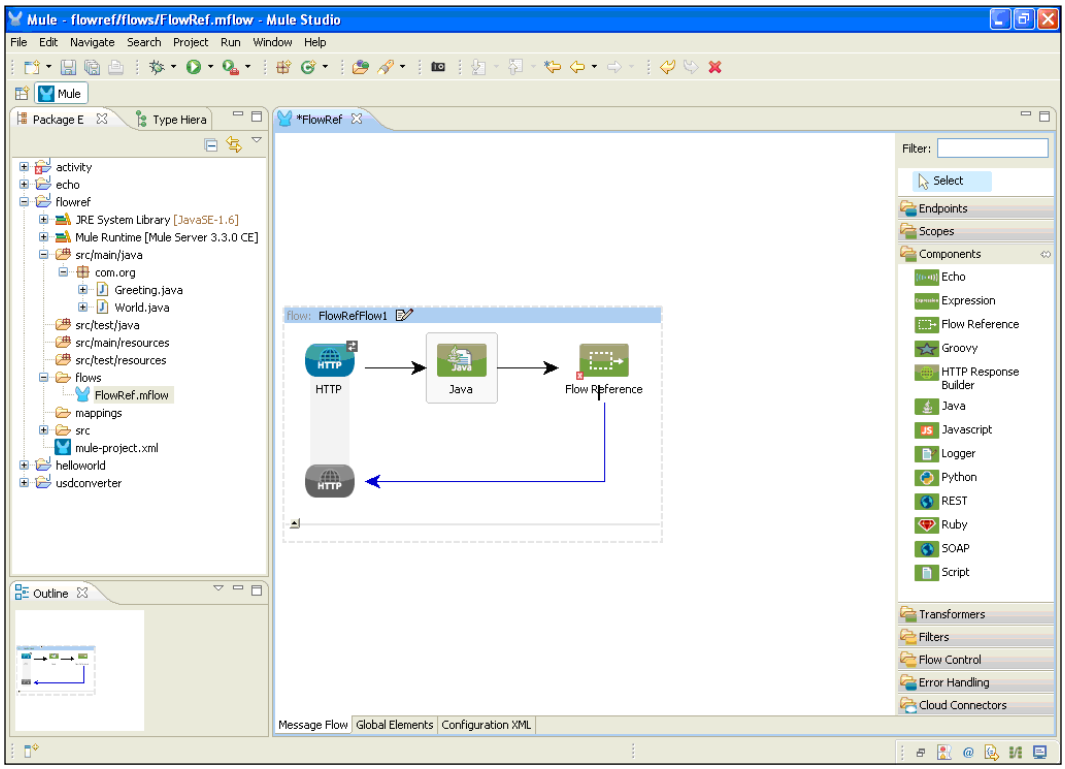
5. To import a class, drag the **Java** component and configure it.



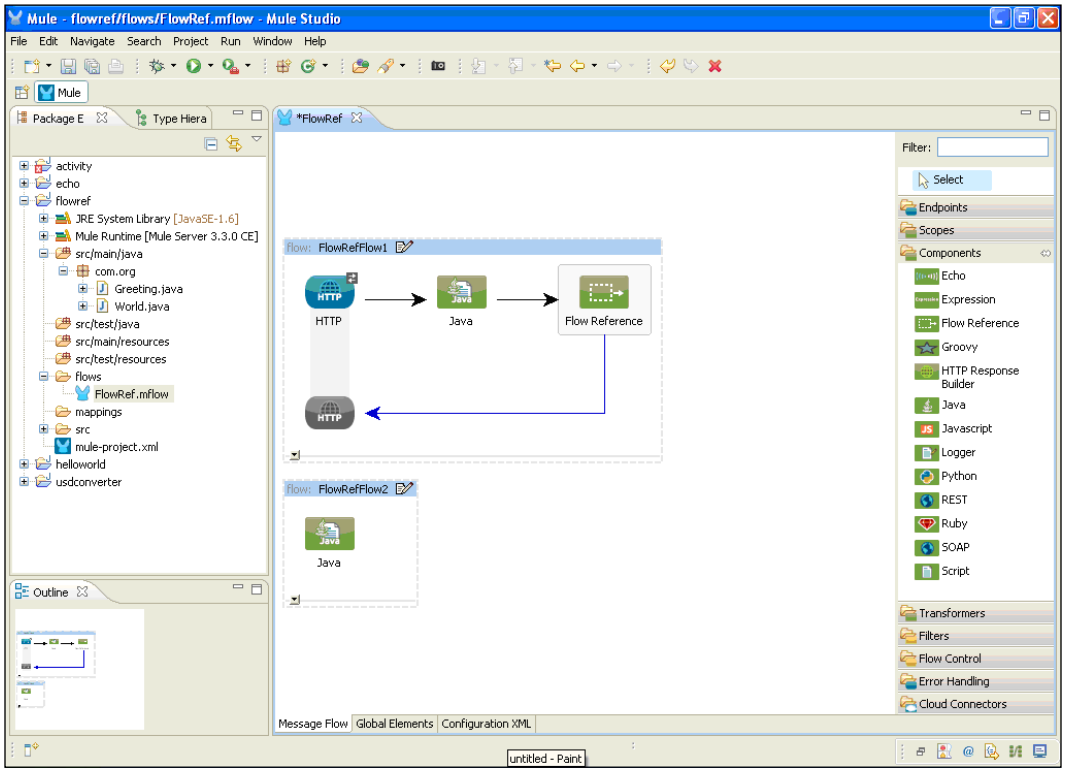
6. Double-click on the **Java** component to configure it. Just click on the **Browse** button and a new window, **Class name browser**, will open. Here you can import the `World.java` class, which was created before, and click on the **OK** button.



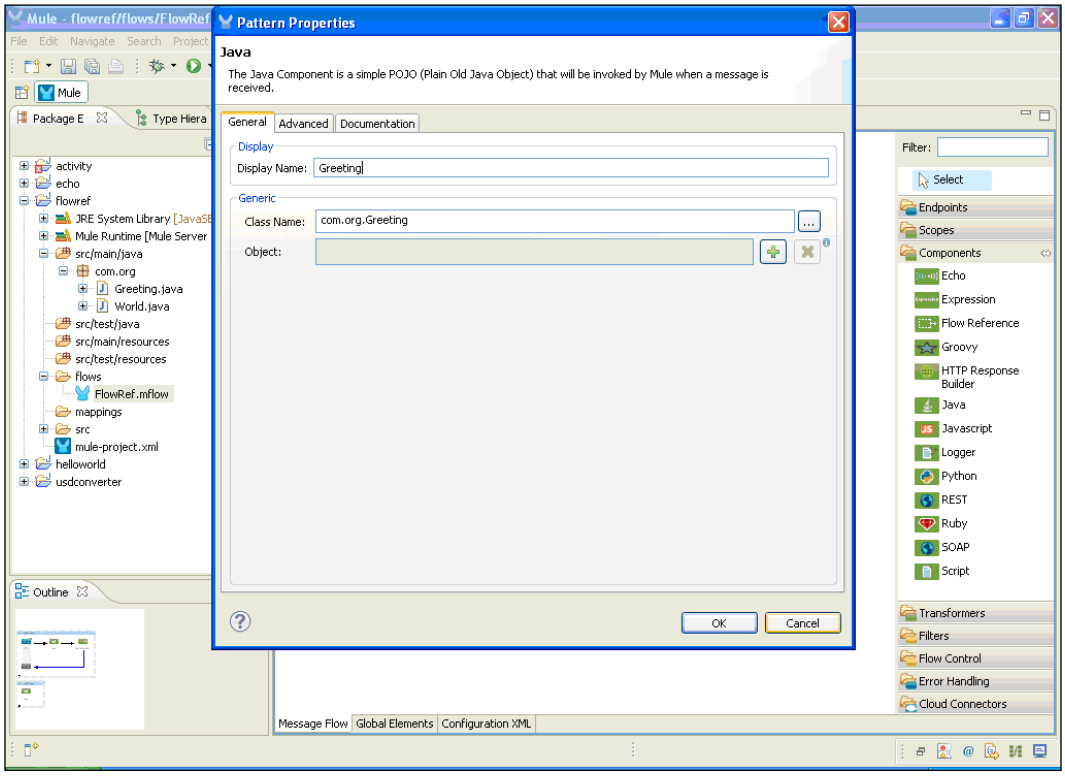
7. To reference another flow name, drag the **Flow Reference** component onto the canvas.



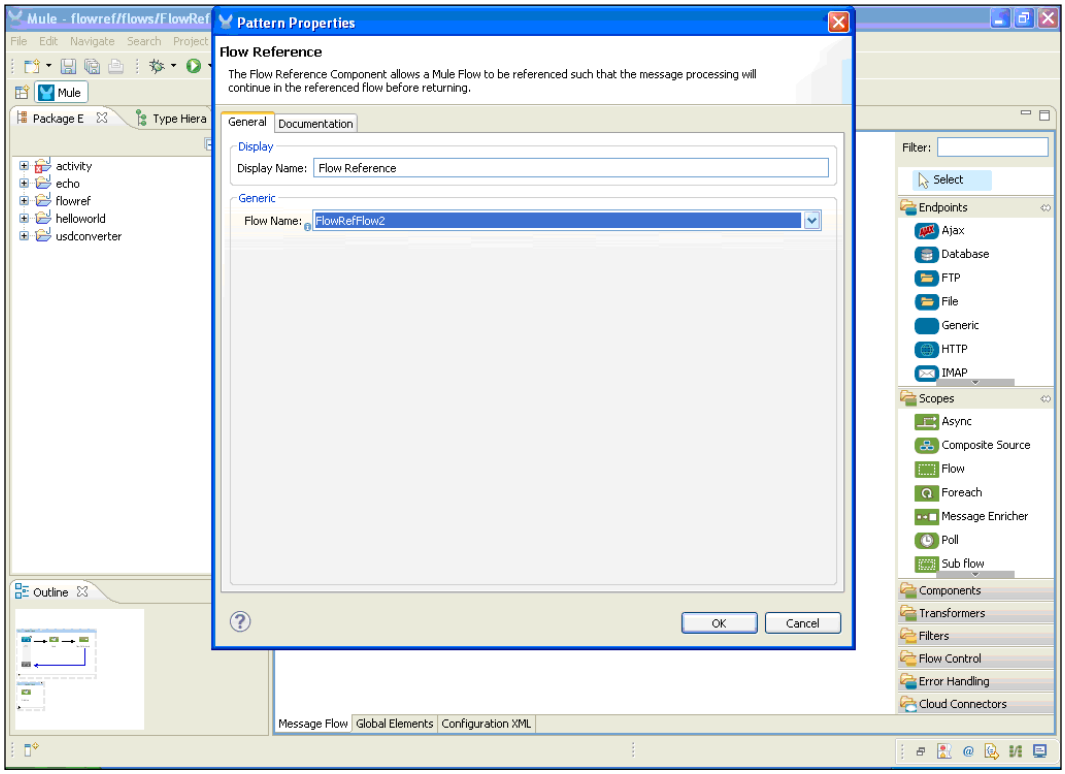
8. Before configuring the **Flow Reference** component, you have to drag the **Java** component onto another flow (you can see this in the following screenshot). Configure that **Java** component. If you create another flow, just drag the component onto the canvas outside the first flow; this will create another flow.



9. Double-click on the **Java** component to configure it; change the display name so you can identify the class name. Import the `Greeting.java` class that was created before and click on the **OK** button.



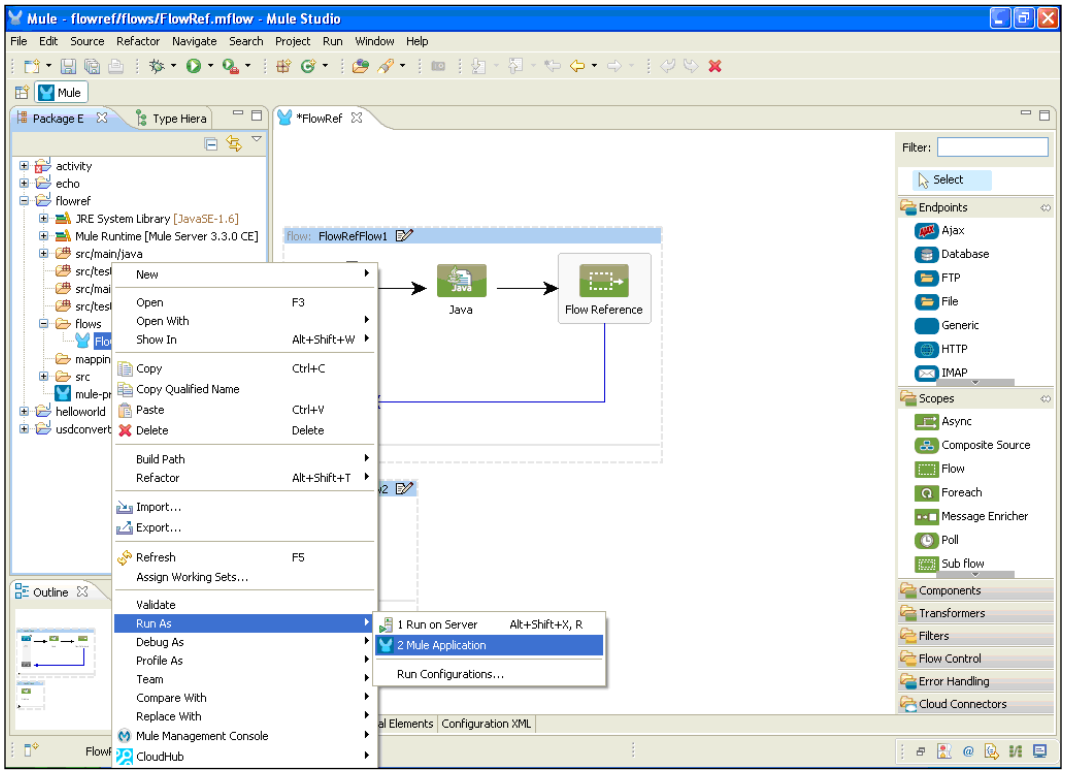
10. Now double-click on the **Flow Ref** component to configure it. Assign another Flow Reference name. Now we are ready to deploy our application.



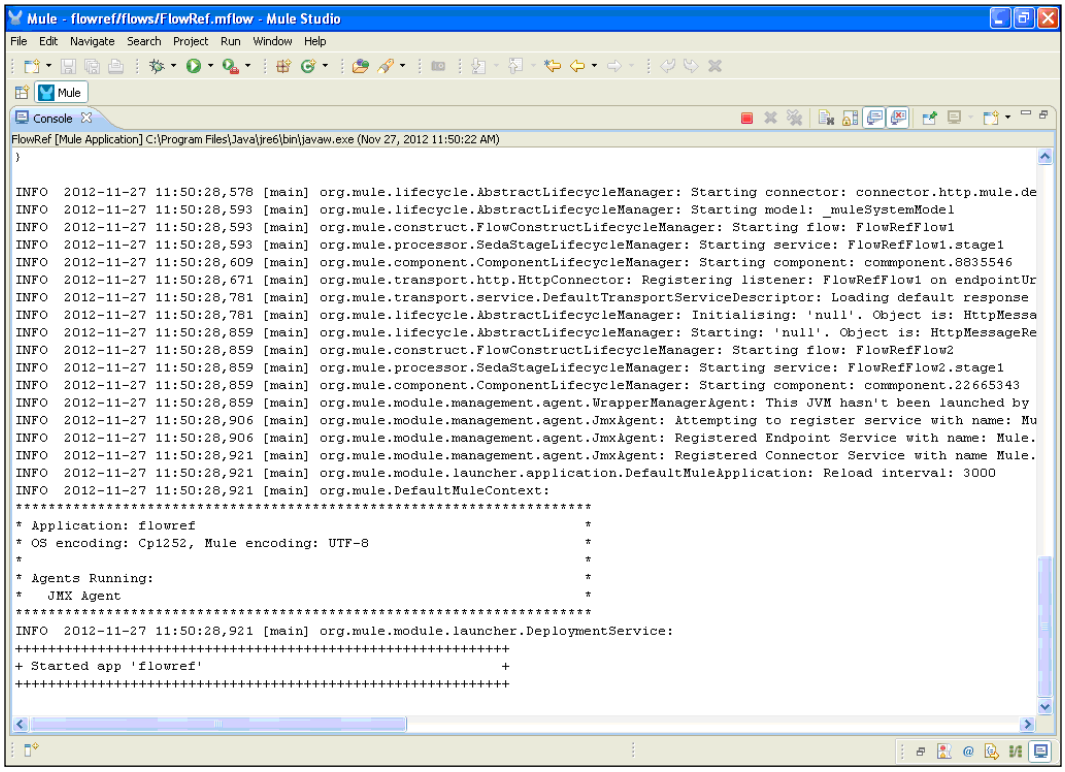
How it works...

In this section, you will see how to deploy the application and how to run the application in the browser.

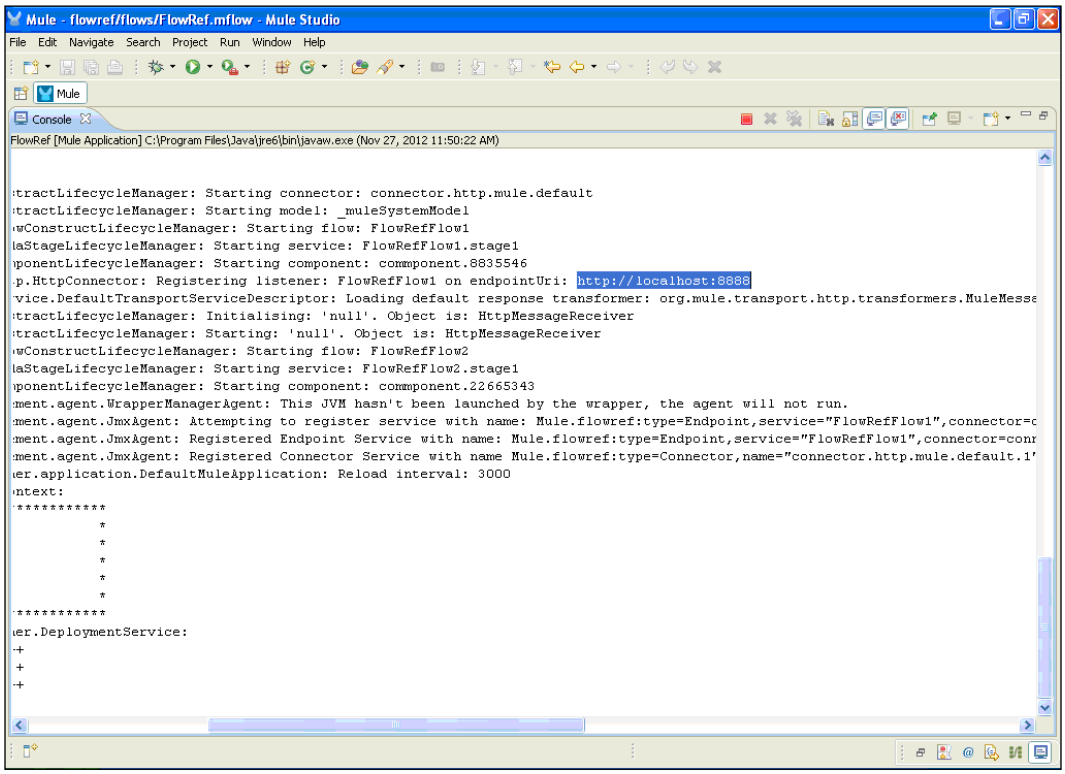
1. Now we are ready for the deployment. If you haven't saved your application code, do save it. After saving your project, right-click on the `Echo.mflow` file and go to **Run As | Mule Application**.



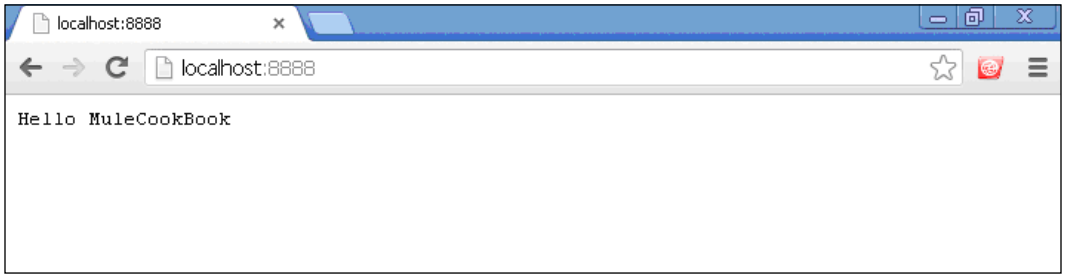
2. If your application code is successfully deployed, you will see the message `Started app 'helloworld'` on the console.



3. Copy the URL `http://localhost:8989` and paste it in your browser.



4. To see the output, paste the URL onto your browser. You will see the output as shown in the following screenshot. Here, `Hello Mule` is called from the `Greeting` class through the **Flow Reference** component, and `CookBook` is called from the `World` class:



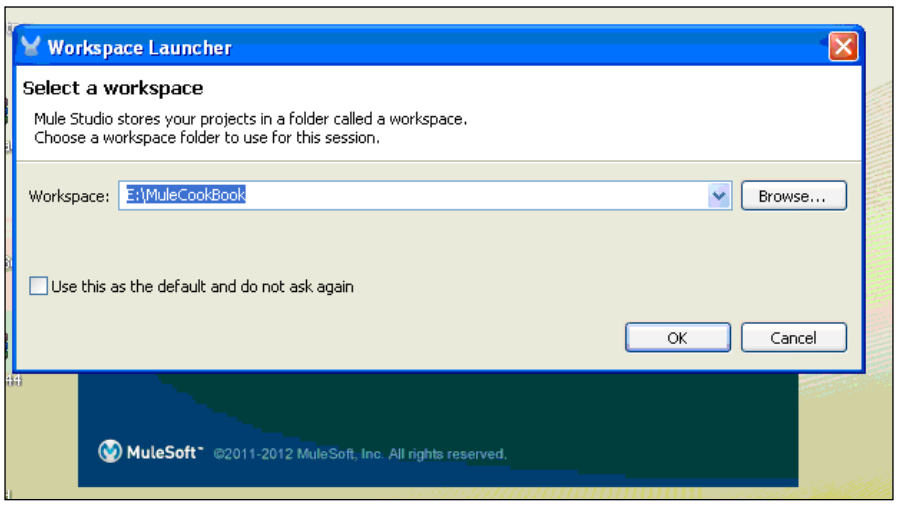
Publishing a RESTful web service using the REST component

REST stands for **Representational State Transfer**. REST exposes a much simpler interface than SOAP. REST components are bound with HTTP. So, if you are designing an application to be used exclusively on the Web, REST is a very good option. RESTful applications simply rely on the built-in HTTP security. A REST design is good for database-driven applications and also when a client wants quick integration.

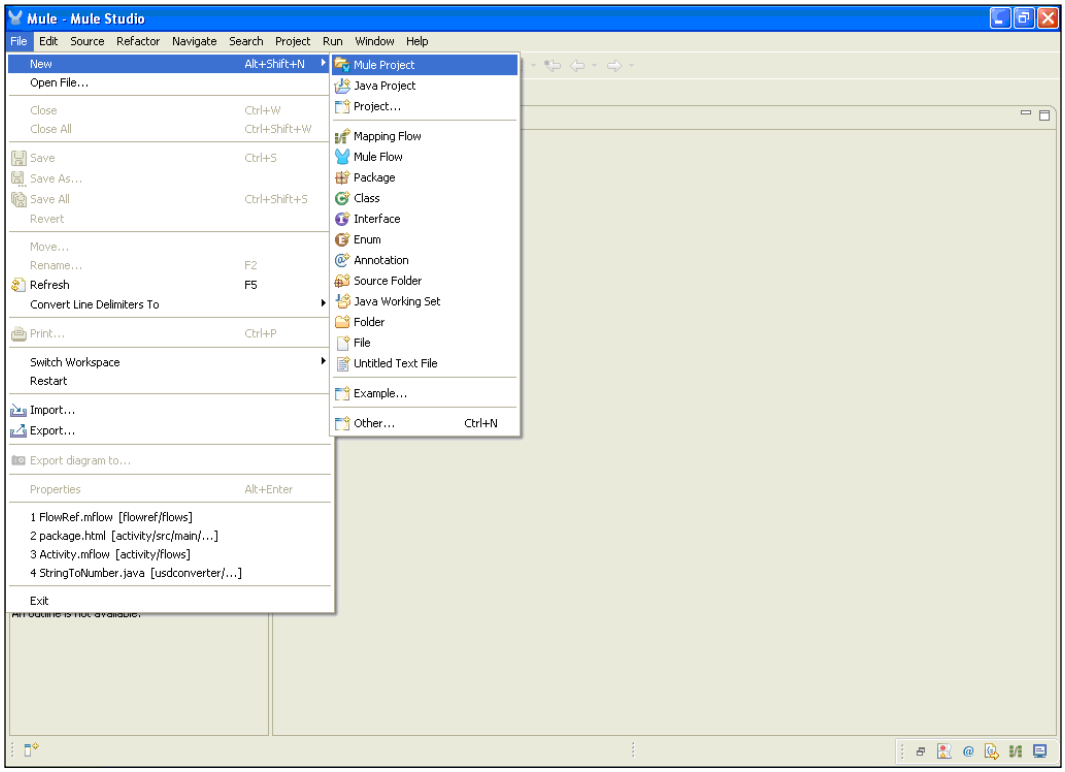
Getting ready

In this example, we'll use three components: HTTP, Logger, and REST.

1. Open Mule Studio and enter the workspace name as shown in following screenshot:



2. To create a new project, go to **File** | **New** | **Mule Project**. Enter the project name, `REST`, and click on **Next** and then on **Finish**. Your new project has been created; now you can start the implementation.



How to do it...

Here we will create a RESTful web service using the annotation. We will create a method named `getwelcomeMsg()`.

1. To create a class, go to `src/main/java` and right-click on it. Create a class named `HelloWorldResource` to print a message. Enter the package name and click on **Next** and then on **Finish**. Here we have used the JAX-WS annotation. For details on the JAX-WS annotation, you can refer to this URL: `http://publib.boulder.ibm.com/infocenter/radhelp/v7r0m0/index.jsp?topic=/com.ibm.ws.jaxws.emitter.doc/topics/rwsandoc002.html`.

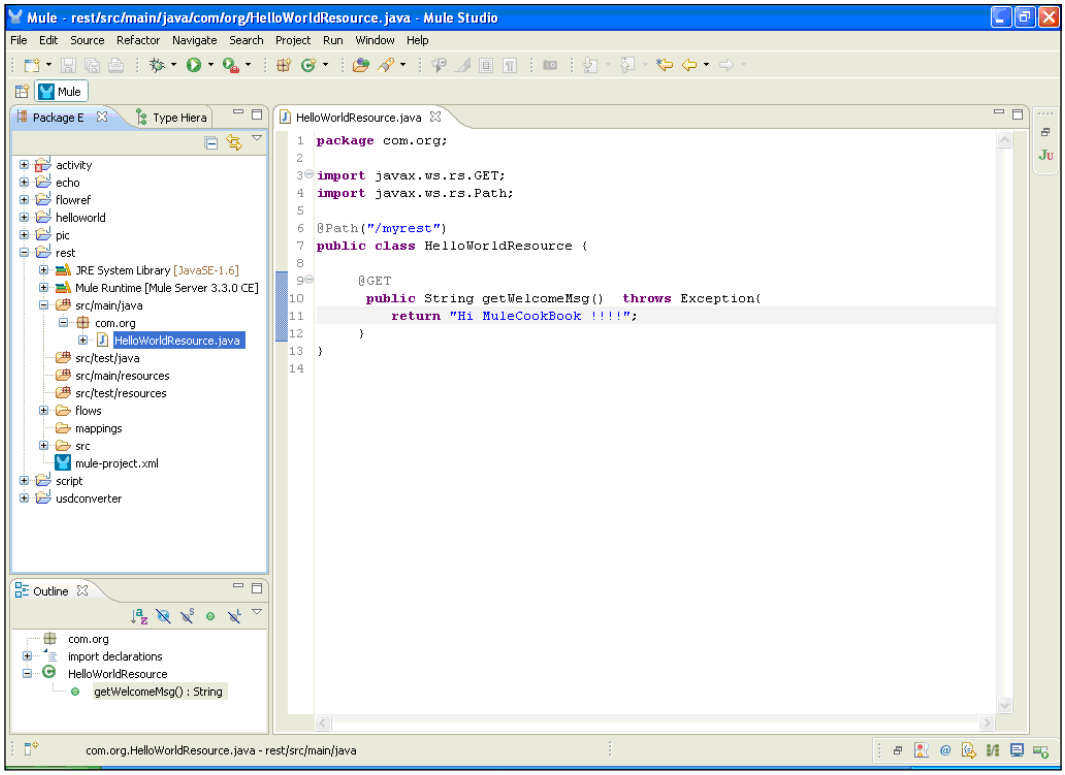
```
package com.org;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
@Path("/myrest")
```

```
public class HelloWorldResource {
  @GET
    public String getWelcomeMsg () throws Exception {
       return "Hi MuleCookBook!!!!";
  }
}
```

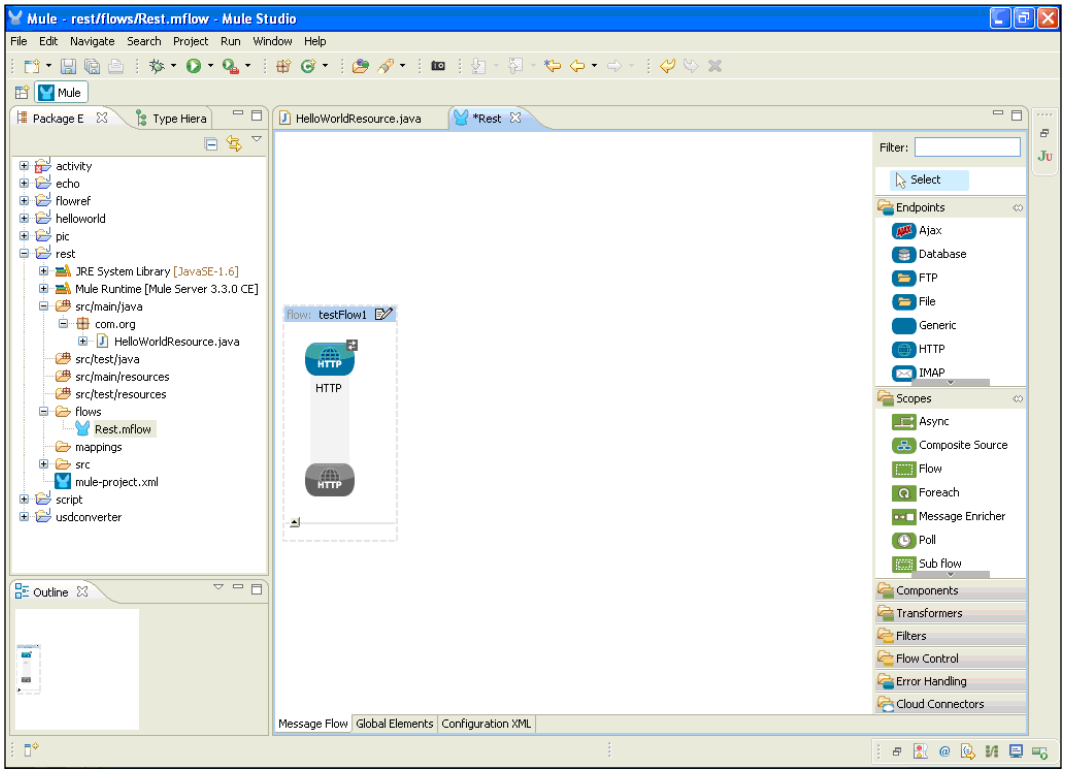
The `@Get` annotation indicates that the annotated method responds to an HTTP GET request.

The `@Path` annotation is used to map a given URL.

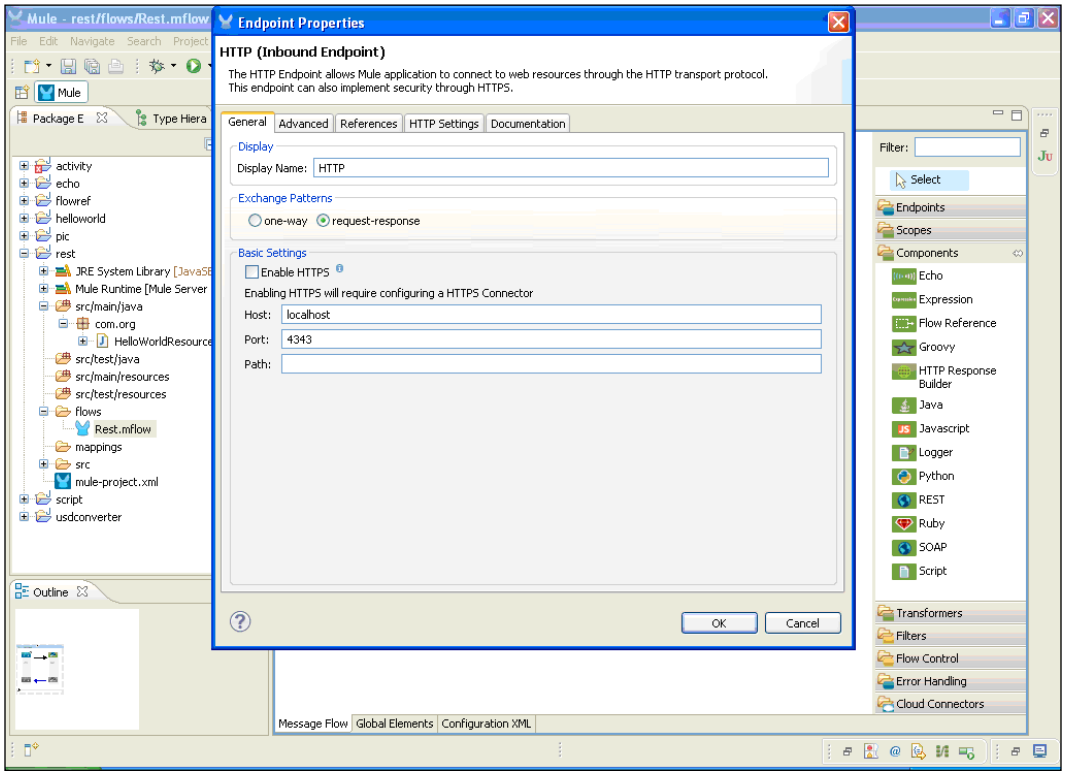
You can see the creation of this method in the following screenshot:



2. To create a flow, go to the `Rest.mflow` file. First of all, you have to drag the **HTTP** Endpoint from the palette and drop it on the canvas area.

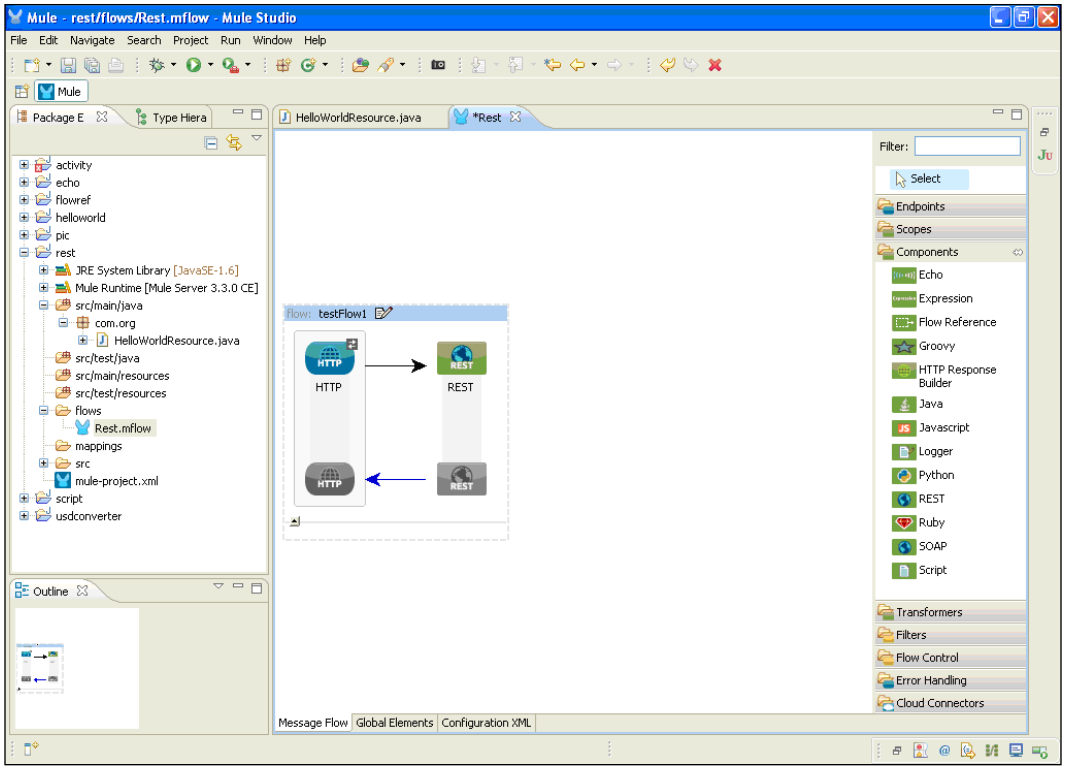


3. Double-click on the **HTTP** Endpoint to configure it. You will see the hostname and port number. You can change the **Host** and **Port** field values. These two fields are mandatory. By default, port number **8081** will be taken by the Mule server. We have used port number `4343` here.

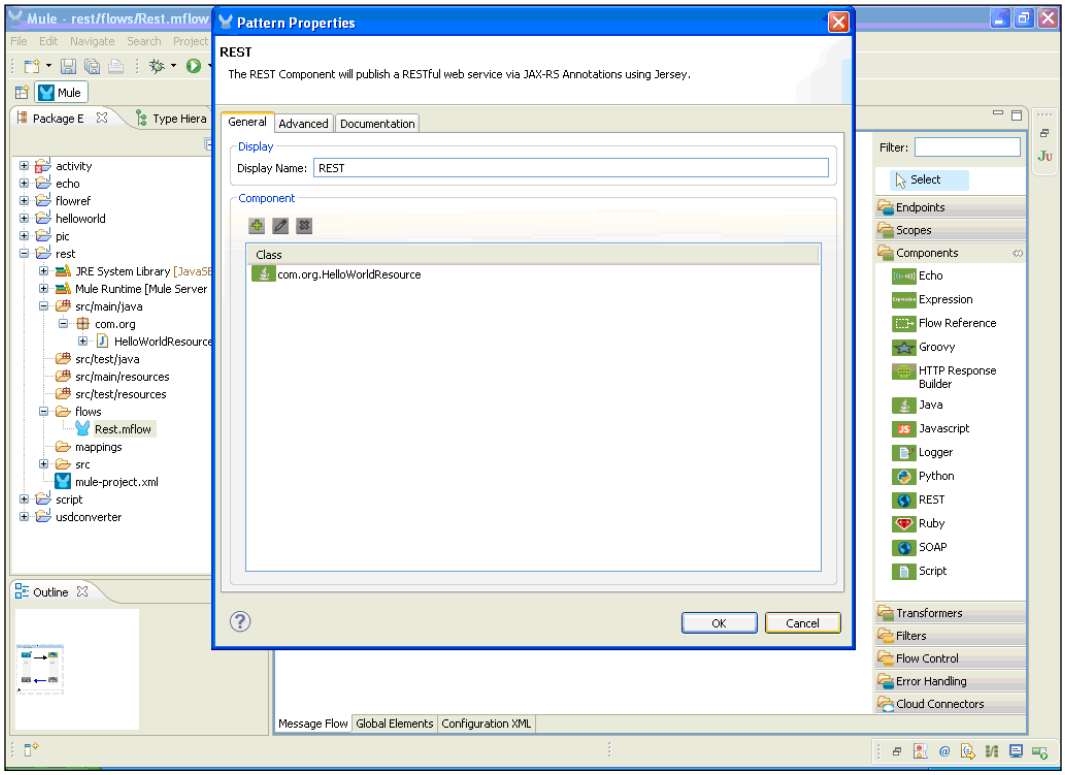


4. To create a RESTful web service, drag the **REST** component from the palette and drop it on the canvas area. This **REST** component is used to make a REST service available via **Jersey**. Jersey is an open source, production-quality, JAX-RS (JSR 311) reference implementation for building RESTful web services.

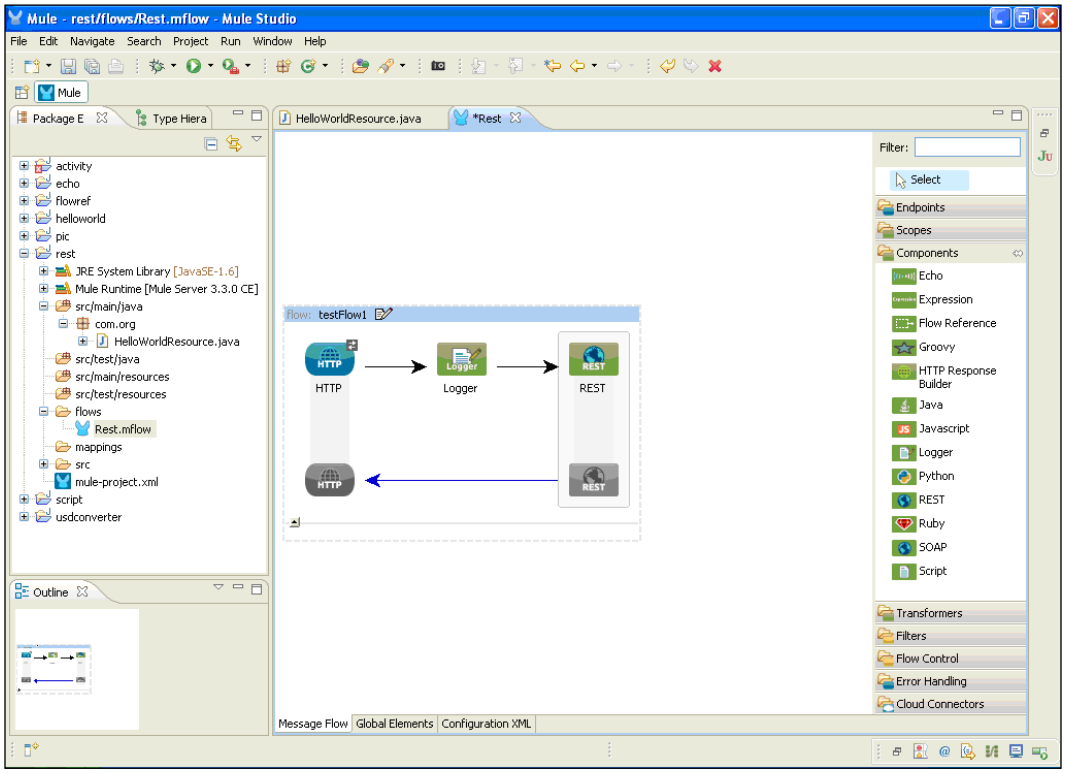
   REST is the formalized architecture of HTTP and is based on concepts of resources, links, and a uniform interface. It uses the HTTP protocol. We can create a web service using the **REST** component.



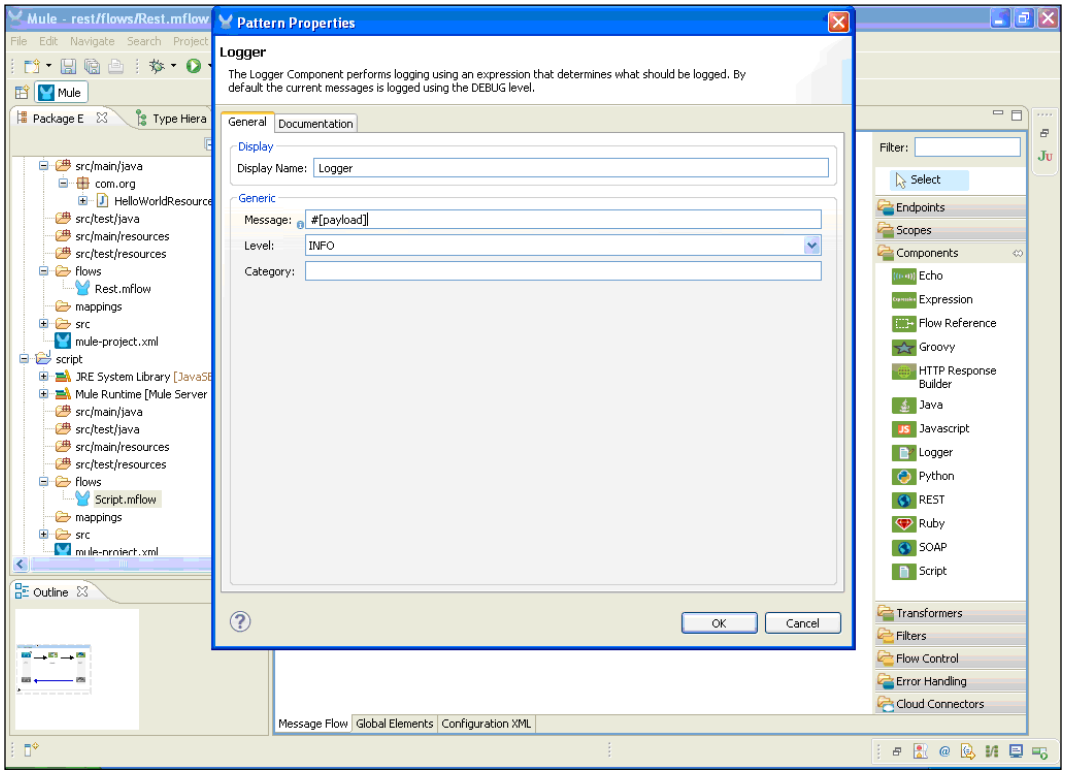
5. Double-click on the **REST** component to configure it. Here you can add a **Java** component that was created before.



6. To display messages on the console, drag-and-drop the **Logger** component from the palette onto the canvas area and configure it.



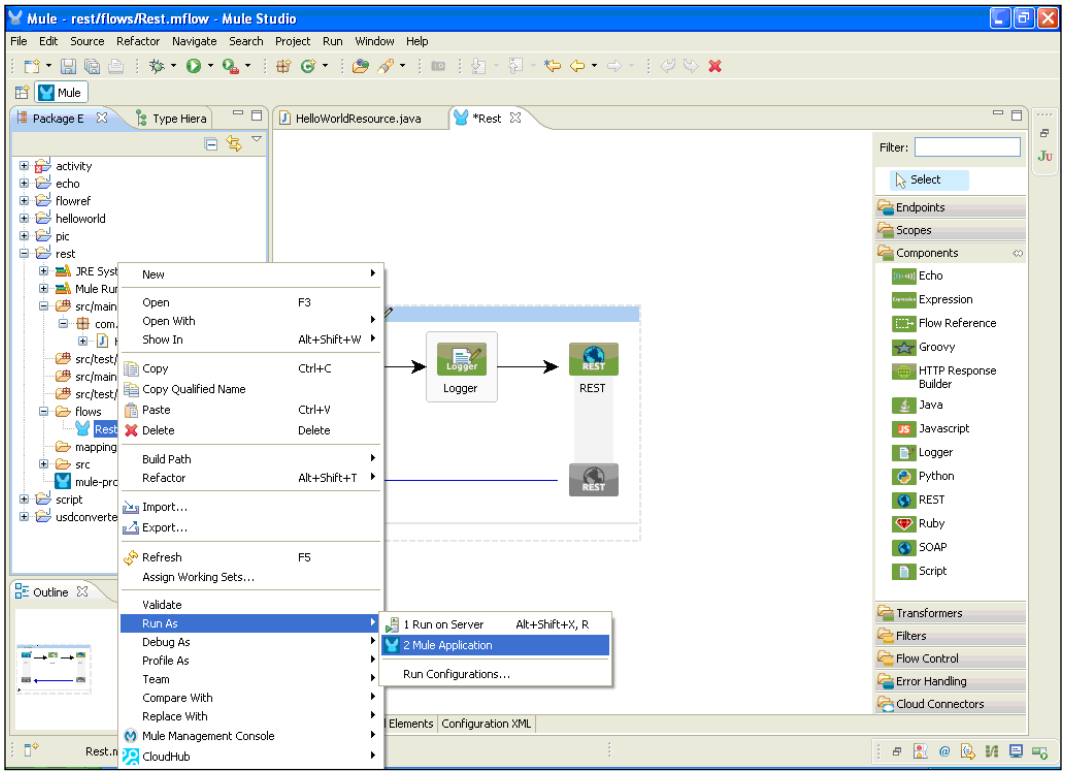
7. To configure the **Logger** component, double-click on it. You will see the **Message:** textbox; just enter the payload expression, `#[payload]`.



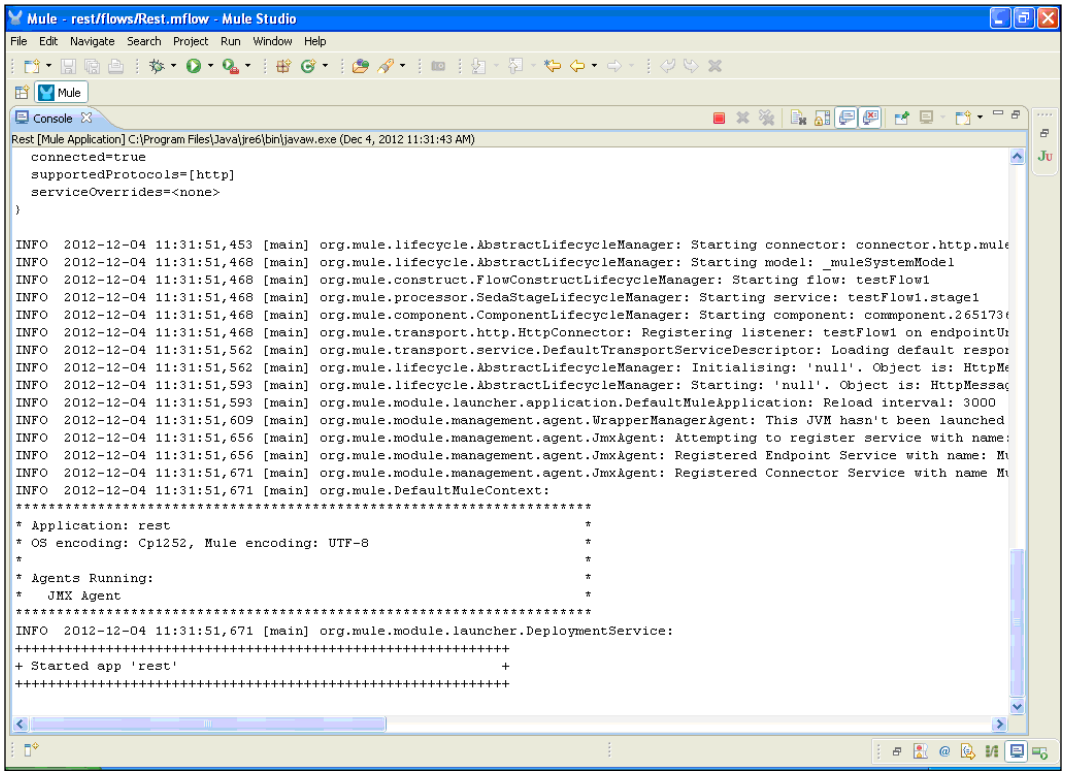
How it works...

To deploy your application go through the following steps:

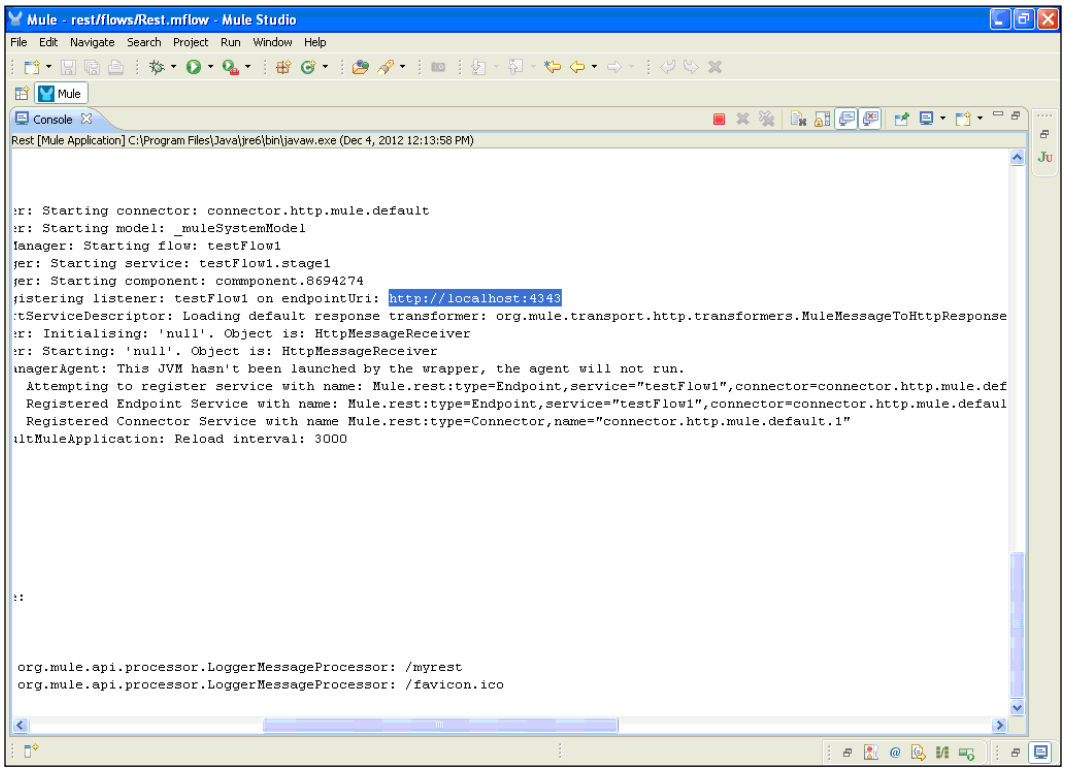
1. If you haven't saved your application code, do save it. After saving your project, right-click on the `Echo.mflow` file and go to **Run As** | **Mule Application**.



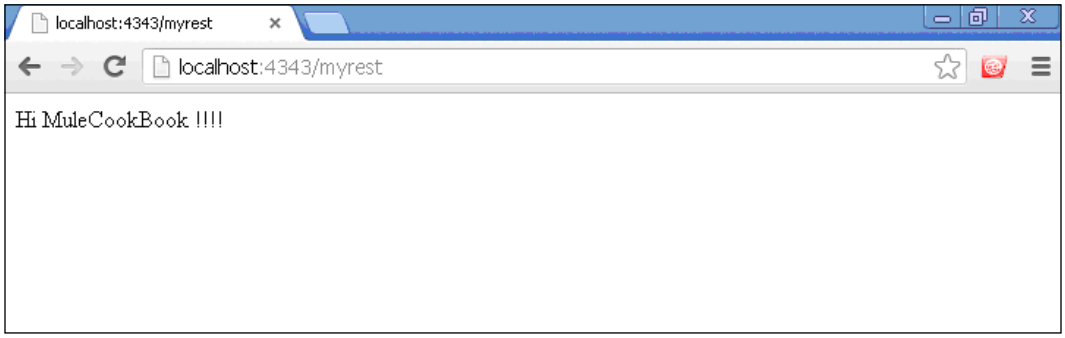
2. If your application code is successfully deployed, you will see the message `Started app 'helloworld'` on the console.



3. Copy the URL `http://localhost:4343/myrest` and paste it in your browser.



4. To see the output, paste the URL on your browser and type `/myrest`; this is required because we have used the `@Path` annotation in the custom Java class.



Publishing a SOAP-based web service using the SOAP component

The Mule **SOAP** component is used for publishing, consuming, and proxying of SOAP web services within a Mule flow. Using the SOAP component, you can also enable Web Service Security. Apache CXF is an open source services framework. CXF helps you build services using frontend programming APIs such as JAX-WS and JAX-RS.

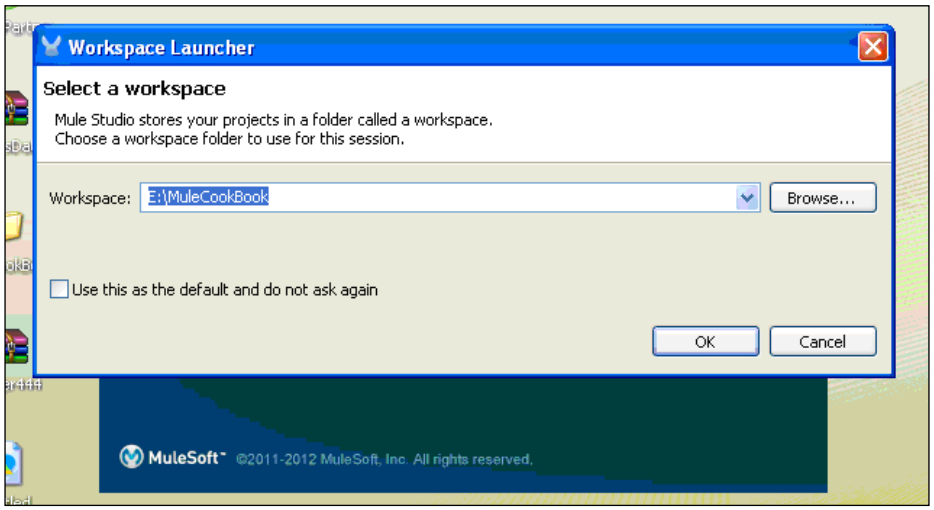
You can create a CXF web service by configuring a SOAP component in your Mule flow to perform any of the following CXF web service operations:

- Publish a simple service
- Publish a JAX-WS service
- Proxy a published service
- Consume a service using a simple client
- Consume a service using the JAX-WS client
- Proxy to a service

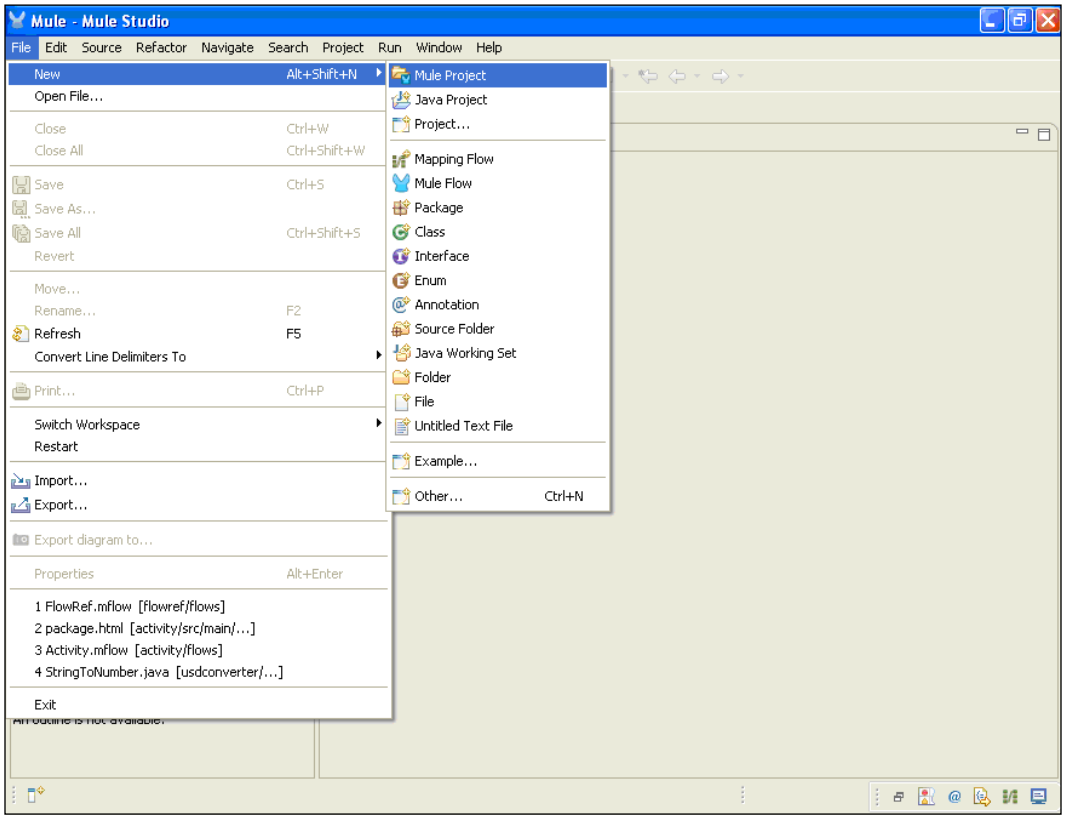
Getting ready

In this example, we will see how to create a SOAP-based web service using the SOAP component. To create a SOAP web service, we'll use three components: HTTP, Java, and SOAP.

1. Open Mule Studio and enter the workspace name as shown in the following screenshot:



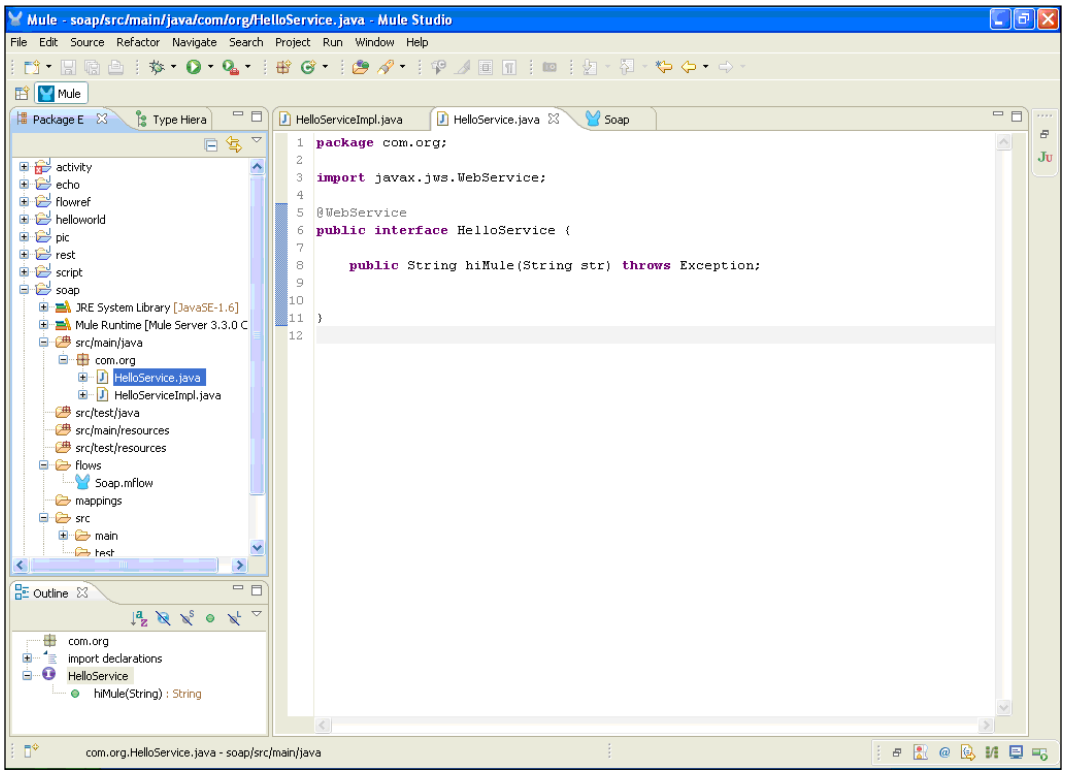
2. To create a new project, go to **File** | **New** | **Mule Project**. Enter the project name, `SOAP`, and click on **Next** and then on **Finish**. Your new project is created and you are now ready to start the implementation.



3. To create a class, go to `src/main/java`, right-click on it, and go to **New** | **Interface**. Create an interface named `HelloService` under the package `com.org`. Here, we create the `hiMule` method and set its return type to `String`.

```
package com.org;
import javax.jws.WebService;
@WebService
public interface HelloService {
  public String hiMule(String str) throws Exception;
}
```

You can see the creation of this class in the following screenshot:



How to do it...

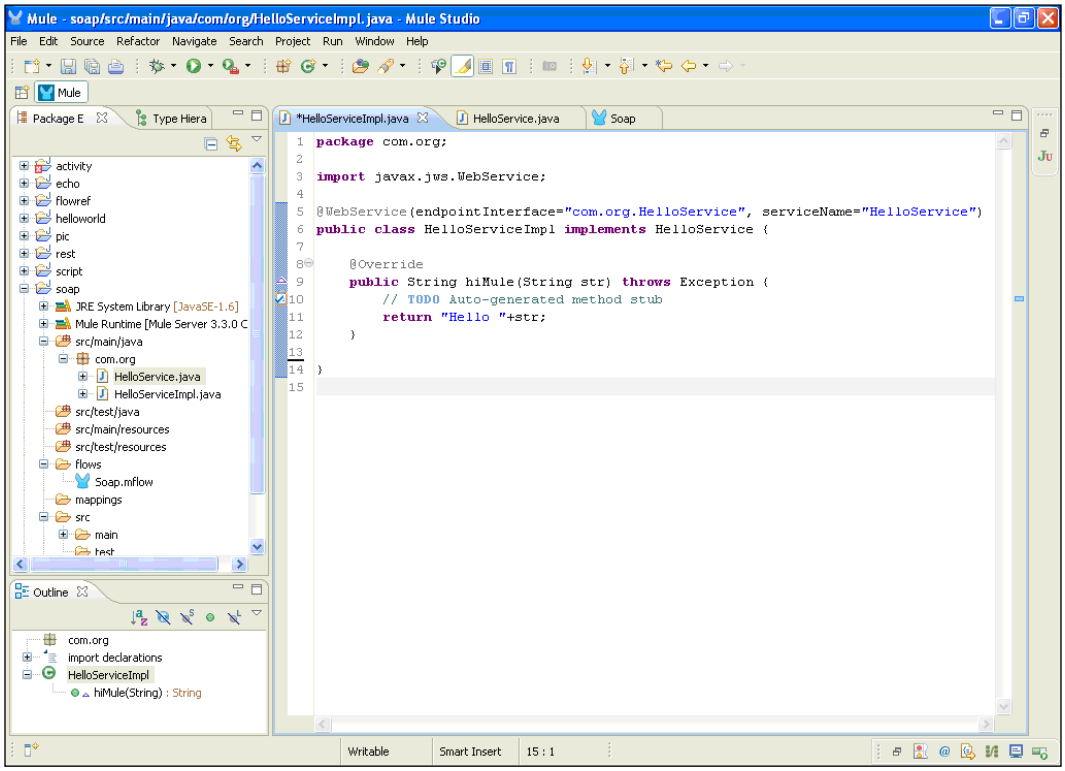
In this section, you will refer to the JAX-WS annotation and create the method `hiMule()`. We will use this method to generate the output using a browser.

1. Create a class called `HelloServiceImpl` under the same package directory (`com.org`) and implement it with the interface. Here, we have used the `@WebService` annotation to create a SOAP-based web service and override the `hiMule` method. You can refer to this URL for more information on the JAX-WS annotation: `http://publib.boulder.ibm.com/infocenter/radhelp/v7r0m0/index.jsp?topic=/com.ibm.ws.jaxws.emitter.doc/topics/rwsandoc002.html`.

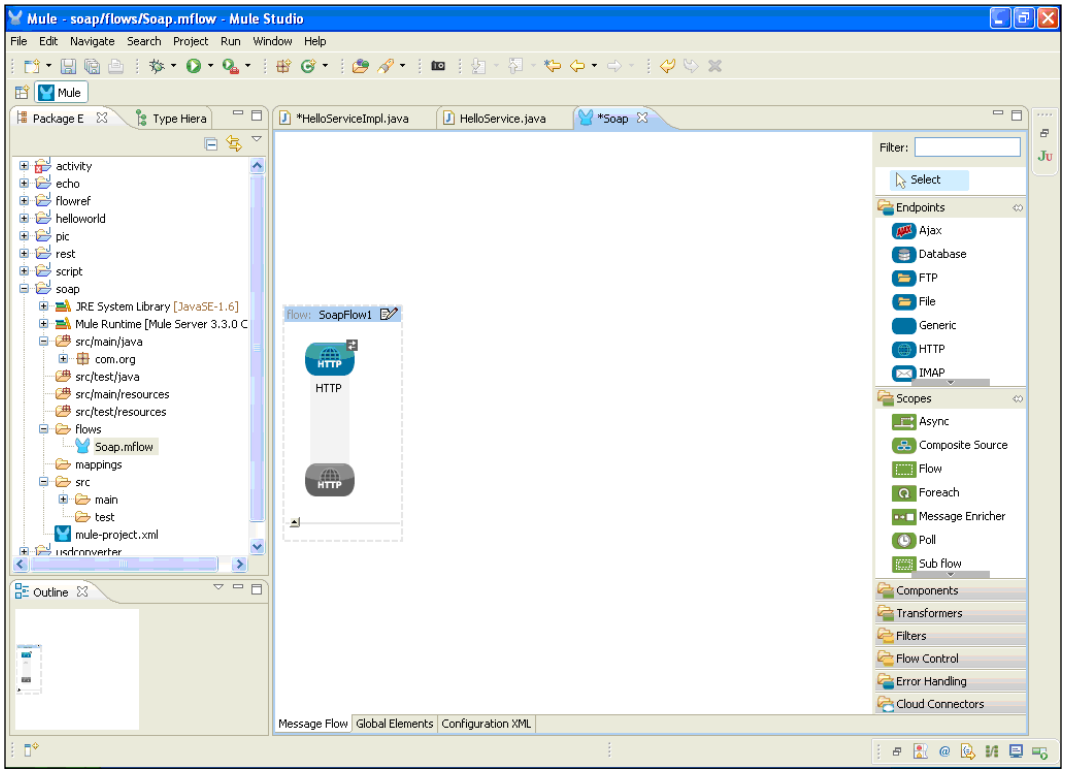
```
package com.org;
import javax.jws.WebService;
@WebService(EndpointInterface="com.org.HelloService",
serviceName="HelloService")
public class HelloServiceImpl implements HelloService {
  @Override
```

```
    public String hiMule(String str) throws Exception {
      // TODO Auto-generated method stub
      return "Hello "+str;
    }
}
```

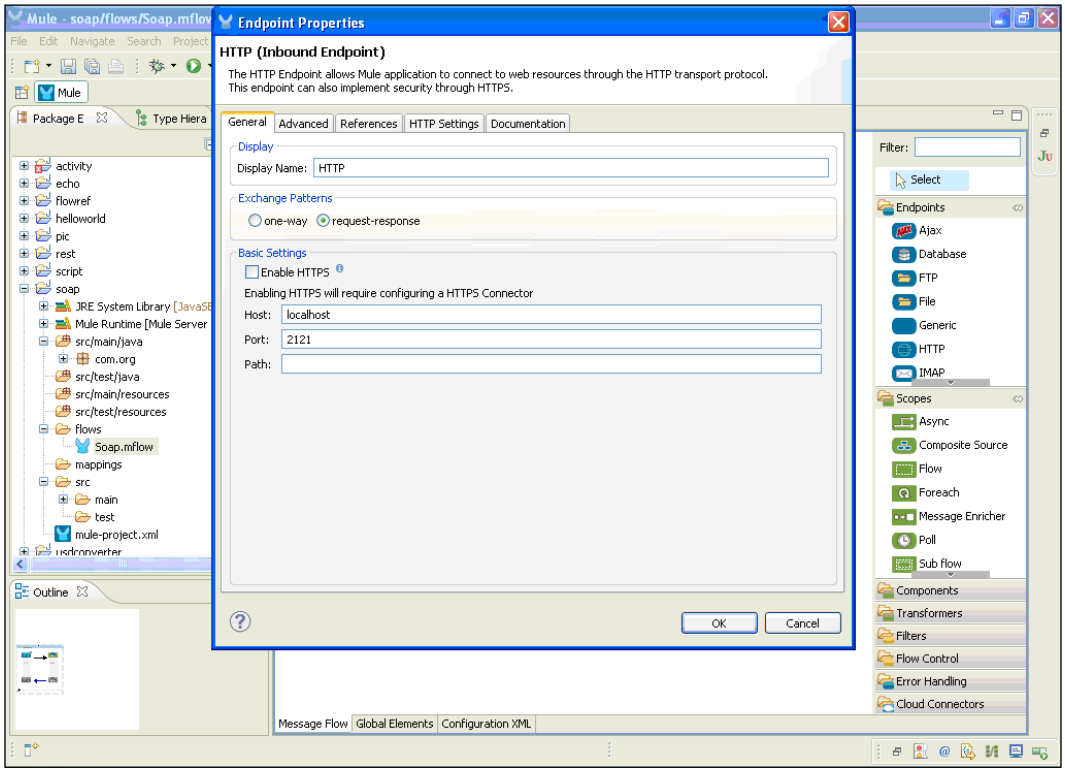
Through the `@WebService` annotation, we call the `HelloService` interface and also provide a service name.



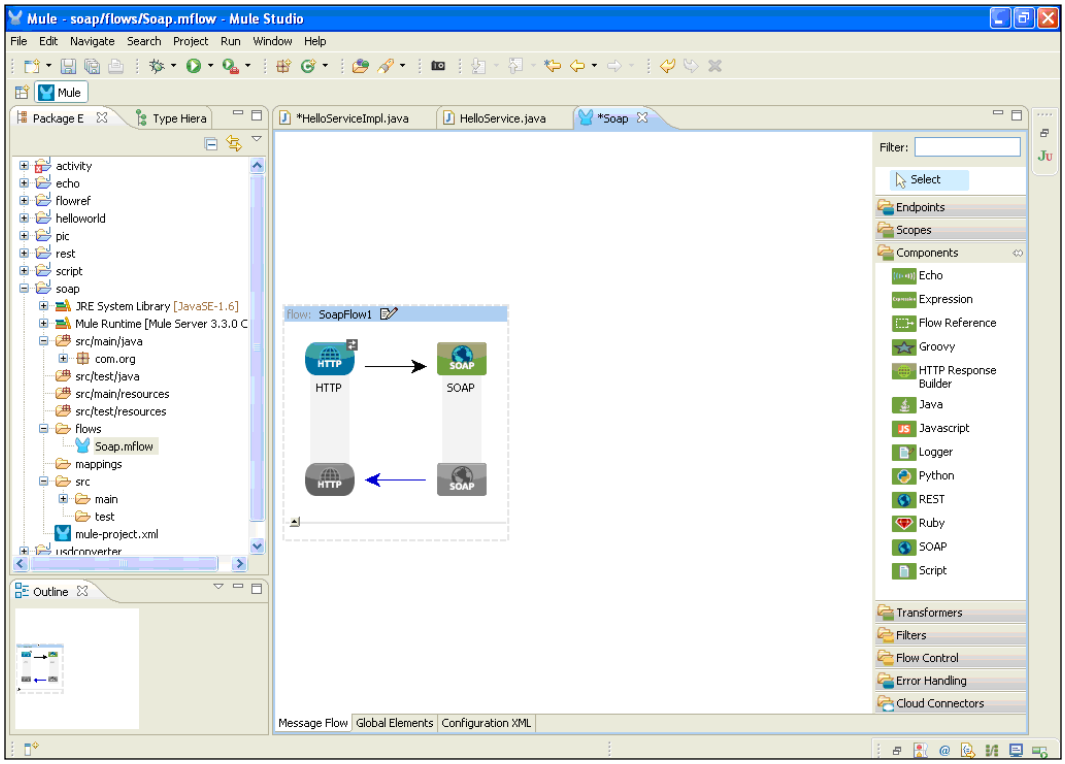
2. To create a flow, go to the `Soap.mflow` file. Drag the **HTTP** Endpoint from the palette and drop it onto the canvas. You have to configure the **HTTP** Endpoint.



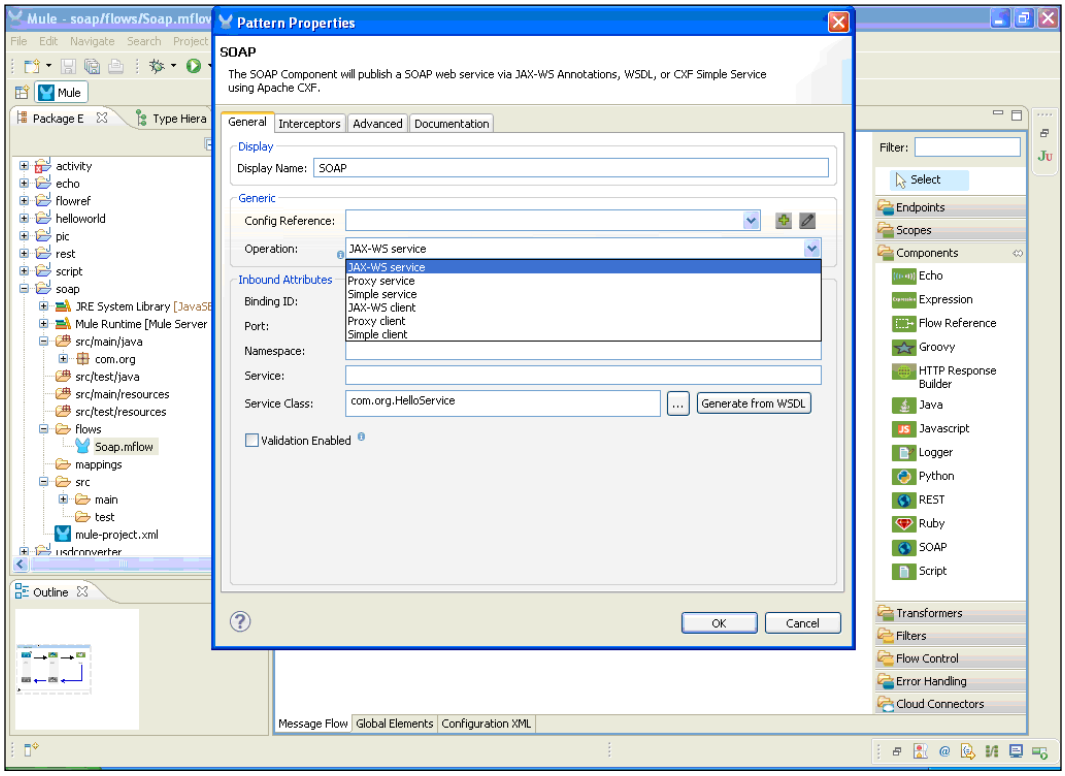
3. Double-click on the **HTTP** Endpoint to configure it. You will see the **Host** and **Port** fields. These two fields are mandatory. In this example, we used `localhost` as the hostname. By default, port number **8081** will be taken by the Mule server. We have used port number `2121` here.



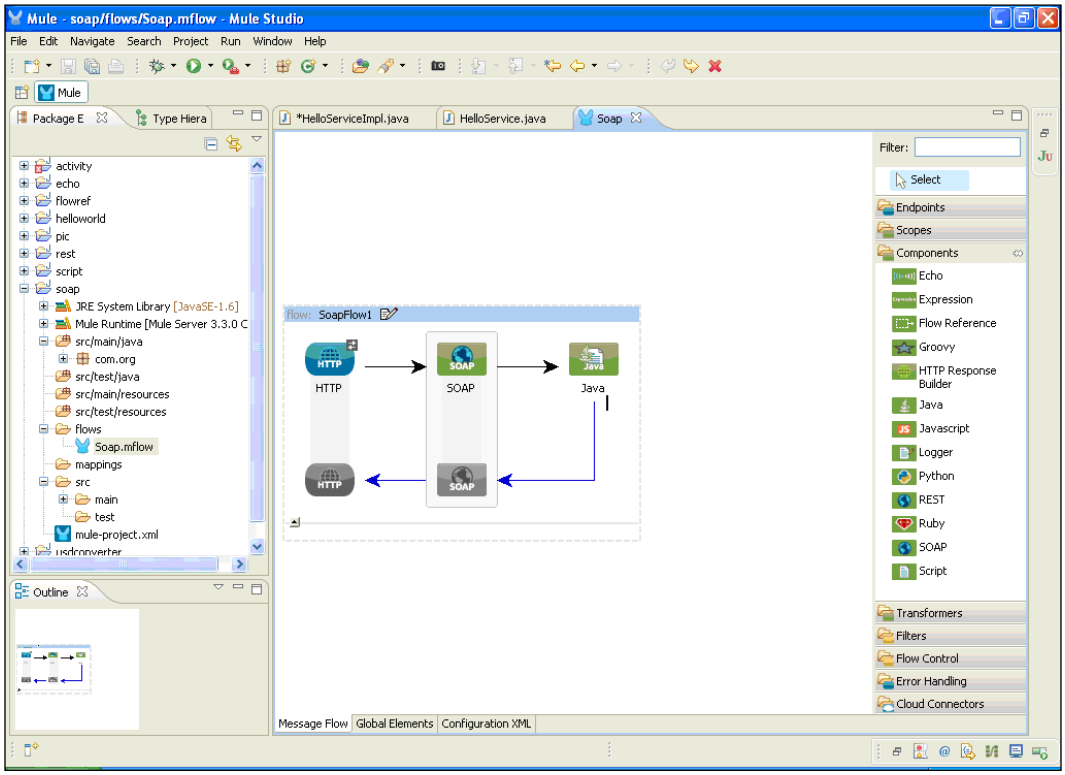
4. To create a SOAP-based web service, drag the **SOAP** component from the palette, drop it onto the canvas, and configure it. Here, we create web services using the **SOAP** component.



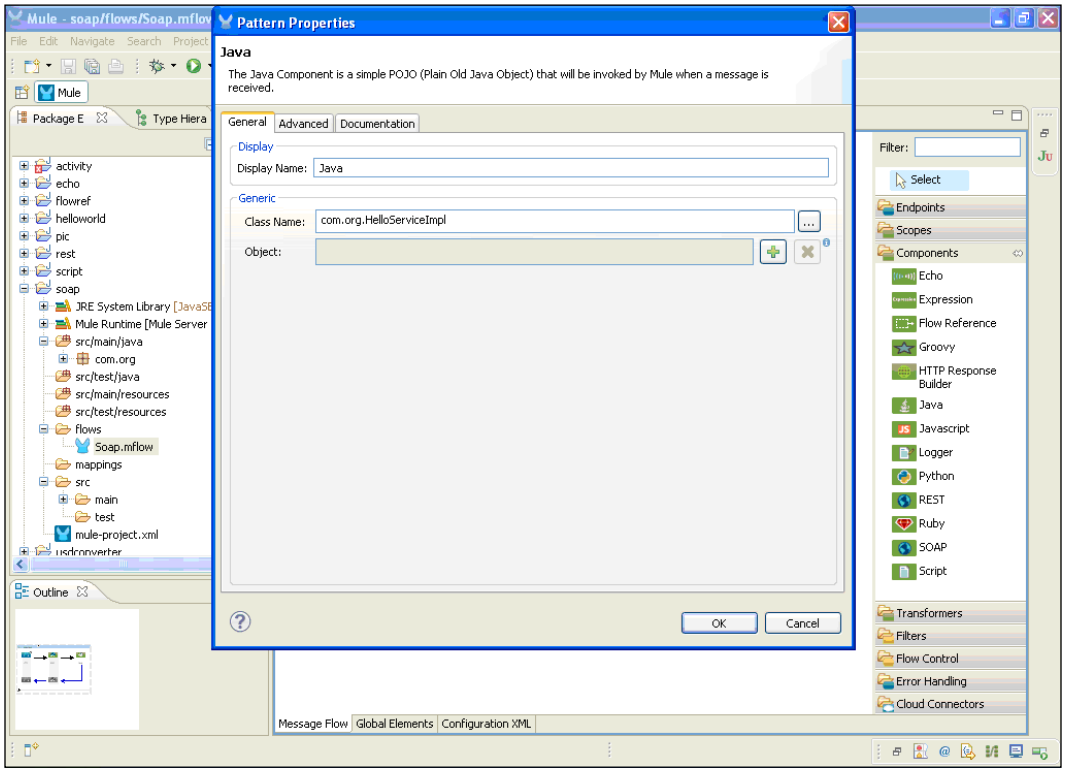
5. Double-click on the **SOAP** component to configure it. We select the **JAX-WS** services for the operation and then we will import the `HelloService` interface that was created before.



6. To import a class, drag the **Java** component from the palette, drop it on the canvas area, and configure it. If you want to change its name, you can do so.



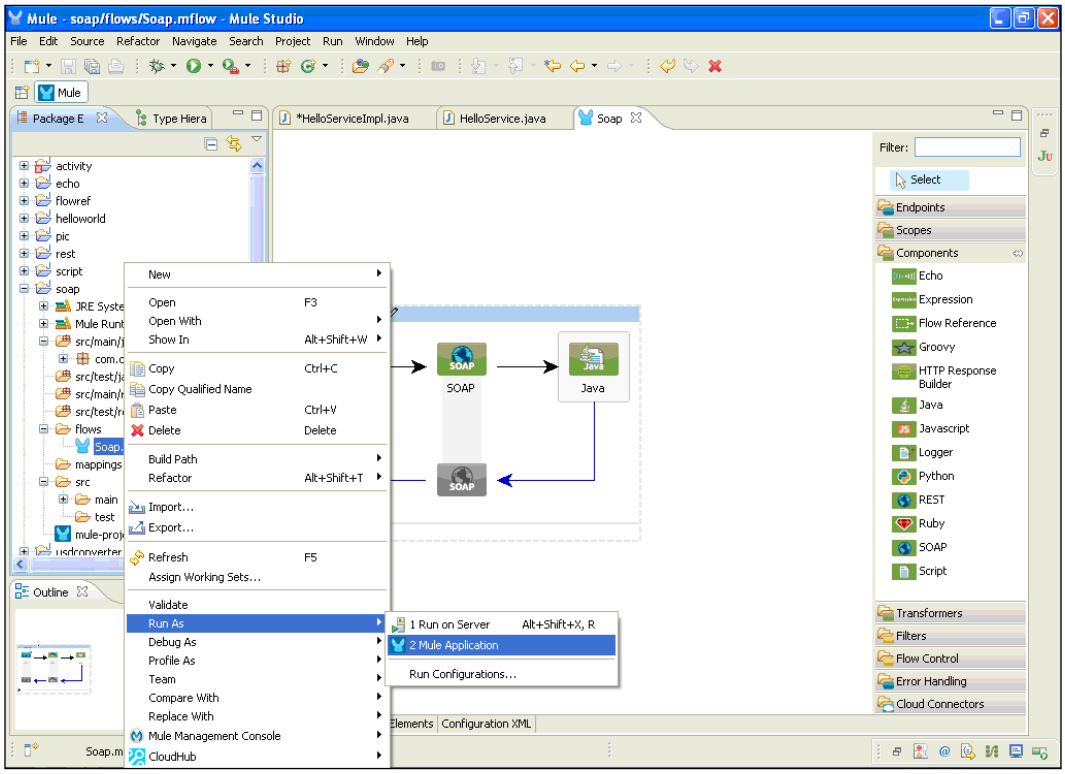
7. Double-click on the **Java** component to configure it. Here, we import the `HelloServiceImpl` class that was created before.



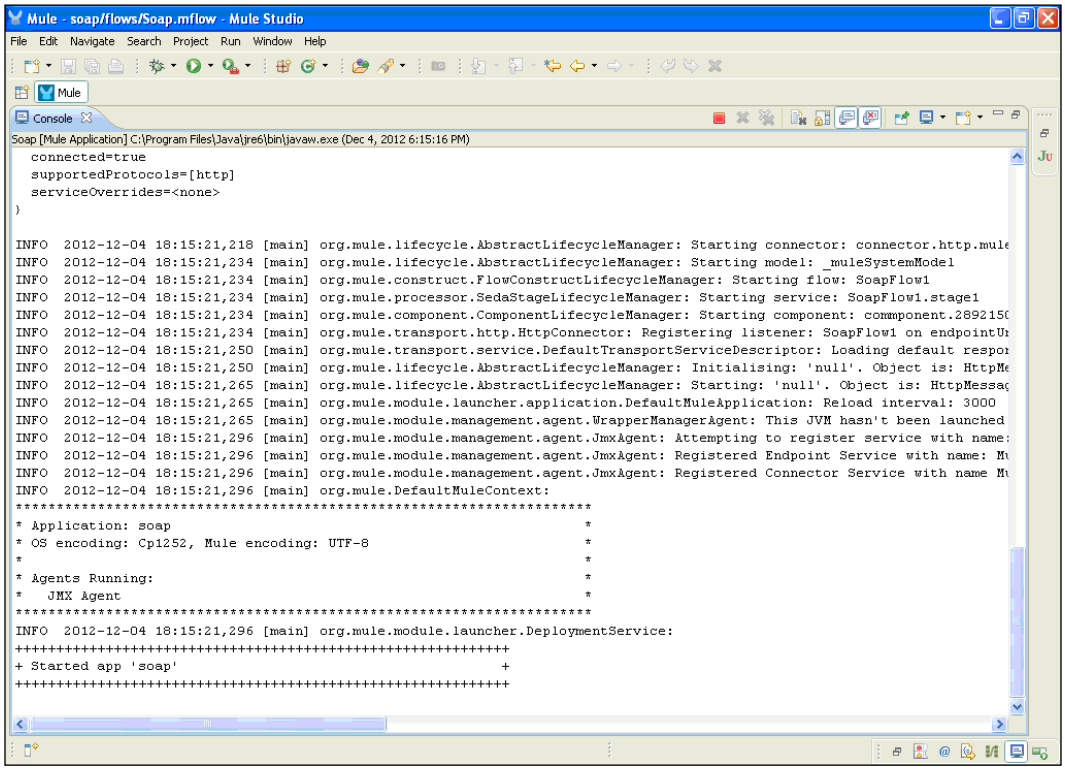
How it works...

To deploy your application, right-click on your `.mflow` file and deploy your Mule application by performing the following steps:

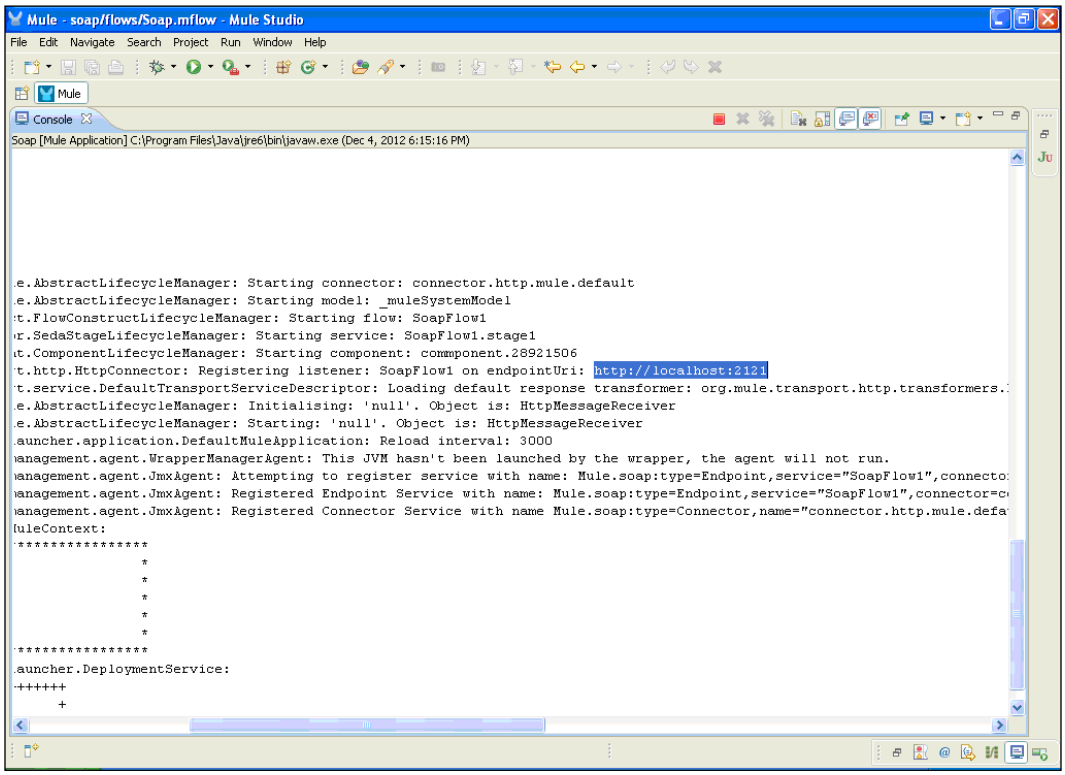
1. If you haven't saved your application code, do save it. After saving your project, right-click on the `Echo.mflow` file and go to **Run As** | **Mule Application**.



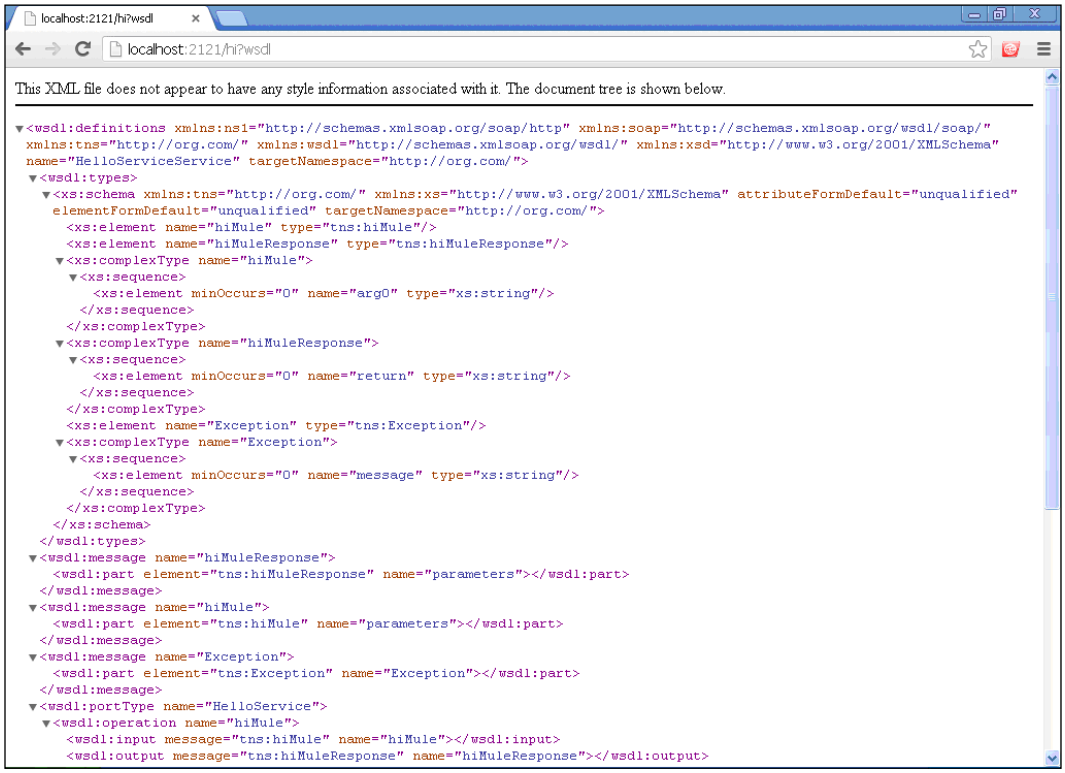
2. If your application code is successfully deployed, you will see the message `Started app 'helloworld'` on the console.



3. Copy the URL `http://localhost:2121/` and paste it on your browser.



4. To see the output, paste the URL on your browser and type in `/hi?wsdl`; here, `wsdl` stands for **Web Services Description Language**.



3

Using Message Property, Processors, and Sources

In this chapter, we will cover the following topics:

- Understanding components
- Understanding message sources
- Using message processors to control the message flow
- Understanding message property scopes

Introduction

A message source is the Endpoint where the Mule inbound elements receive messages. Message sources can be Inbound Endpoints, polls, or the custom message receiver. All of these Endpoints receive messages and depend on their corresponding message processors for further execution. Mule has transformers, filters, components, Routers, and other message-processing elements to be used and nested freely as required. They all implement a common message processor interface and can be used interchangeably.

Understanding components

The **Script** component is used for executing different types of scripts such as Ruby, Java, JavaScript, Python, and Groovy. We execute a script that receives a response from the client before the payload is processed. The Script component also provides the option of integrating custom script into a flow.

Getting ready

When using a Script component, the developer must select a script engine that is compatible with the language used to create the custom script.

The **Java** component is used to create custom Java code that is executed when the component receives a message. The **Java** component (whose icon is shown in the following screenshot) can be used to enhance the functionality. To configure the Java component, import a custom Java class; additionally, you have to configure the **Spring** and **Singleton** objects. The Singleton object's purpose is to control object creation and limiting the number to one, while allowing the flexibility to create more objects if the situation demands it.

The **Python** component (whose icon is shown in the following screenshot) can be used to configure the Python scripting language. You can write a custom script in Python for an application. You can also add a scripting file inside the `src/main/resources` folder. The Python script is executed at runtime only.

In the **Ruby** component (whose icon is shown in the following screenshot), you have to integrate the custom script language or you can add a script file inside the `src/main/resources` folder.

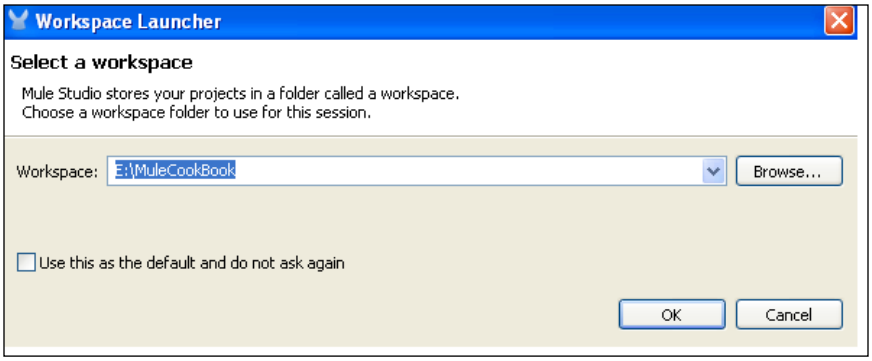
In the **Groovy** component (whose icon is shown in the following screenshot), you have to integrate the custom script language or you can add a script file inside the `src/main/resources` folder.

In the **Javascript** component (whose icon is shown in the following screenshot), you have to integrate a custom script language in that component or you can add a script file inside the `src/main/resources` folder. It also allows the developer to configure interceptors and alter the values or references of particular properties in a script. All the scripting components are configured in a similar way.

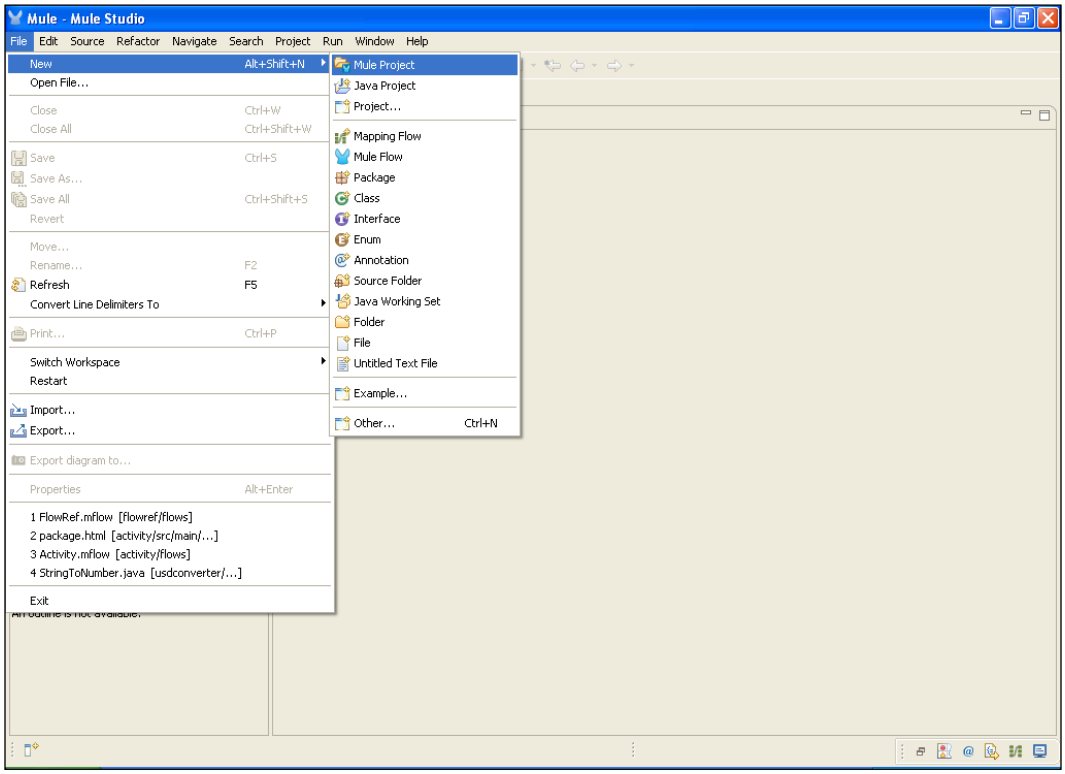
How to do it...

We will now use the **Groovy** component to demonstrate how to change the filename of an image.

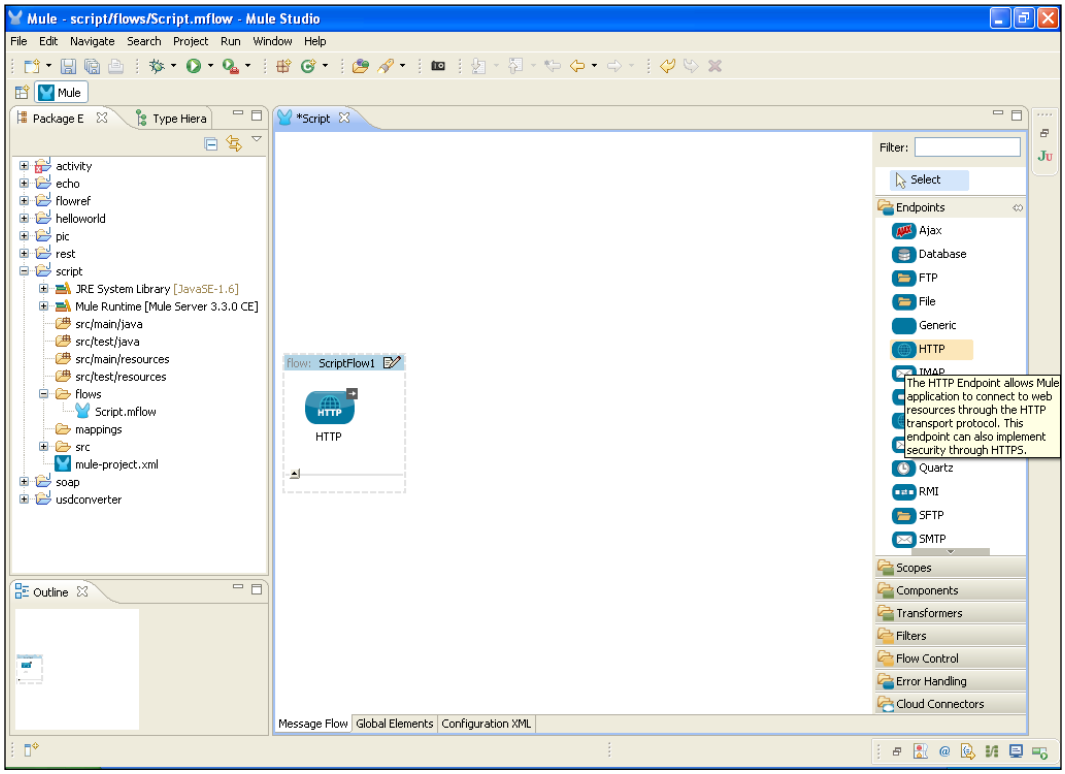
1. Open Mule Studio and enter the name for the workspace. We have to use the **Groovy** component, the **Logger** component, and the **HTTP** Endpoint.



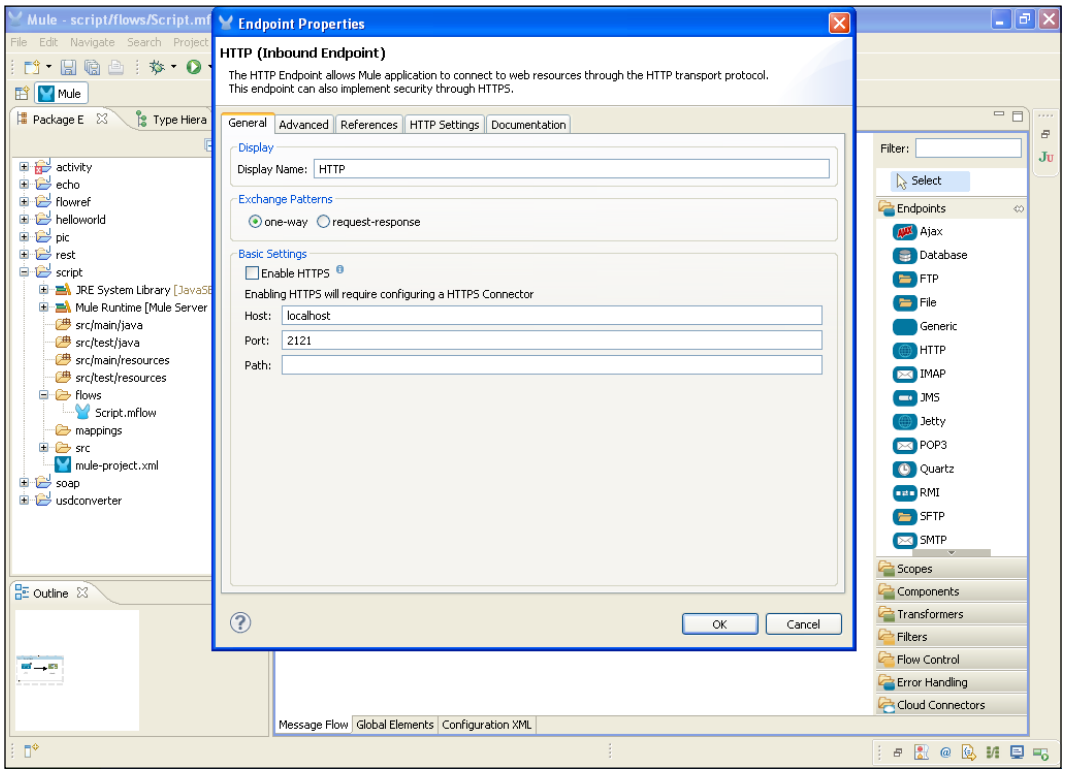
2. To create a new project, go to **File** | **New** | **Mule Project**. Enter the project name as `Script` and click on **Next** and then on **Finish**. Your new project has been created. You can now start implementing it.



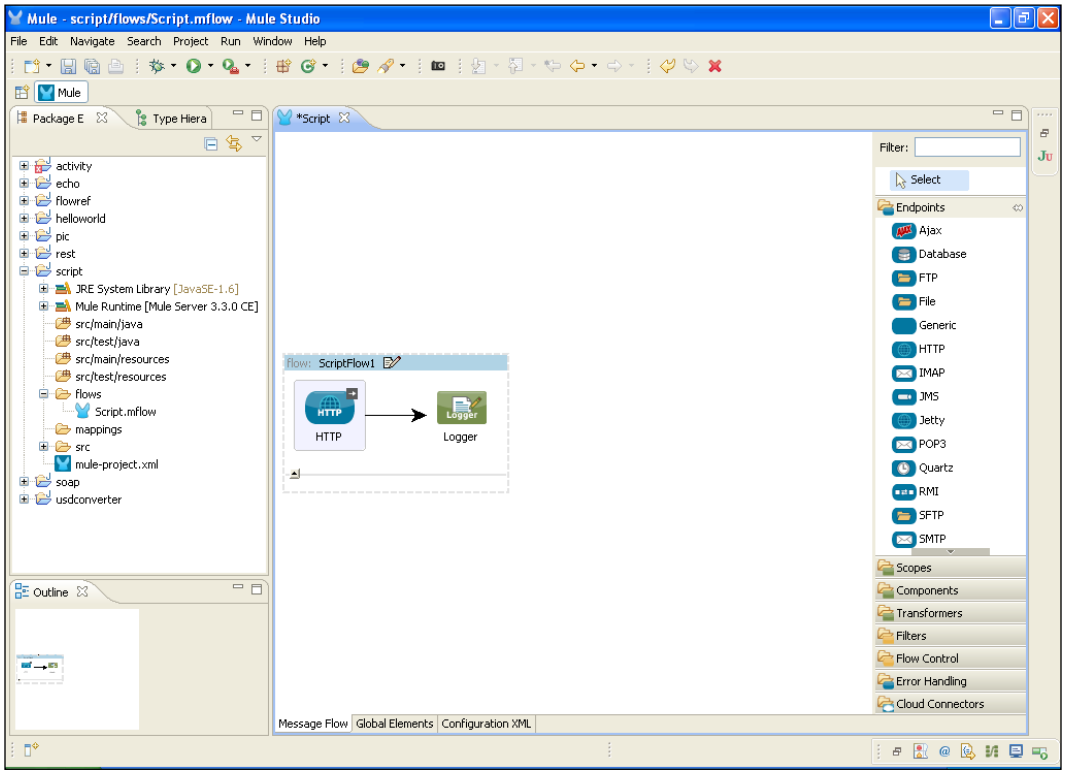
3. Go to the file `Script.mflow`. Drag the **HTTP** Endpoint from the palette and drop it on the canvas. You will now have to configure the **HTTP** Endpoint.



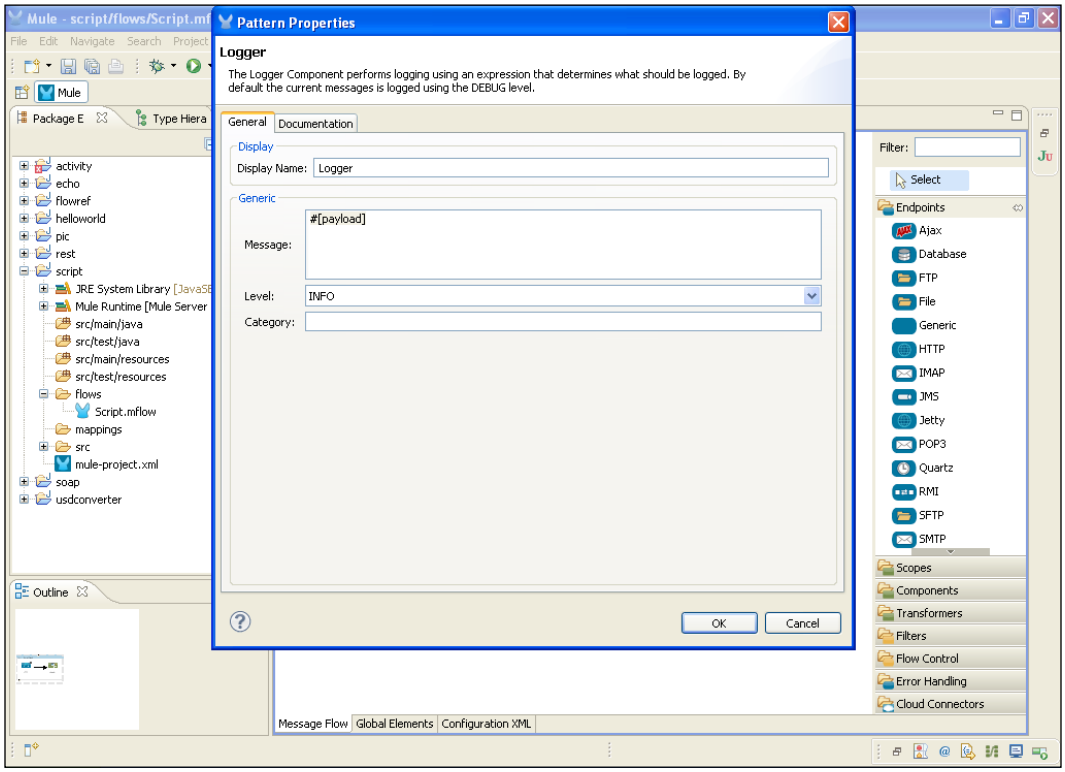
4. Double-click on the **HTTP** Endpoint to configure it. You will see the **Host:** and **Port:** fields. You can change the hostname if you want to. In this example, we have used `localhost`. If you want to change the port number, you can do that as well. By default, the Mule server takes up the port number 8081. Here we have used port number **2121**. We have used only the one-way exchange pattern, so we send requests from the **HTTP** component and responses will come from the **Script** component.



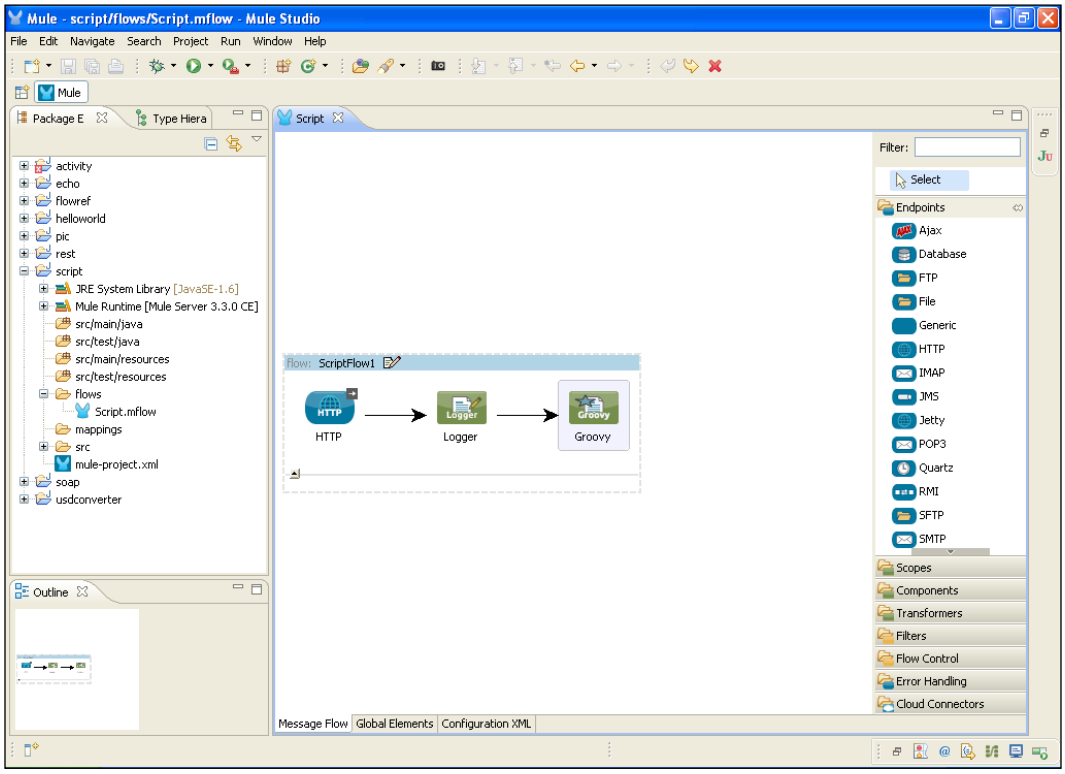
5. Drag the **Logger** component from the palette and drop it on the canvas. Configure the **Logger** component.



6. Double-click on the **Logger** component to configure it. You will see the **Message:** textbox. Enter the payload expression in it. After configuring this, click on the **OK** button and you will see a screen similar to the following screenshot:

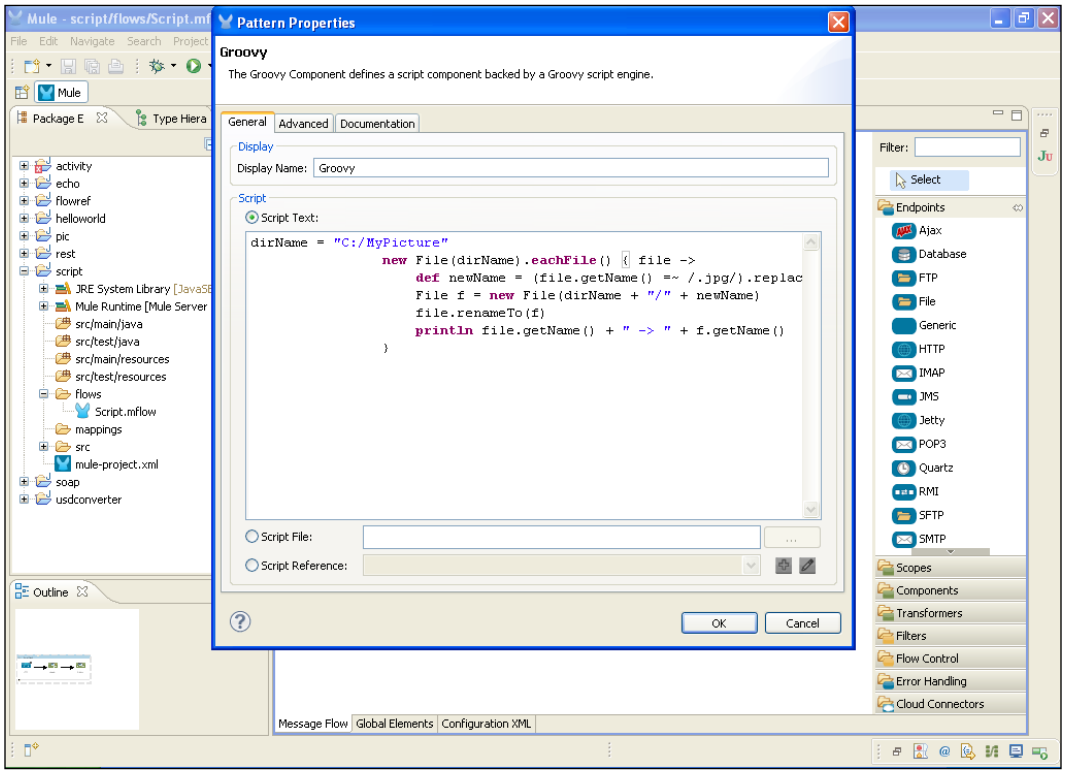


7. Drag the **Groovy** component from the palette and drop it on the canvas. Configure the **Groovy** component.



8. Double-click on the **Groovy** component to configure it. Here we have written a simple Groovy script. With that script, we can change the image filename that is located in the location `C:/MyPicture`. You can also add a script file inside the `src/main/resources` folder.

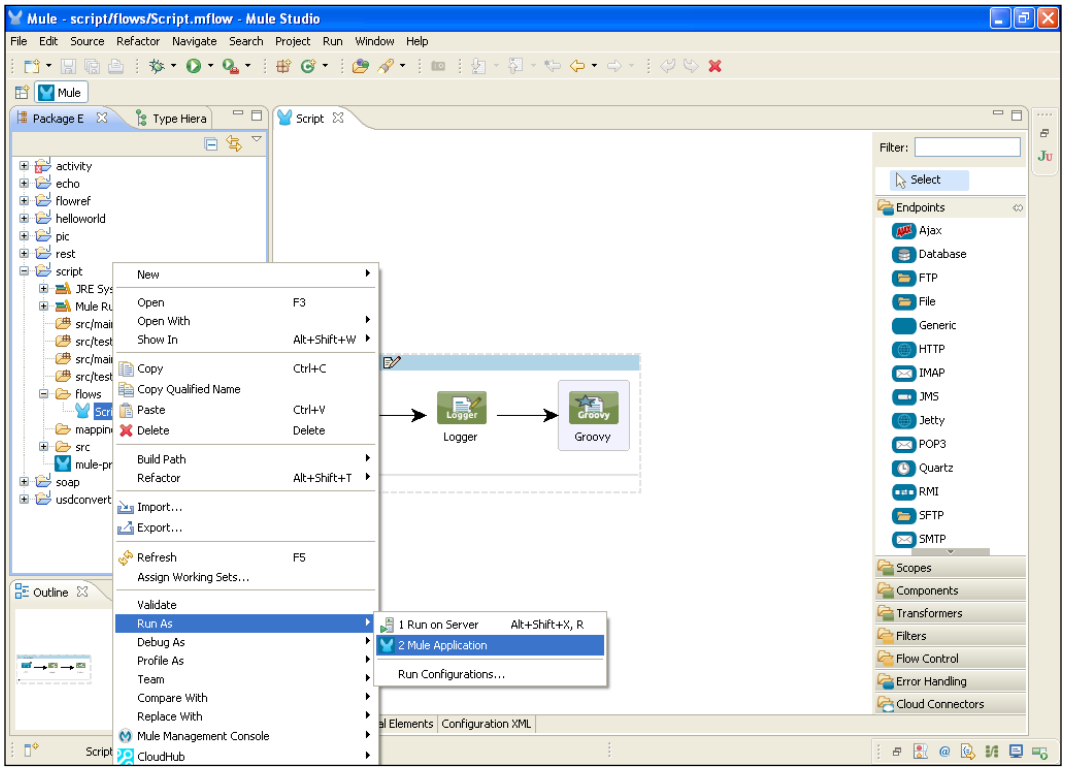
```
dirName = "C:/MyPicture"
new File(dirName).eachFile() { file ->
defnewName = (file.getName() =~ /.jpg/).replaceFirst("infocom.
jpg")
          File f = new File(dirName + "/" + newName)
file.renameTo(f)
println file.getName() + " -> " + f.getName() }
```



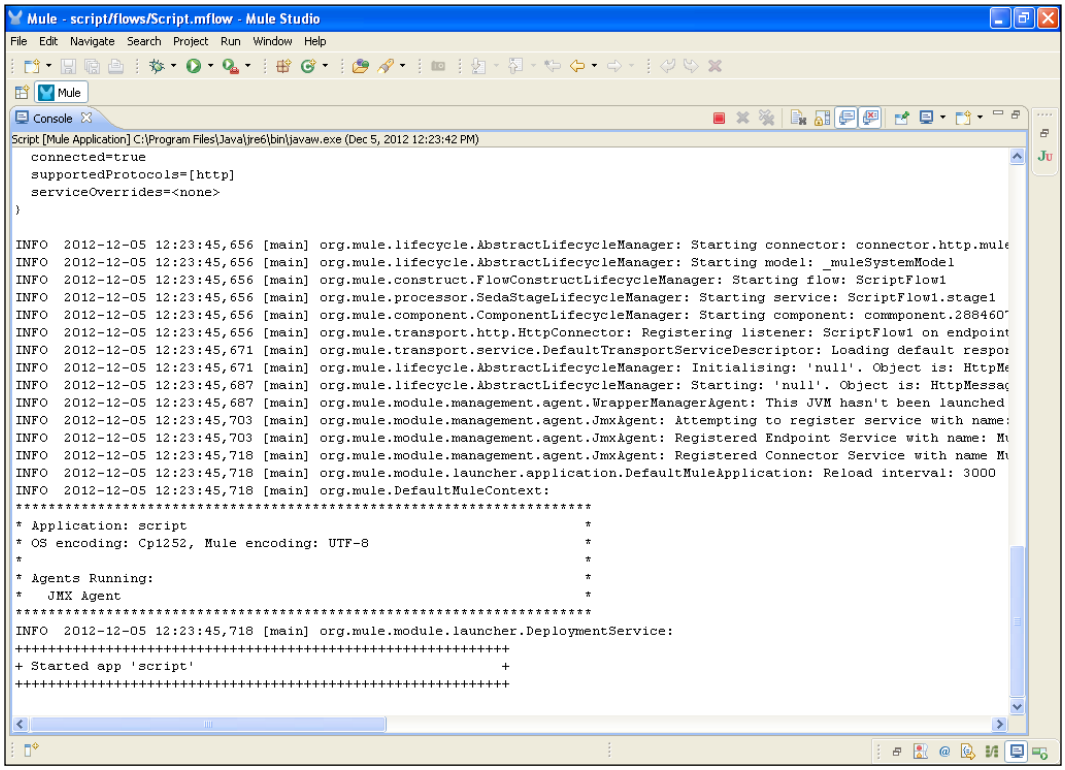
How it works...

In this section, you will learn how you can deploy your application using Mule Studio. After deploying this application, you will see how it works.

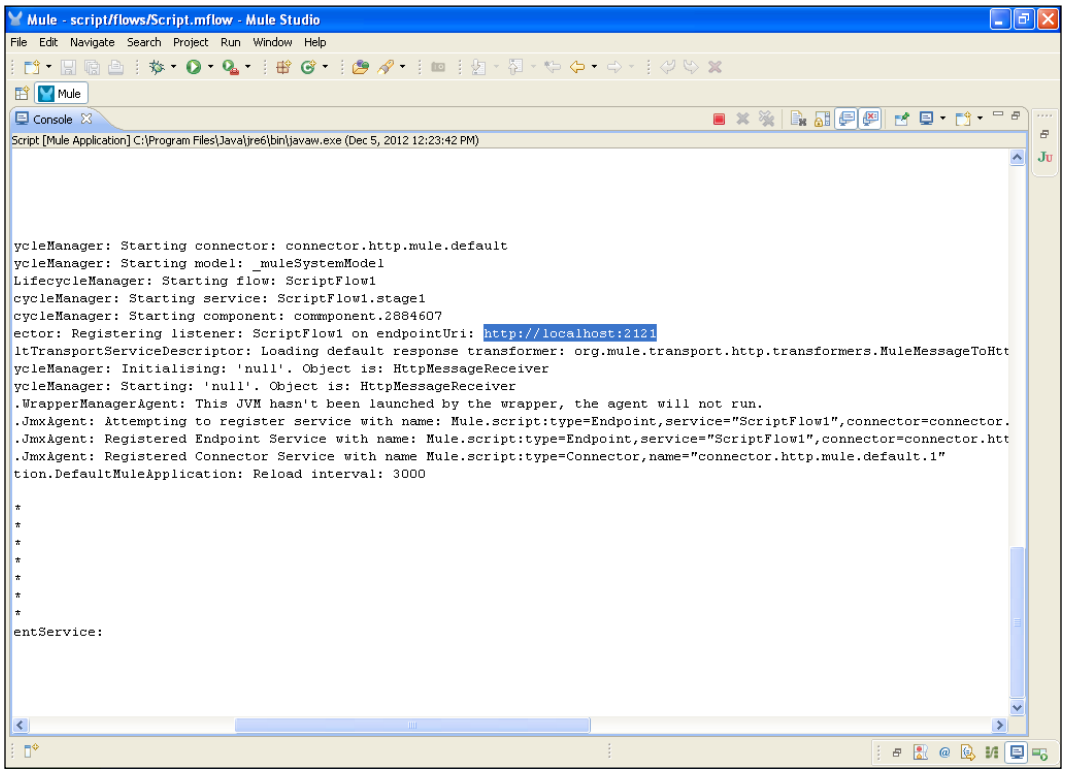
1. If you haven't saved your application code, do save it. To deploy the application code in the Mule server, go to **Run As** | **Mule Application**; the Mule server will deploy your application.



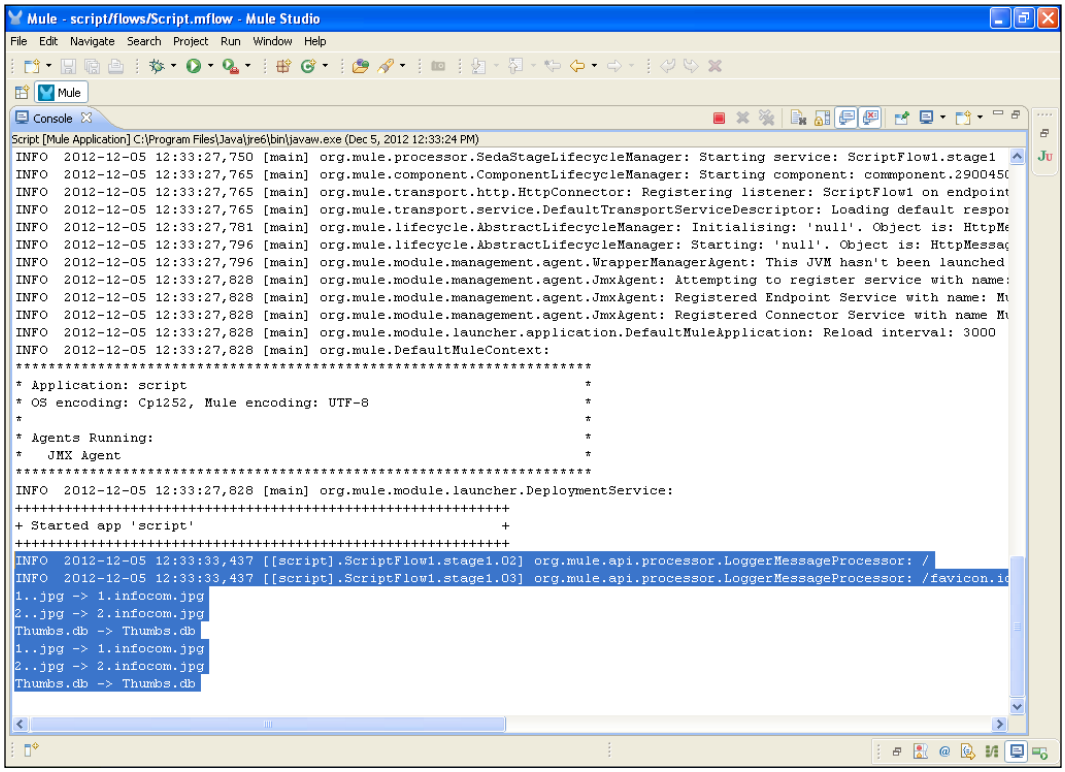
2. If your application code is successfully deployed, you will see the following screenshot as the output on your console:



3. Copy the URL `http://localhost:2121/` and paste it on your browser.



4. Here you can see the output on the console, and you can see that we have changed the filename using the **Script** component.



Understanding message sources

A message source generally receives or generates new messages to be processed by Mule. Once a message has been received from a message source, it is processed by Mule using one or more message processors.

Getting ready

We can use message processors in the flow in two ways: the one-way exchange pattern and the request-response pattern. Setting the exchange pattern of a message source to one-way enables asynchronous processing of a flow, while setting the exchange pattern of a message source to request-response enables synchronous processing of a flow.

How to do it...

As mentioned in the introduction to this chapter, a message source is the Endpoint where Mule inbound elements receive messages. Message sources can be Inbound Endpoints, polls, or the custom message receiver. All of these Endpoints receive messages and depend on their corresponding message processors for further execution. Mule supports the following types of message sources:

- Inbound Endpoints
- Polls
- Custom message sources

Inbound Endpoints

Inbound Endpoints receive new messages from channels. The following is an example of a code snippet that configures an Inbound Endpoint for a flow:

```
<flow name="MessageSources">
<http:inbound-endpoint address="http://localhost:8080/endpoint"
exchange-pattern="one-way"/>
<jms:outbound-endpoint queue="messages"/>
</flow>
```

This flow indicates how to asynchronously bridge an HTTP request to the JMS.

Polls

Instead of using an Inbound Endpoint, you can poll any message processor and use the result as the source of your flow. Regularity can be configured with milliseconds as the unit or else the default of one second can be used. To arrange polling, use the `<poll>` section. The following is an example of a code snippet that configures a poll for a flow:

```
<flow name="PollExample">
<poll frequency="500">
<http:outbound-endpoint host="localhost"port="4343"/>
</poll>
<processor ref=""/>
<processor ref=""/>
</flow>
```

Custom message sources

Custom message sources are used to restore any Inbound Endpoint in a flow. You organize the custom message source using the `<custom-source>` element. You can further organize the routine message source using Spring bean properties. The following is an example of a code snippet that configures a custom message source for a flow:

```
<flow name="MyCustomMessage">
<custom-source class="com.org.cookbook.Message">
<spring:property name="threads" value="100"/>
</custom-source>
<vm:outbound-endpoint path="output" exchange-pattern="one-way"/>
</flow>
```

How it works...

You can use message processors in a flow. A message source receives or generates new messages to be processed by Mule.

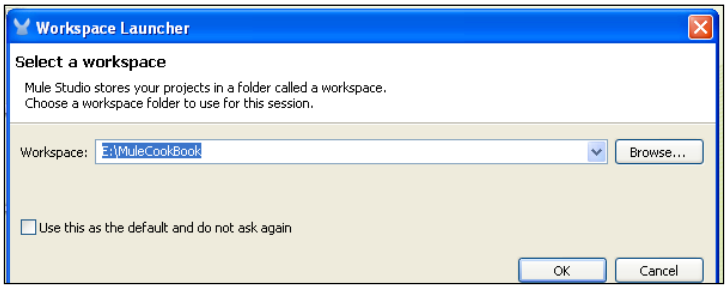
Using message processors to control the message flow

A message processor is the basic building block of all elements in Mule. These blocks can be glued together to create Mule flows. The message processor is a necessary building block for any project in Mule. You will often need to perform some business logic as part of your flow.

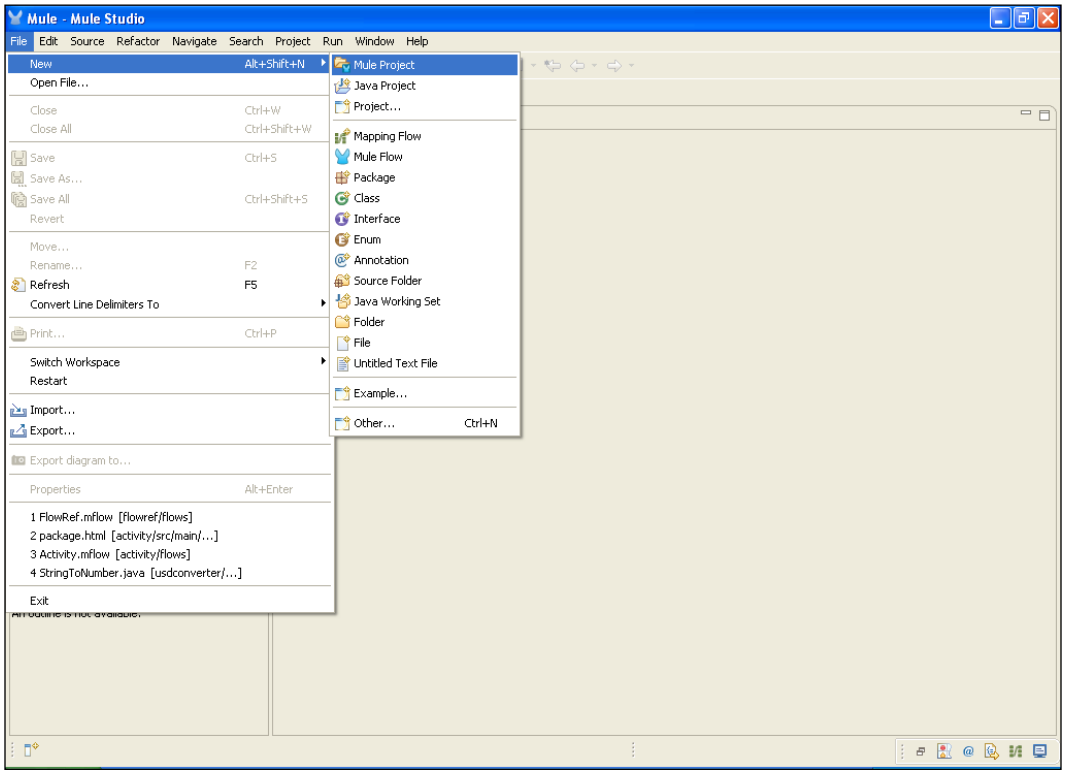
Getting ready

Let's see an example of message processors. In this example, we will see how we can send and receive messages within a flow.

1. We use two components: **STDIO** and **Java**. Open Mule Studio and enter the name of the workspace as shown in the following screenshot:



2. To create a new project, go to **File** | **New** | **Mule Project**. Enter the project name as `StudioConnector` and click on **Next** and then on **Finish**. Your new project has been created now. You are now ready to start the implementation.



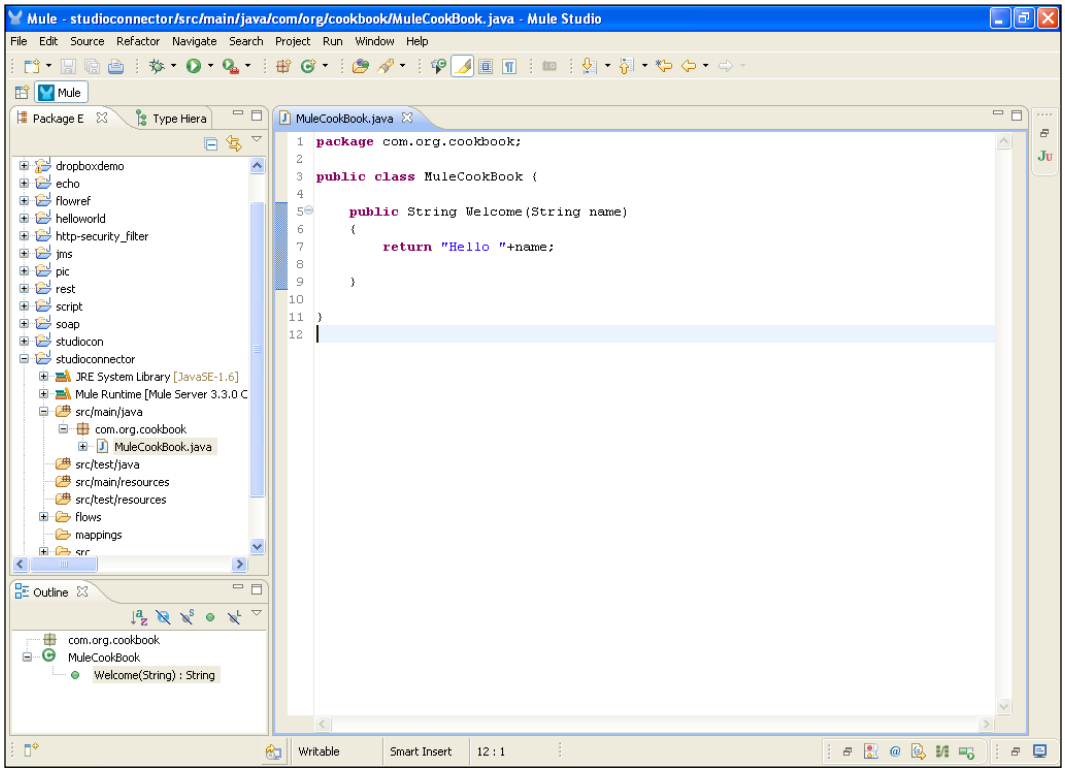
How to do it...

Mule supports components implemented in Java using scripting languages. Message processors are used within flows to control how messages are sent and received within that flow. In this section, we will learn how to use the STDIO connector and how the message processer works.

1. To create a class, go to the folder `src/main/java` and right-click on it. Go to **New** | **Class**. Create a class named `MuleCookBook` under the package `com.org.cookbook`. Here, we have created the `Welcome` method and its return type is set to `String`.

```
public String Welcome(String name)
  {
    return "Hello "+name;
  }
```

The STDIO connector allows reading and writing of streaming data to Java's `System.out` and `System.in` objects for debugging:

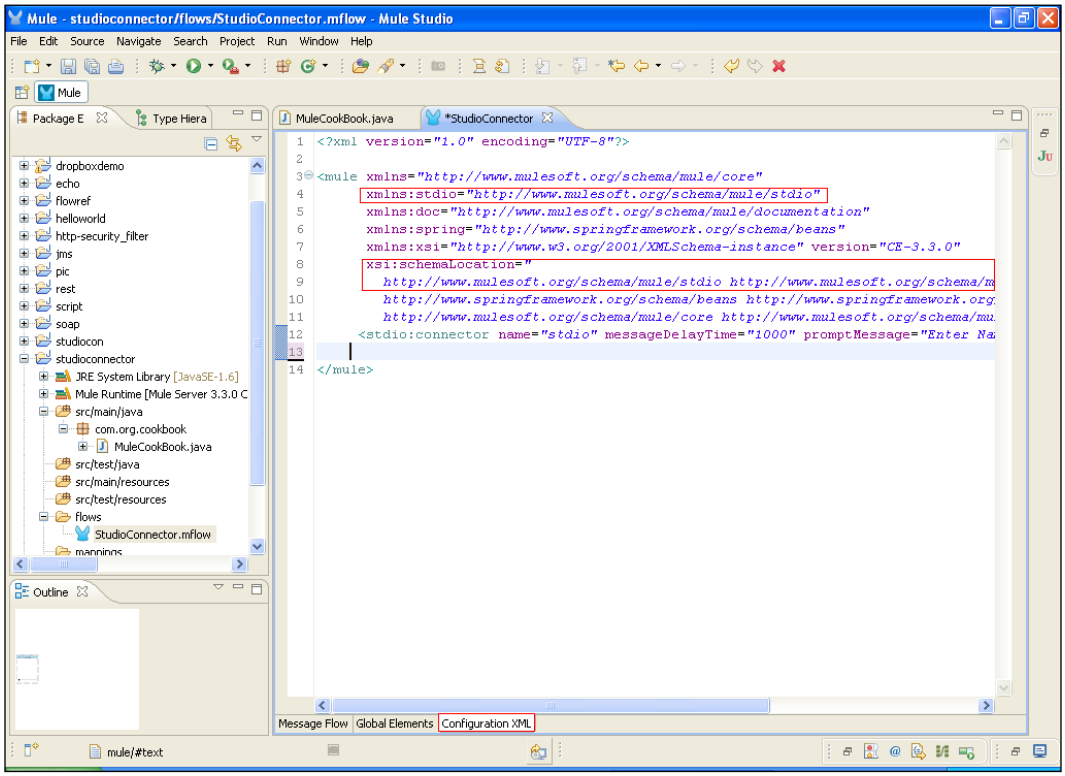


2. Go to the `StudioConnector.mflow` file and click on the **Configuration XML** tab. Here you create the custom `stdio` connector tag. First, insert the namespace and the `schemaLocation` attribute for the STDIO connector; `schemaLocation` is used in the configuration file.

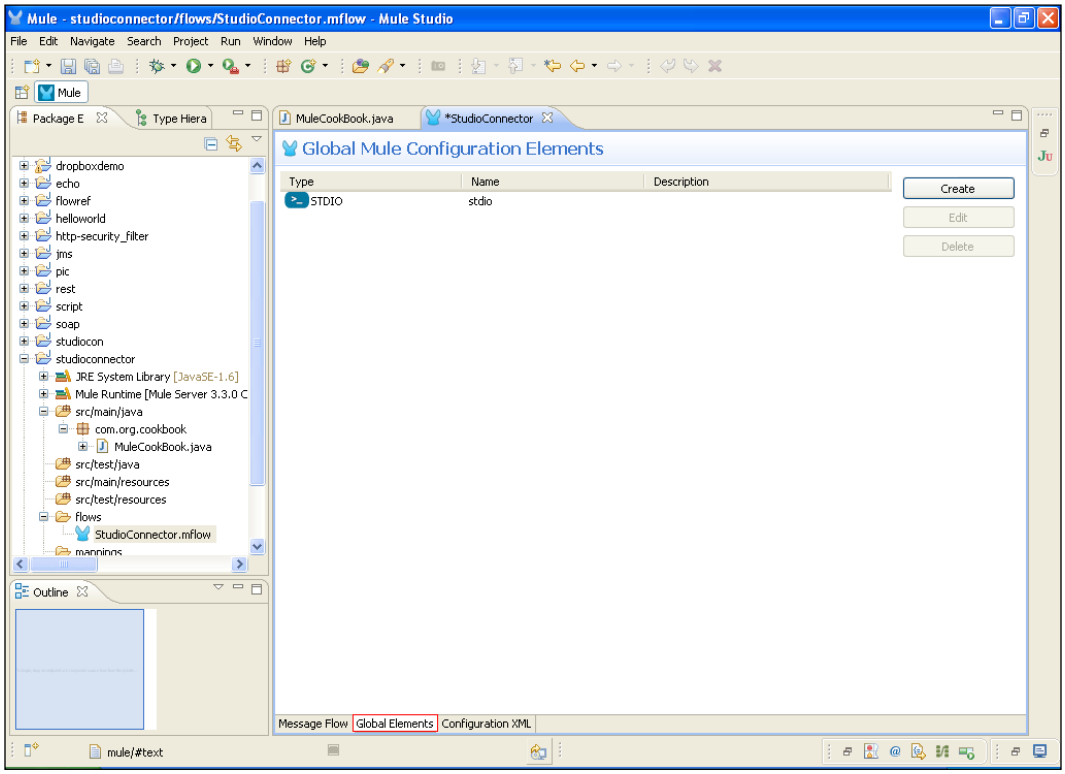
```
xmlns:stdio=http://www.mulesoft.org/schema/mule/stdio
xsi:schemaLocation="
    http://www.mulesoft.org/schema/mule/stdiohttp://www.mulesoft.
org/schema/mule/stdio/current/mule-stdio.xsd
```

You have to create the `<stdio:connector>` tag; inside the configuration file, you have to use three parameters, `name`, `messageDelayTime`, and `promptMessage`. This connector is configured globally, which means you can use this connector in the flow.

```
<stdio:connector name="stdio" messageDelayTime="1000"
promptMessage="Enter Name :" doc:name="STDIO"/>
```



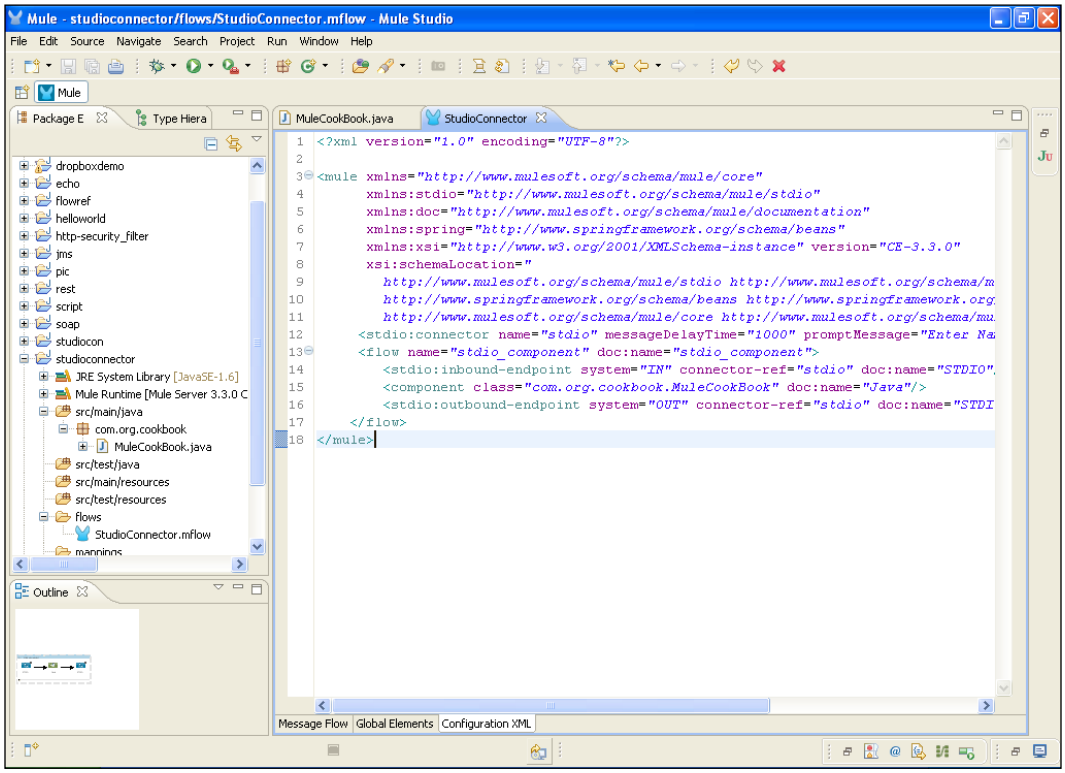
3. Click on the **Global Elements** tab; you can see that the STDIO connector has been created. This will be used in the flow.



4. Here, we have used two STDIO connectors `<stdio:inbound-endpoint>` and `<stdio:outbound-endpoint>`, one is for input and the other is for output. We used the `connector-ref` parameter. Assign the name of the global STDIO connector. Between the two STDIO connectors, we used the Java component tag to import the `Java` class:

```
<flow name="stdio_component" doc:name="stdio_component">
<stdio:inbound-endpoint system="IN" connector-ref="stdio"
doc:name="STDIO"/>
<component class="com.org.cookbook.MuleCookBook" doc:name="Java"/>
<stdio:outbound-endpoint system="OUT" connector-ref="stdio"
doc:name="STDIO"/>
</flow>
```

The following screenshot shows the entire `config.xml` file. In this XML file, we can configure the global STDIO connector.

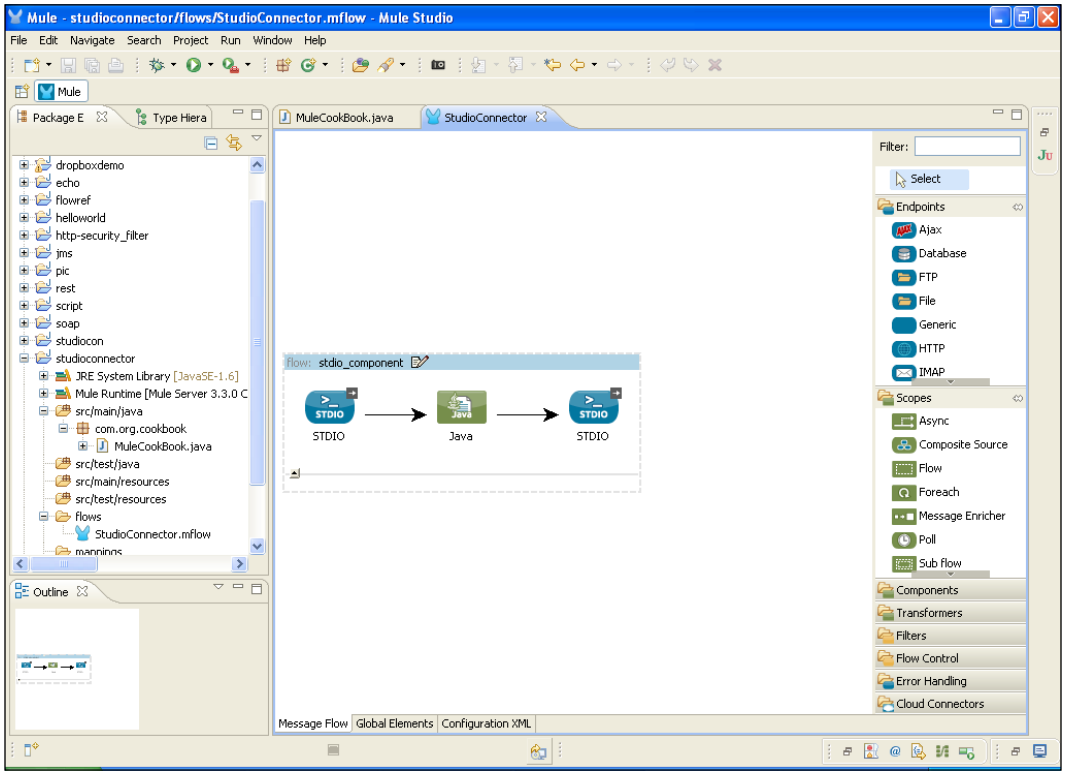


The full configuration file will look like the following code:

```
<?xml version="1.0" encoding="UTF-8"?>

<mulexmlns="http://www.mulesoft.org/schema/mule/core"
xmlns:stdio="http://www.mulesoft.org/schema/mule/stdio"
xmlns:doc="http://www.mulesoft.org/schema/mule/documentation"
xmlns:spring="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
version="CE-3.3.0"
xsi:schemaLocation="
http://www.mulesoft.org/schema/mule/stdio http://www.mulesoft.org/
schema/mule/stdio/current/mule-stdio.xsd
    http://www.springframework.org/schema/beans http://www.
springframework.org/schema/beans/spring-beans-current.xsd
```

```
    http://www.mulesoft.org/schema/mule/core http://www.mulesoft.
org/schema/mule/core/current/mule.xsd ">

<stdio:connector name="stdio" messageDelayTime="1000"
promptMessage="Enter Name :" doc:name="STDIO"/>
<flow name="stdio_component" doc:name="stdio_component">
<stdio:inbound-endpoint system="IN" connector-ref="stdio"
doc:name="STDIO"/>
<component class="com.org.cookbook.MuleCookBook" doc:name="Java"/>
<stdio:outbound-endpoint system="OUT" connector-ref="stdio"
doc:name="STDIO"/>
</flow>
</mule>
```

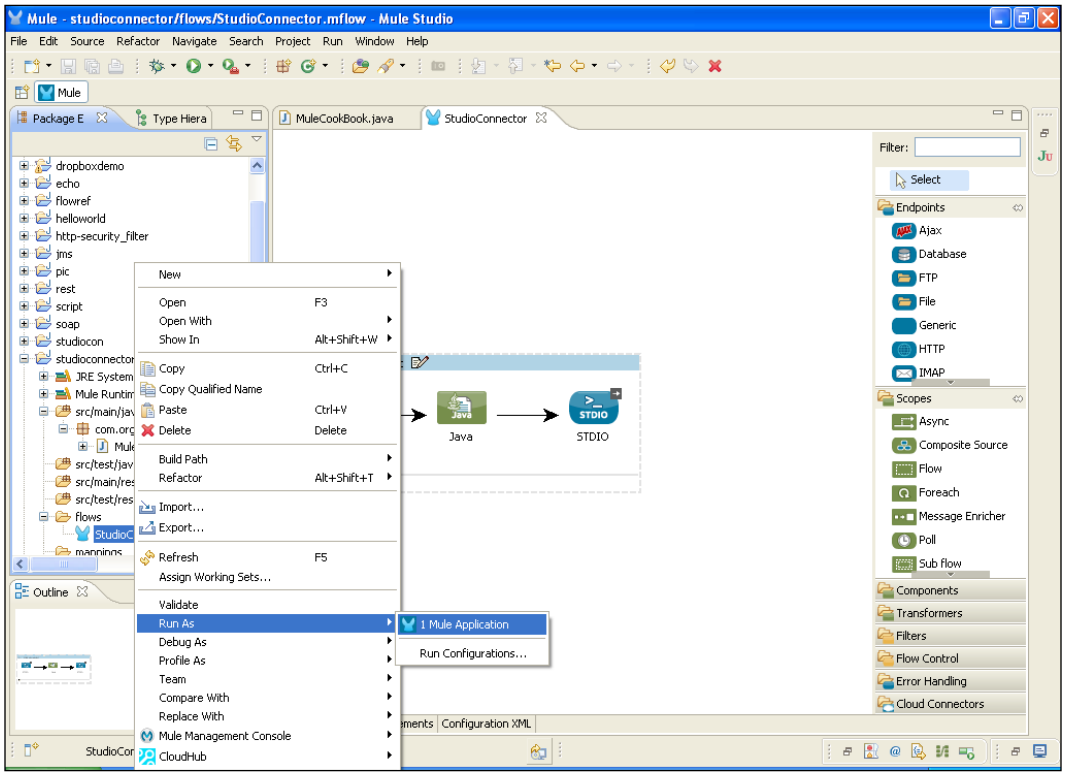
Your graphical flow will look like the following screenshot:



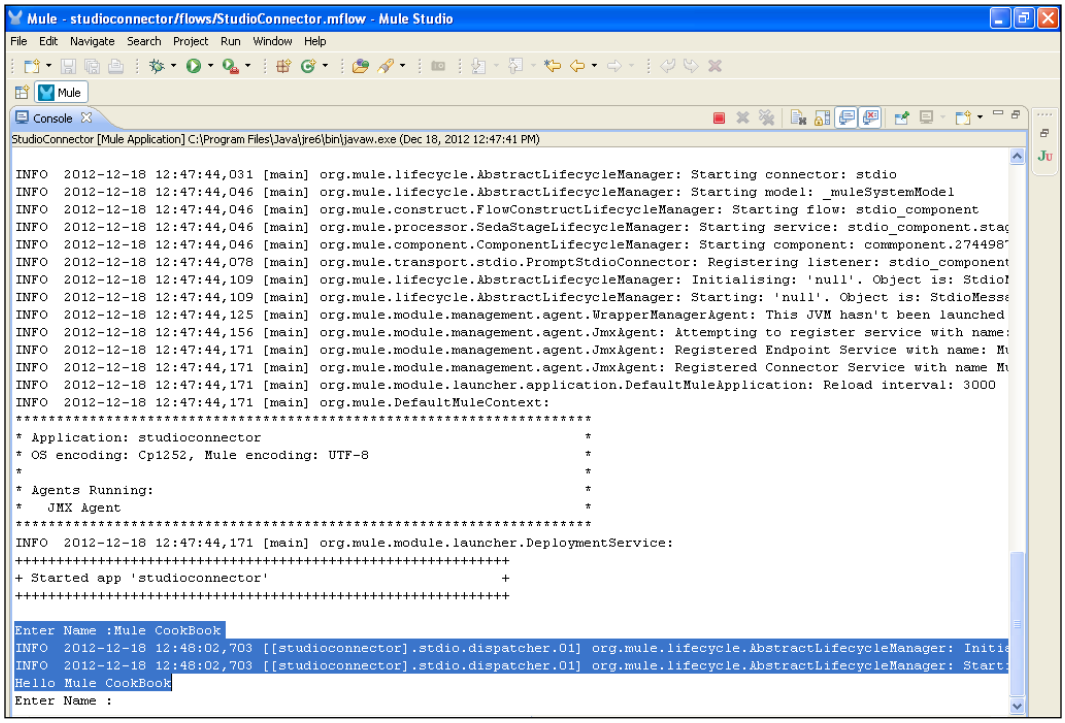
How it works...

To deploy the application, right-click on the `.mflow` file and deploy your Mule application. After deploying the application, you will see how to run that application on the console.

1. To deploy the application code in the Mule server, go to **Run As** | **Mule Application**; the Mule server will deploy the application.



2. Enter `MuleCookBook` on the console; you will see the output on the console. So, in this way, we can send and receive messages with the flow through the STDIO connector.



Understanding message property scopes

The Mule Studio message property transformer has been deprecated and replaced by four new transformers grouped under the title Message and Variable transformers. There are four types of scopes in the message properties, as follows:

- **Inbound**: Inbound properties are those that are received in an Inbound Endpoint or as the response of a call to an Outbound Endpoint. For instance, when an HTTP message is received, each HTTP header will be placed as a Mule message inbound property.

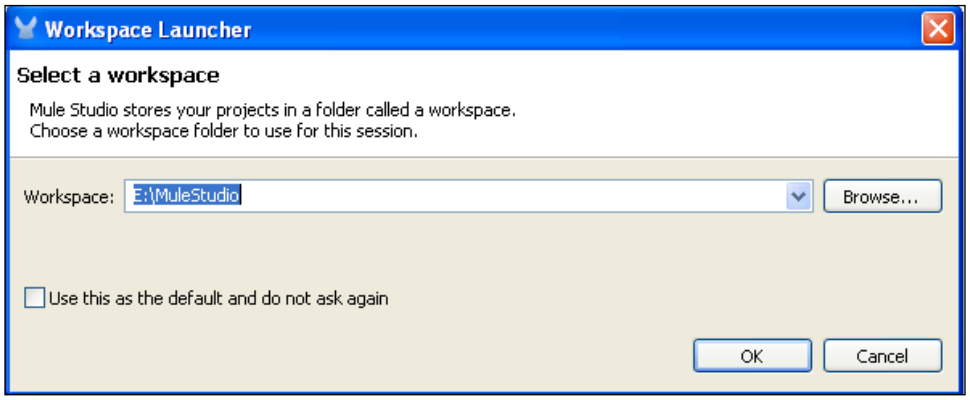
- **Outbound**: Outbound properties are the ones that will be part of the outgoing messages. For instance, if a message with an outbound property content type is sent through HTTP, the `content-type` property will be placed as an HTTP header in the outbound message.
- **Invocation**: Invocation is used mostly internally by Mule for the duration of this service's call, not typically utilized, nor meant for the end user.
- **Session**: Session values are passed from invocation to invocation.
- **Application**: This scope is used when you create two different applications.

Use invocation properties if you need to set a property that you want to then use in the same flow. A typical example of where you will use an invocation property is when you wish to make a variable available for use during a flow. Typically, the invocation property is created for a value that is to be re-used in multiple places in a flow.

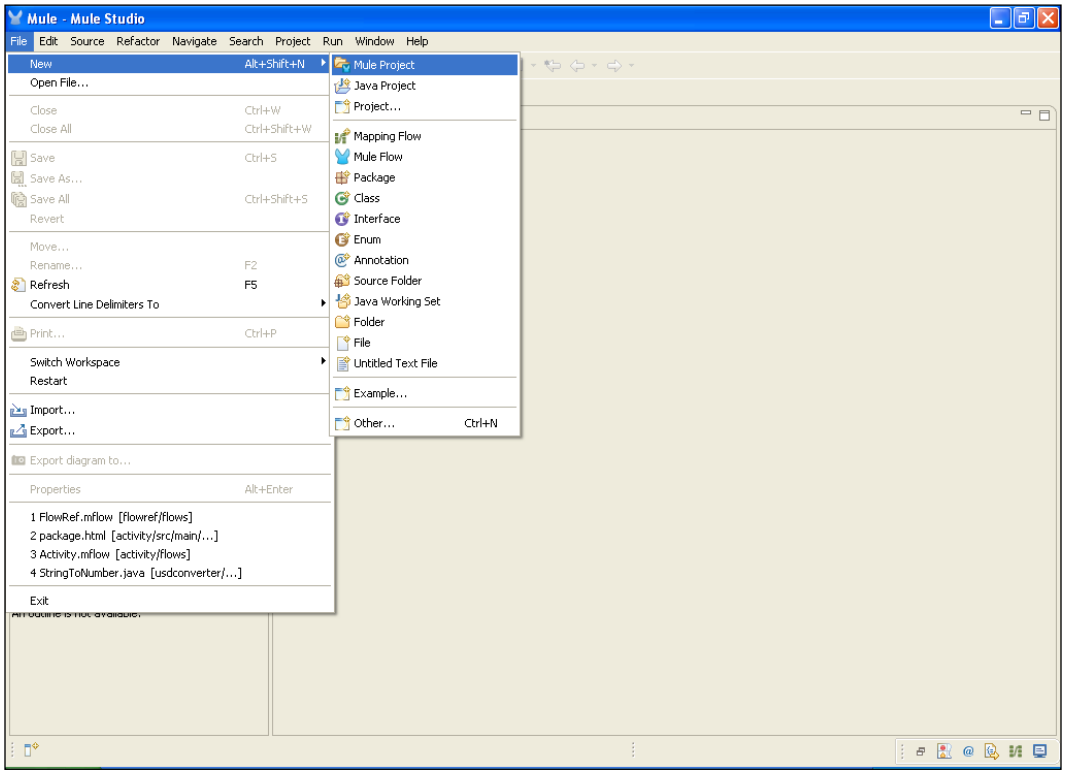
Getting ready

In this section, we will see how to create a custom component. We have to use one HTTP Endpoint and one Java component.

1. Open Mule Studio. Enter a name for the workspace.



2. To create a new project, go to **File** | **New** | **Mule Project**. Enter the project name as `HelloWorld` and click on **Next** and then on **Finish**. Your new project has been created now. We are ready to start the implementation.



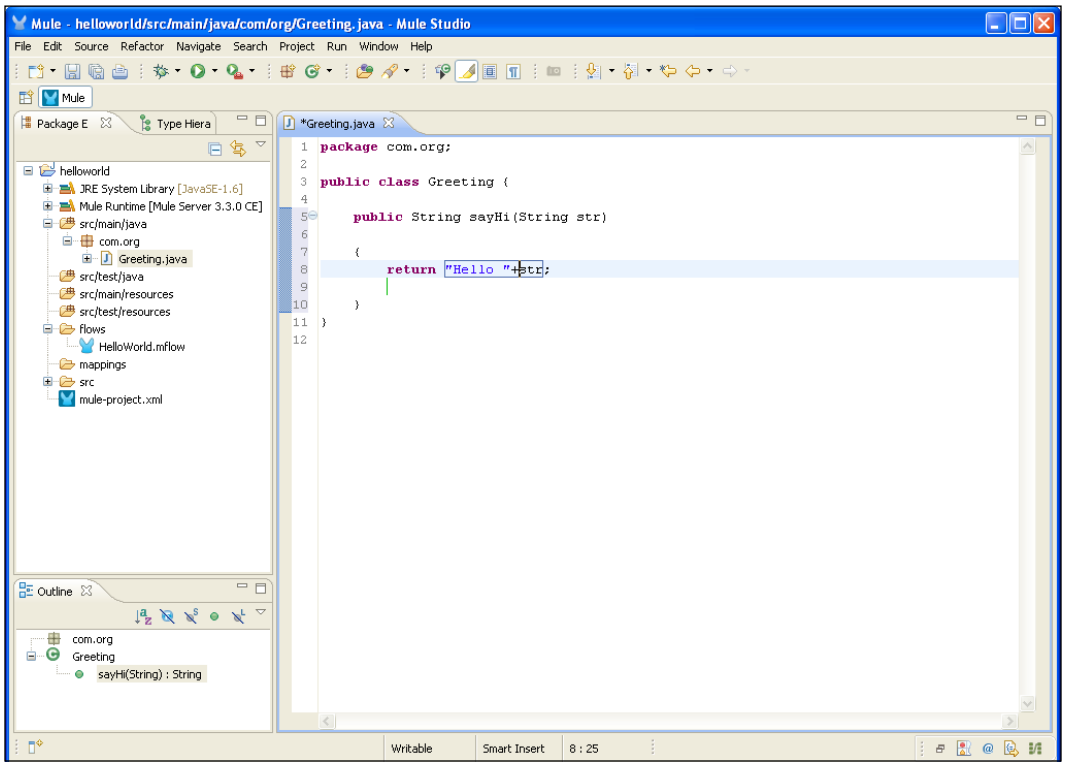
How to do it...

In this section, you will see how to create a custom component using business logic.

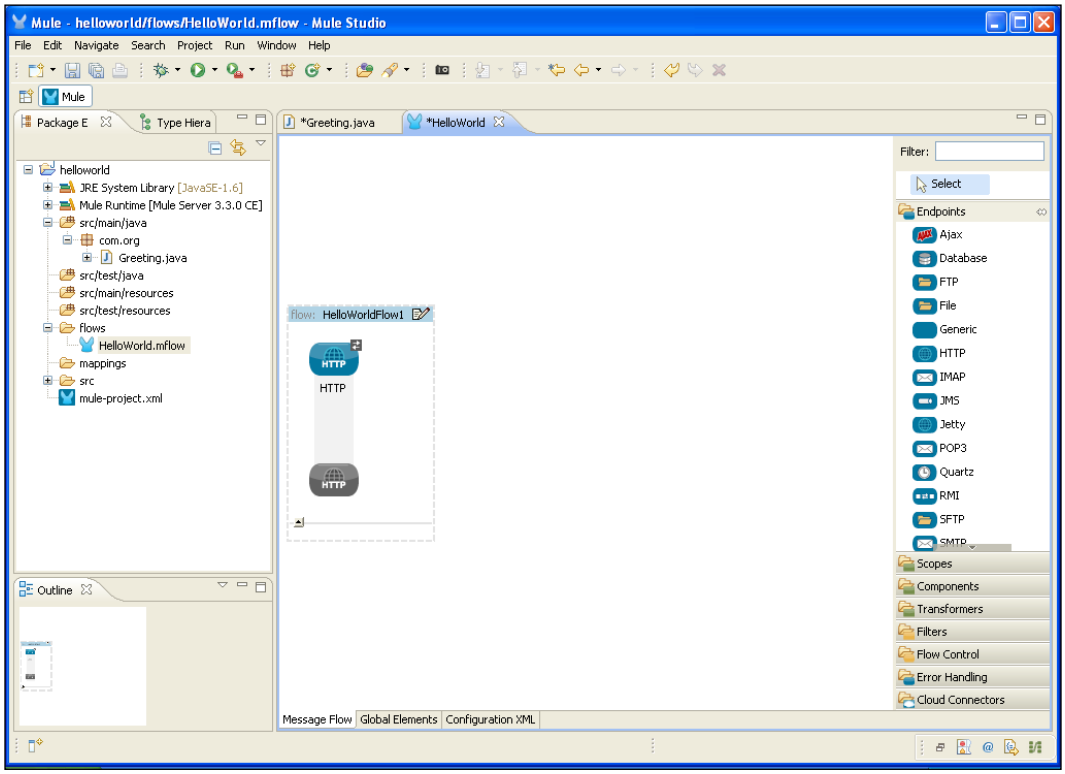
1. To create a class, go to `src/main/java` and right-click on it. Go to **New** | **Class**. Create a class under the package `com.org` and name it `Greeting`; here, we have created the `sayHi()` method and its return type is set to `String`.

```
public String sayHi(String str)
  {
    return "Hello "+str;
  }
```

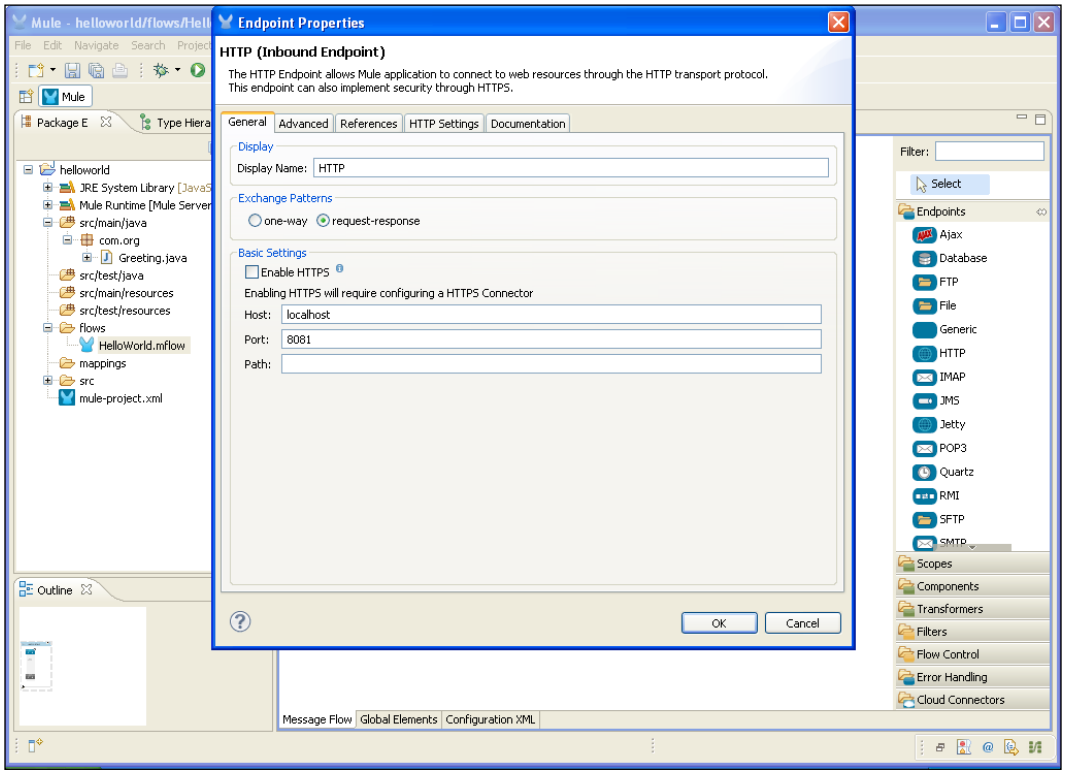
You can see the creation of this method in the following screenshot:



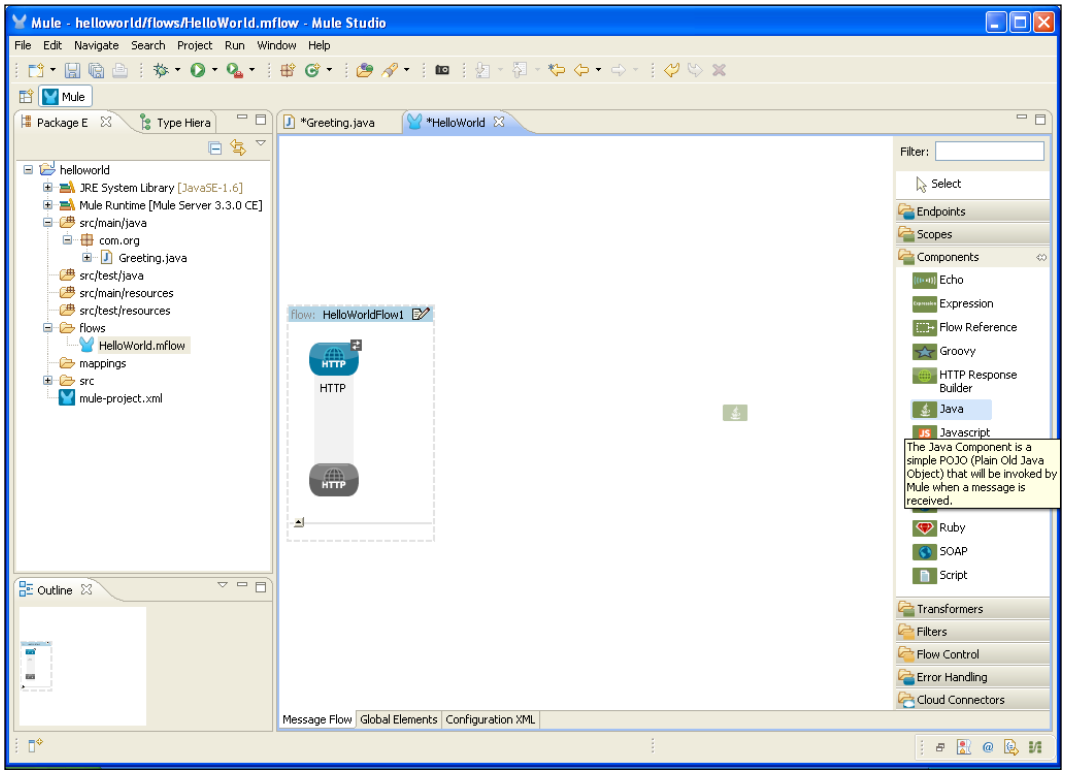
2. To create a flow, go to the `Greeting.mflow` file. In the following screenshot, the central part is called a **canvas**, which has been explained in previous chapters, where we insert graphical elements. On the right-hand side, you can see a group of elements, which is called **palette**. First of all, you have to drag the **HTTP** Endpoint from the palette and drop it on the canvas area.



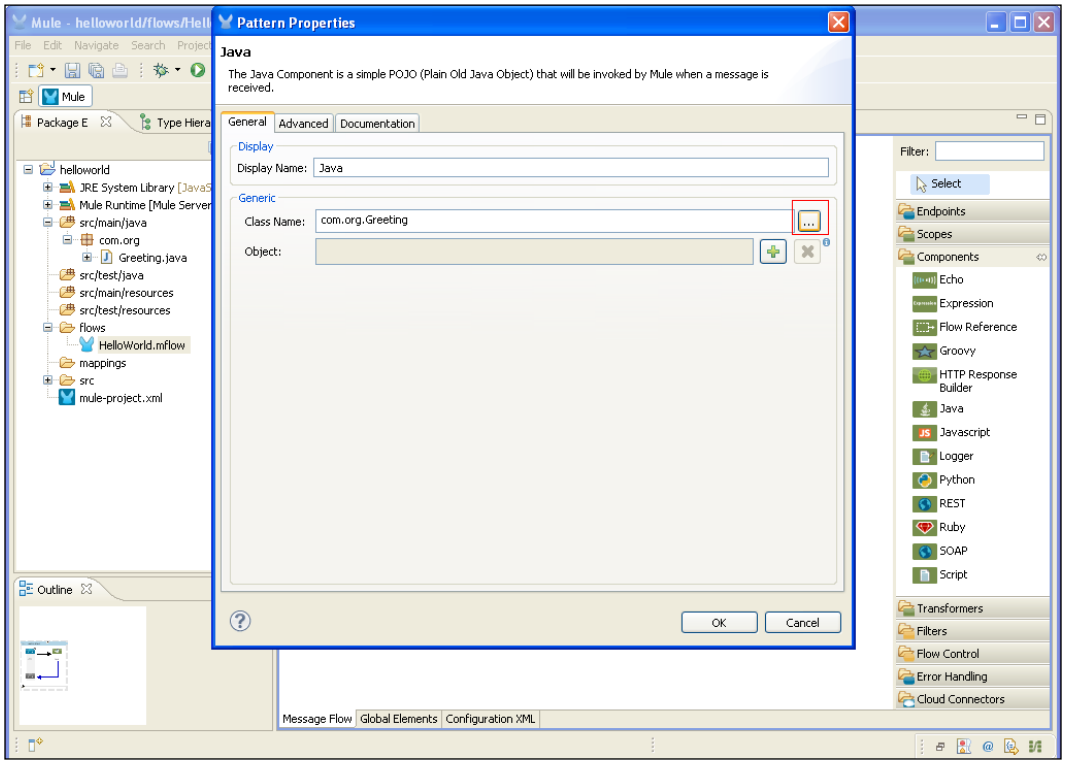
3. Double-click on the **HTTP** Endpoint to configure it. You will see the hostname. Here, the **Host:** and **Port:** fields are mandatory. You have to configure both the attributes.



4. Drag the **Java** component from the palette and drop it on the canvas area. We will store the custom Java class in the **Java** component. Configure the **Java** component. In this example, we have configured the `Greeting` class that we had created earlier.



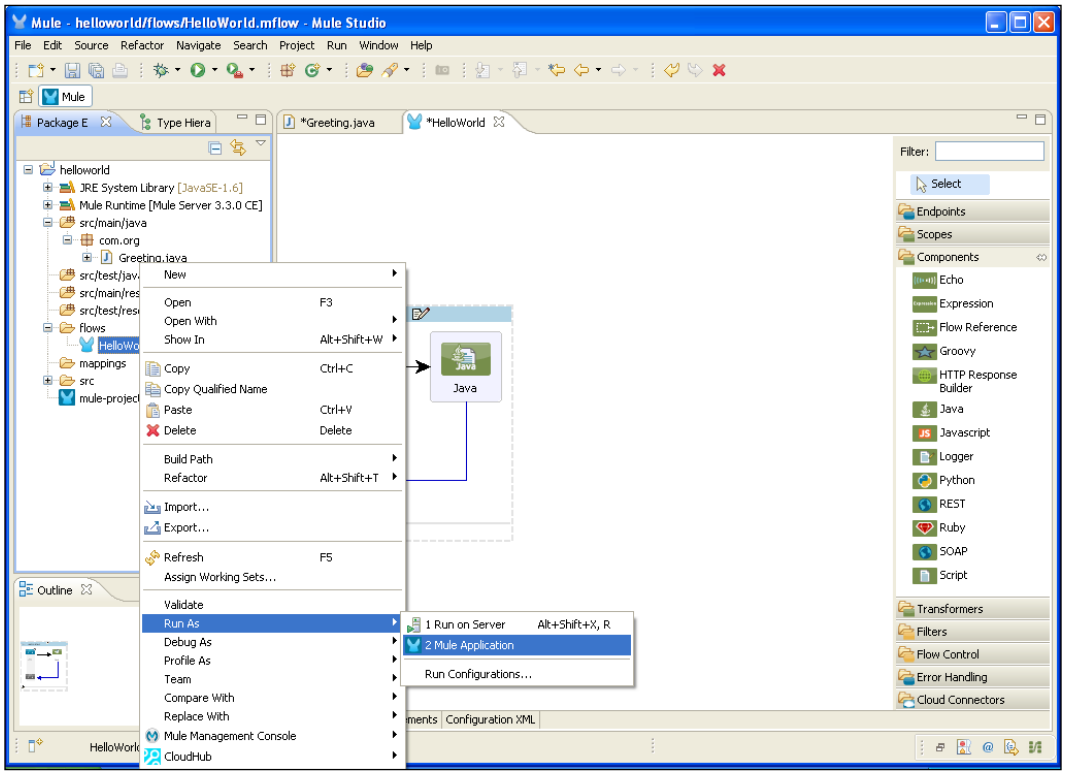
5. Double-click on the **Java** component to configure it and then configure the `Greeting` class.



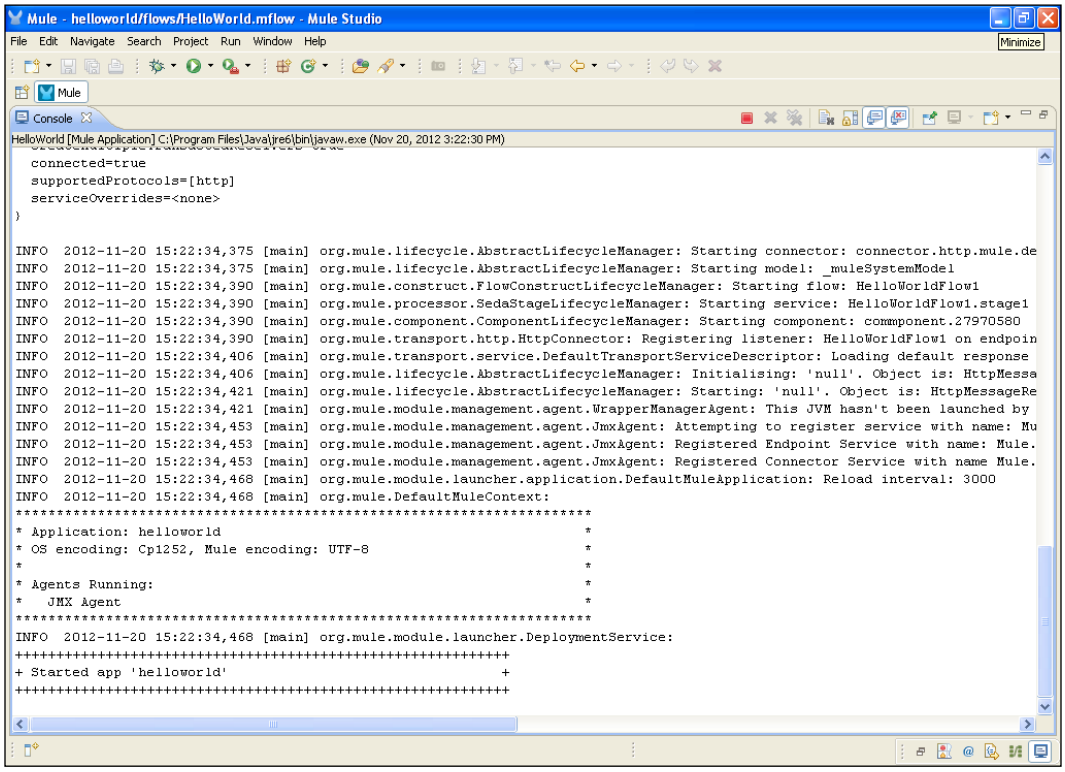
How it works...

To deploy the application, right-click on the `.mflow` file and deploy the Mule application.

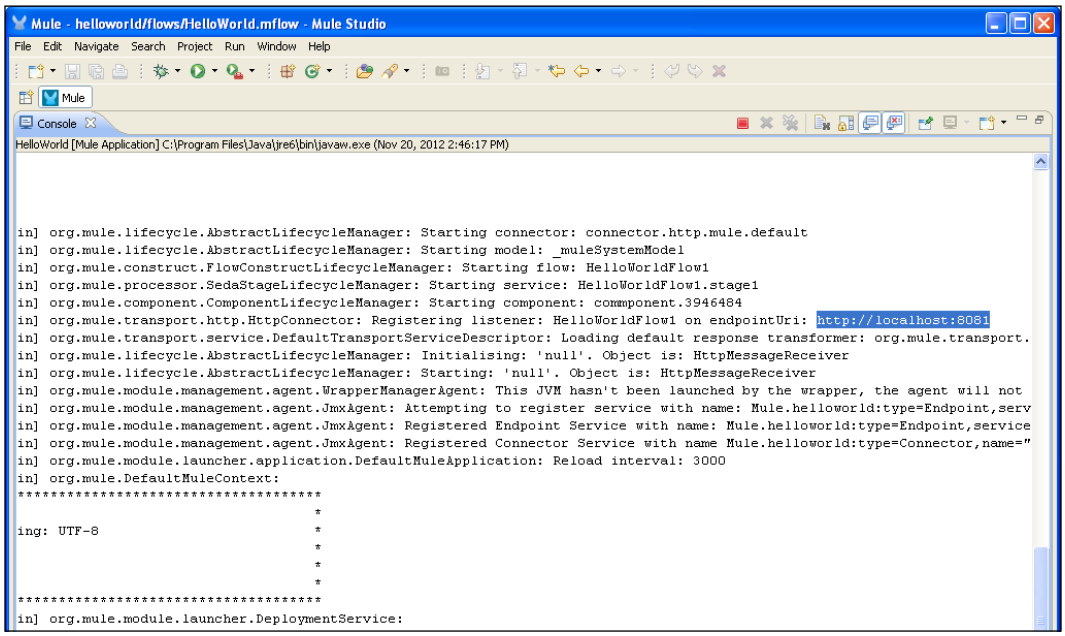
1. To deploy the application code in the Mule server, go to **Run As** | **Mule Application**; the Mule server will deploy your application.



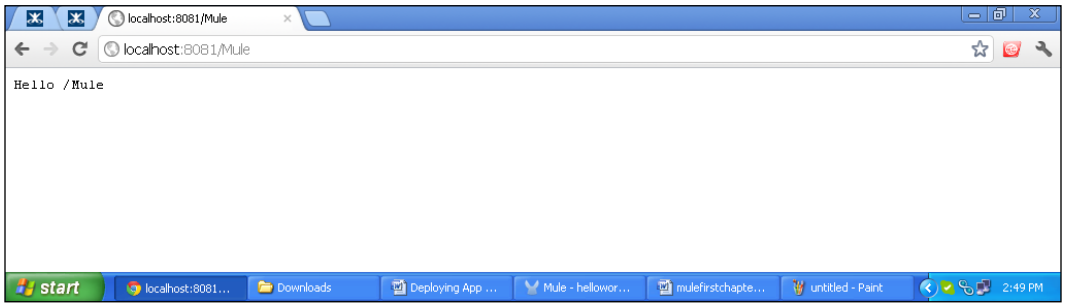
2. If your application code is successfully deployed, you will see the output screen as shown in the following screenshot:



3. Copy the URL `http://localhost:8081` and paste it on your browser.



4. After pasting the URL in your browser, type in `/Mule` (it is the string type in your browser) and you will see the output. In this example, the message hits the component as it passes through the Inbound Endpoint. Mule properties are handled by Mule and move between scopes either implicitly or explicitly.



4
Endpoints

In this chapter, we will cover the different types of endpoints. You will learn the following:

- Configuring the Generic Endpoint
- Configuring the HTTP Endpoint
- Configuring the IMAP Endpoint to retrieve e-mails
- Using the JDBC Endpoint to connect to the database
- Implementing the File Transport channel using the File Endpoint
- Sending messages asynchronously using the AJAX Endpoint
- Using the Servlet Endpoint to listen to events or messages from servlet requests

Introduction

An Endpoint is used for sending and receiving messages through a service. Endpoints can be Inbound or Outbound. An Inbound Endpoint receives messages via its associated transport. Each transport implements its own Inbound Endpoint element. An Outbound Endpoint sends messages via its associated transport. Each transport implements its own Outbound Endpoint element. Different types of Endpoints are available in Mule, such as HTTP, JMS, IMAP, SMTP, and AJAX. We will see how to configure the Endpoint.

Configuring the Generic Endpoint

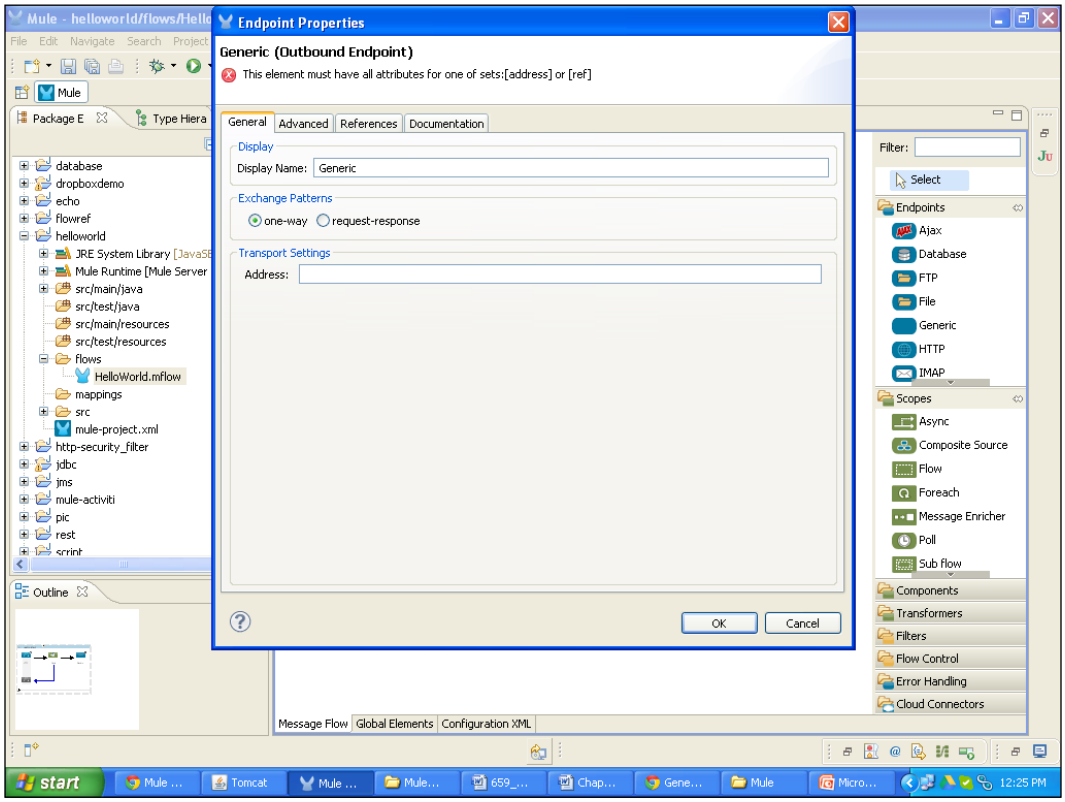
The Generic Endpoint is a string representation of the information you use to configure your Endpoint. This Endpoint is configured by the path specified in the **Address:** field.

Getting ready

The Generic Endpoint employs two types of exchange pattern, one-way and request-response. The Generic Endpoint uses the one-way pattern by default. The VM Endpoint can be created from a Generic Endpoint by specifying the VM transport in this field, for example, `vm://foo`. Similarly, an FTP Endpoint can be created with the Generic Endpoint component by specifying the FTP transport string in the component's **Address:** field under **Transport Settings**; for example: `ftp://user:password@host/directory?connector-ref=myFtpConnector&binary=true`.

How to do it...

After dragging the **Generic** Endpoint from the palette and dropping it onto the canvas, double-click on the Endpoint icon. This invokes the **Endpoint Properties** window for configuration of this Endpoint. The **Address:** field in this window is where you set up the path for the **Generic** Endpoint, as shown in the following screenshot:



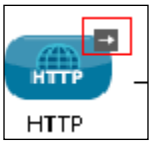
How it works...

The Endpoint will be defined based on Mule expressions or a specific address. For instance, if the address includes HTTP at the beginning, you are configuring an HTTP Endpoint. If the address includes File, you are configuring a File Endpoint.

Configuring the HTTP Endpoint

Mule uses HTTP Endpoints to send and receive requests over the HTTP transport protocol. Configured as either Inbound (also known as message sources) or Outbound, HTTP Endpoints use one of the two message exchange patterns: request-response or one-way. We will see an example of how it works.

The arrows indicate the request-response type of exchange pattern.

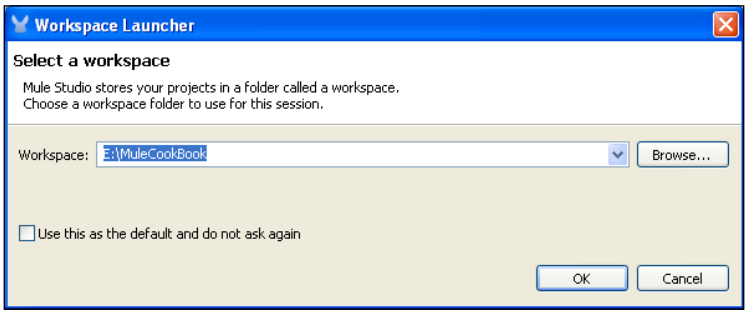


The arrow indicates the one-way type of exchange pattern.

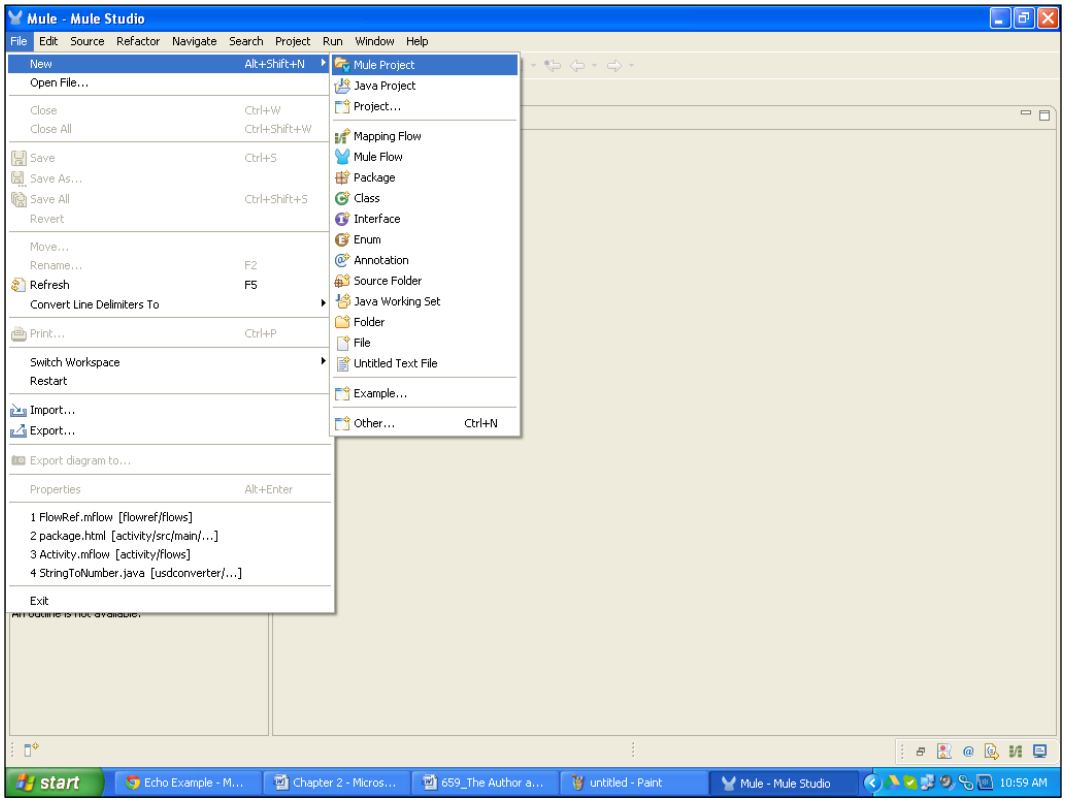
Getting ready

We have to use two components: the HTTP Endpoint and the Java component.

1. Open Mule Studio and enter the workspace name as shown in the following screenshot:



2. For creating a new project, go to **File** | **New** | **Mule Project**. Enter the project name called `Echo` and click on **Next** and then on **Finish**. Your new project is created now, so you can start the implementation.

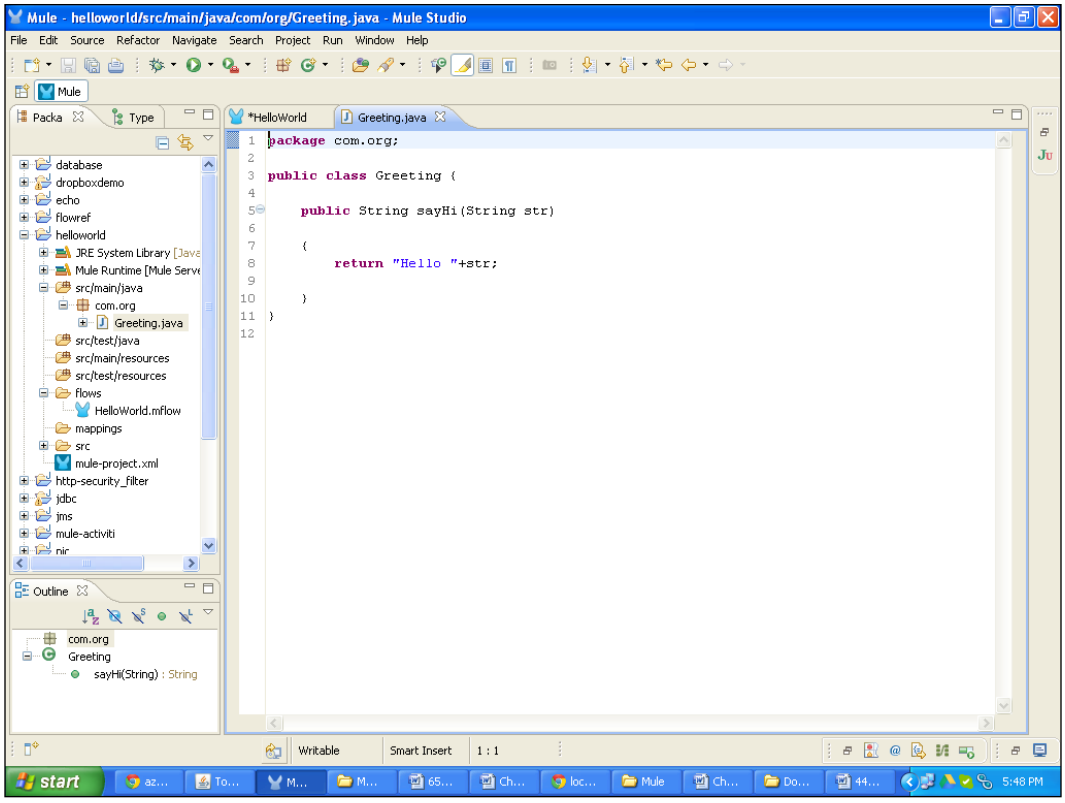


How to do it...

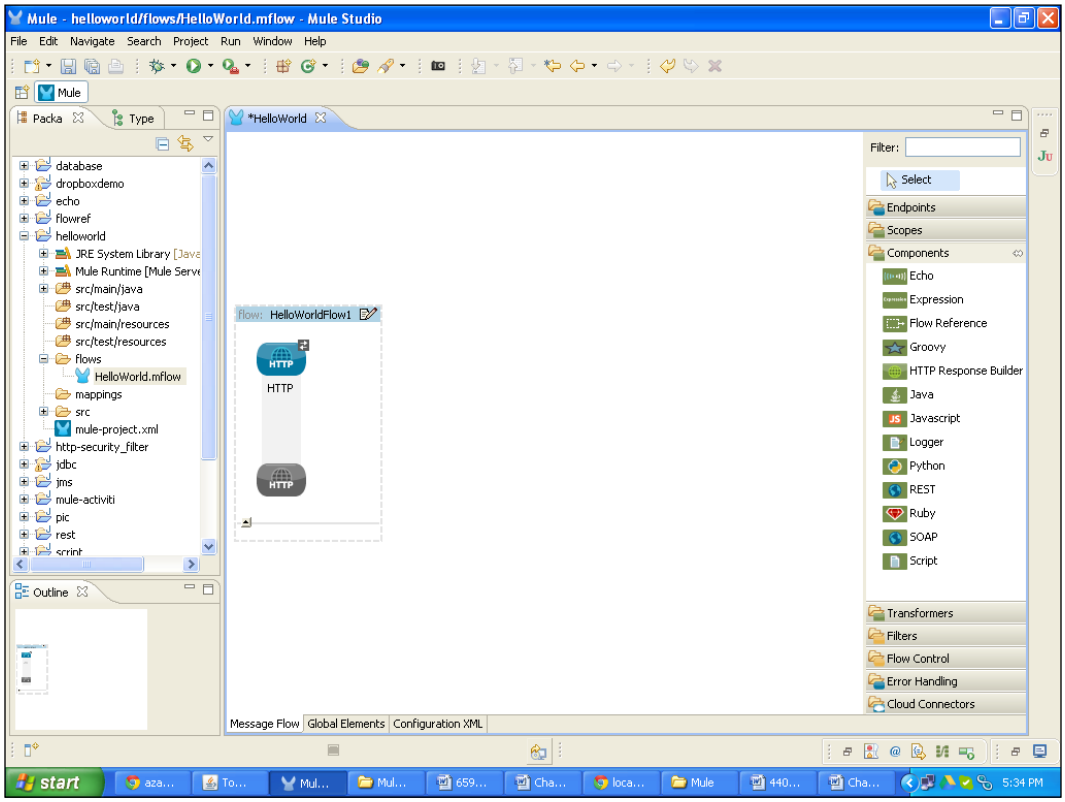
In this section you will see the Hello World example, which is created in Mule Studio. In this example two components are used: one is the HTTP Endpoint and the other is the Java component.

1. To create a class, go to `src/main/java`, right-click on it, and go to **New** | **Class**. Create a class called `Greeting` under the package `com.org`; here, we have created the `sayHi` method and its return type is set to `String`.

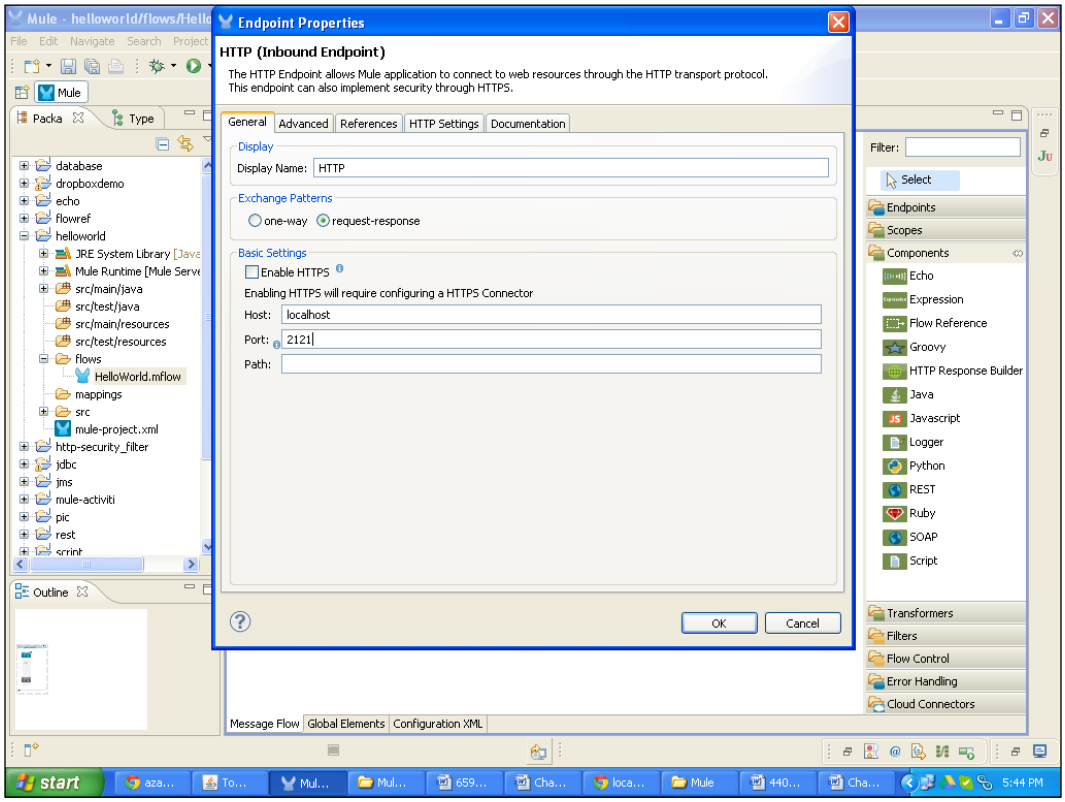
```
public String sayHi(String str)
  {
    return "Hello"+str;
  }
```



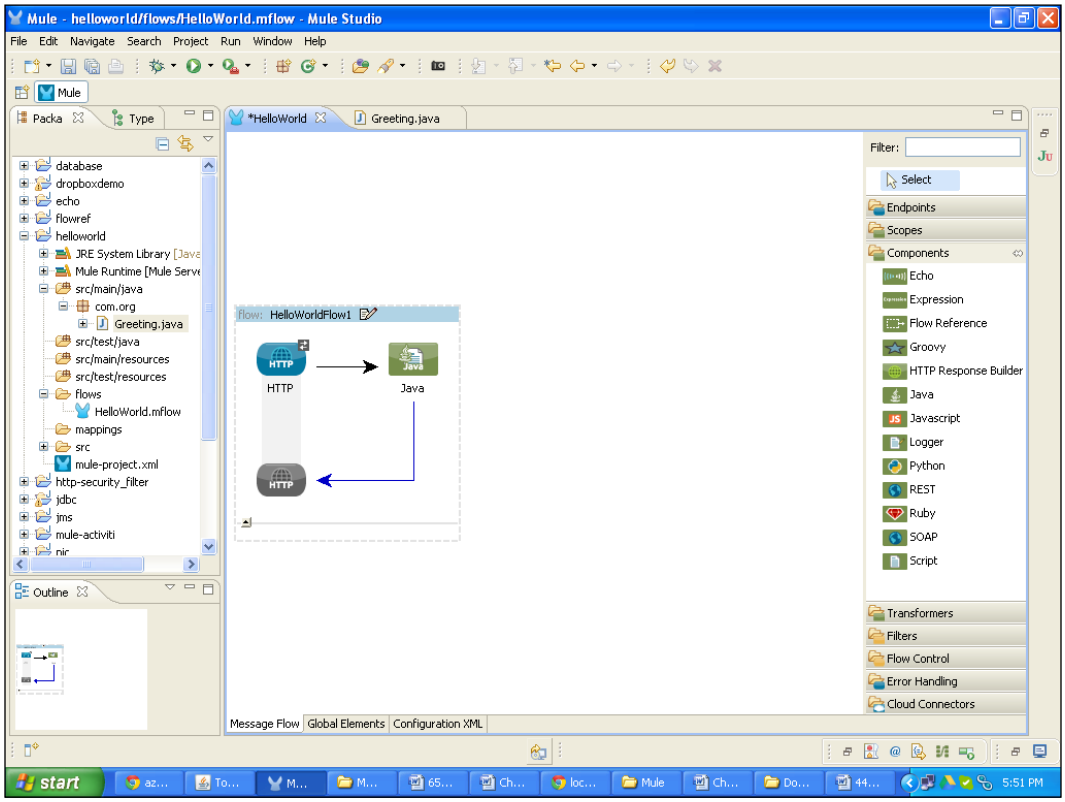
2. Go to the `HelloWorld.mflow` file. Firstly, you have to drag the **HTTP** Endpoint onto the canvas; to configure it, double-click on the **HTTP** Endpoint.



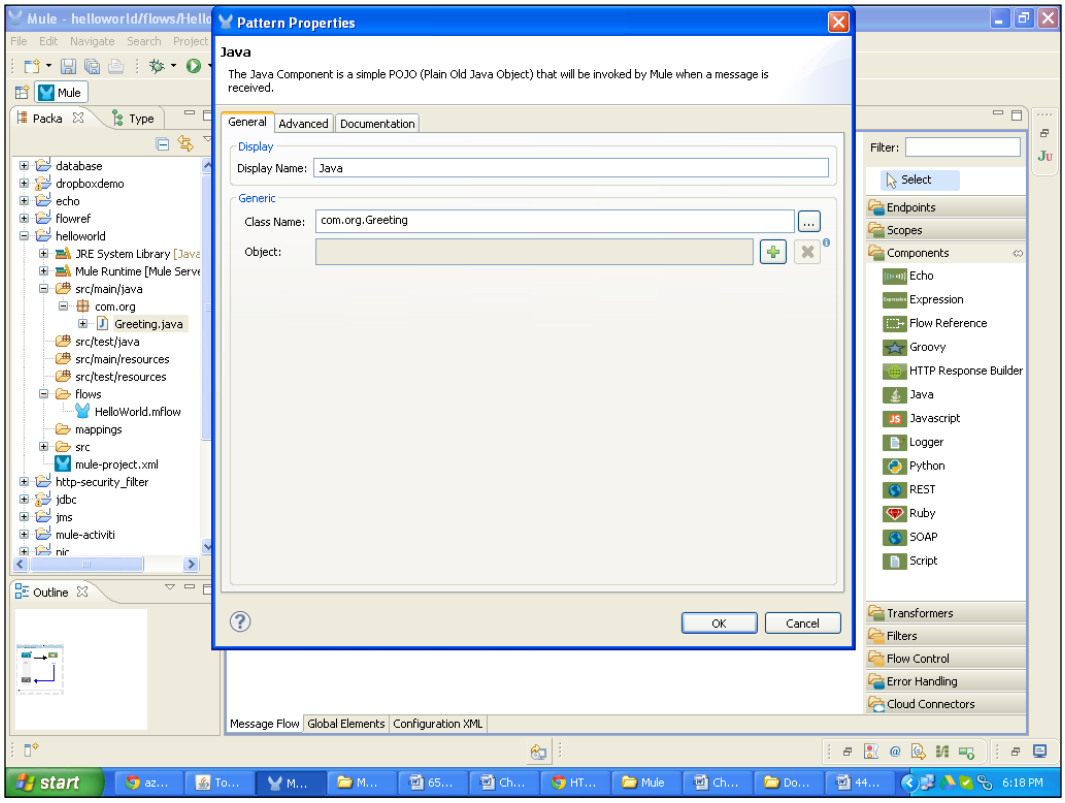
3. You will see a similar screen on your system. You have to enter the **Host** and **Port** field values. By default, the port number is **8081**. You can change the hostname and port number, but note that these two fields are mandatory.



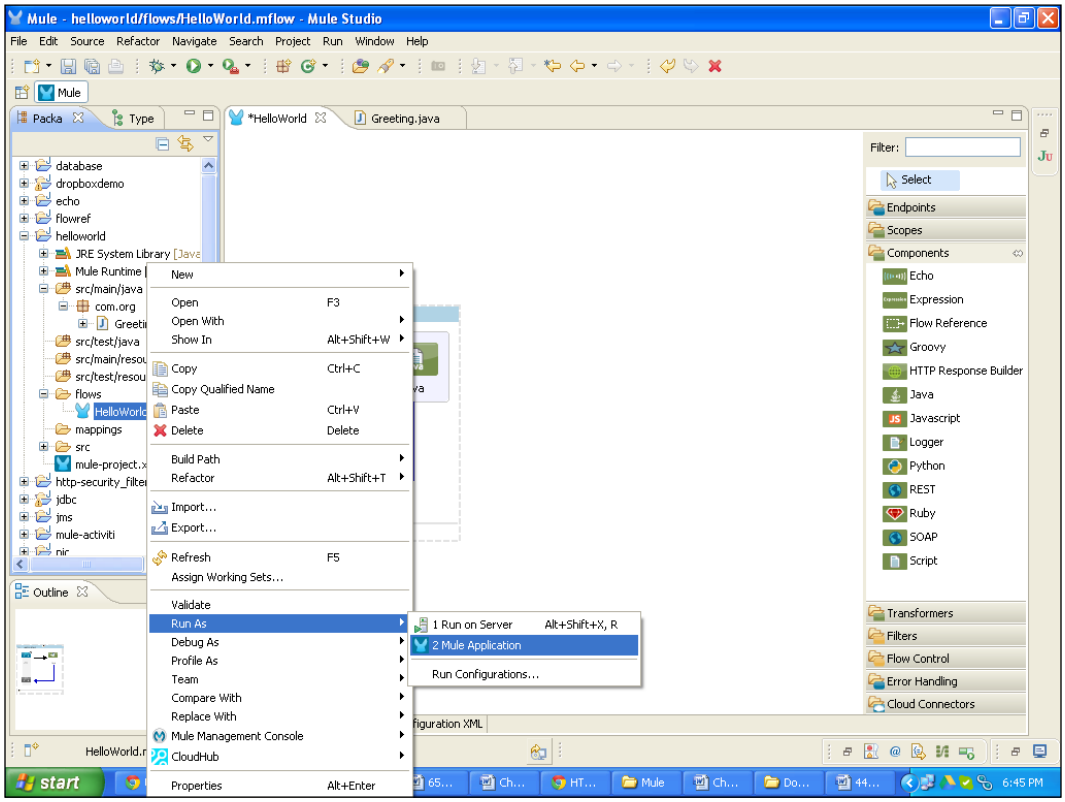
4. To import a class, drag the **Java** component and configure it. Here we have imported the `Greeting.java` class that was created before.



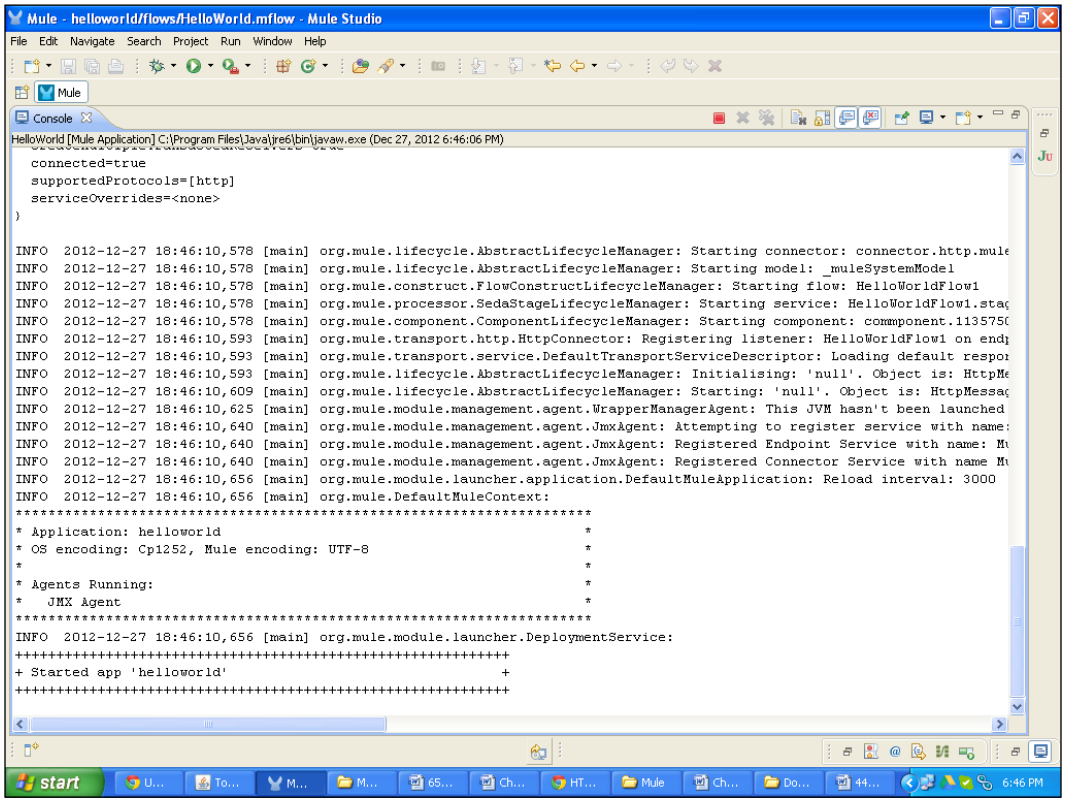
5. To configure the **Java** component, double-click and configure it as we have seen earlier. Just click on the Browse button (beside the **Class Name:** field), and a new window will open. Here, you can import the `Greeting.java` class that was created before and click on the **OK** button.



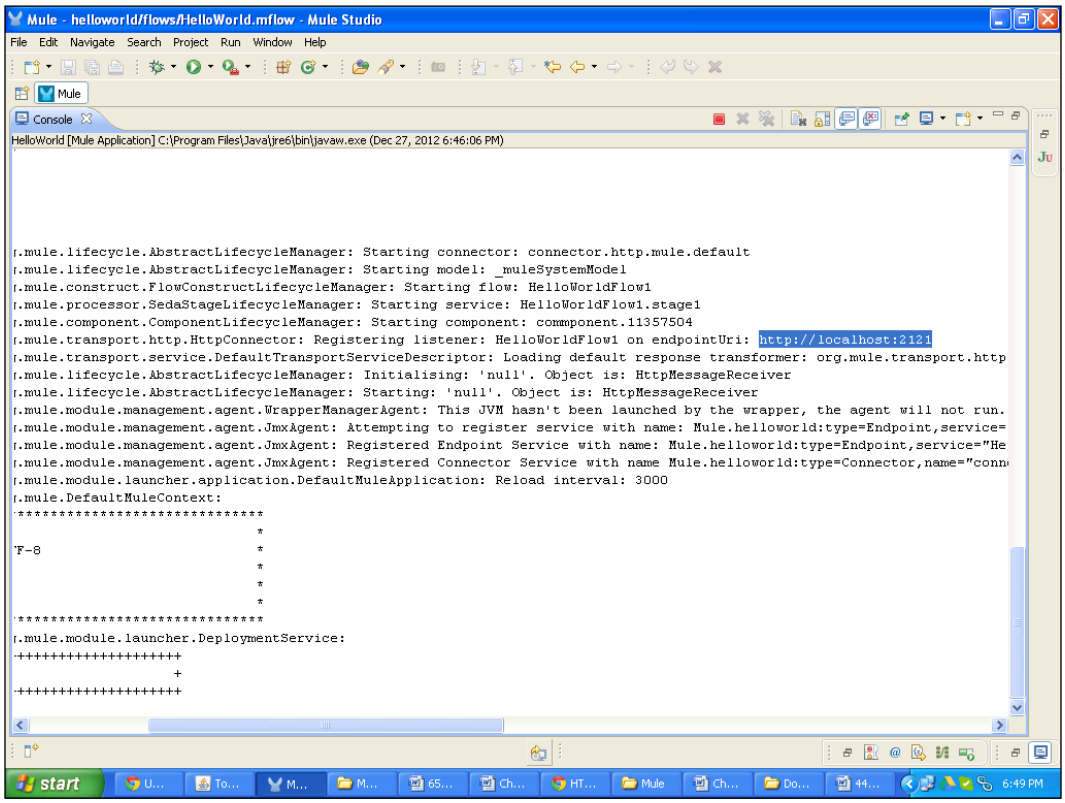
6. To deploy the application code in the Mule Server, go to **Run As** | **Mule Application**, and the Mule Server will deploy your application.



7. If your application code is successfully deployed, you will see the message `Started app 'helloworld'` on the console.

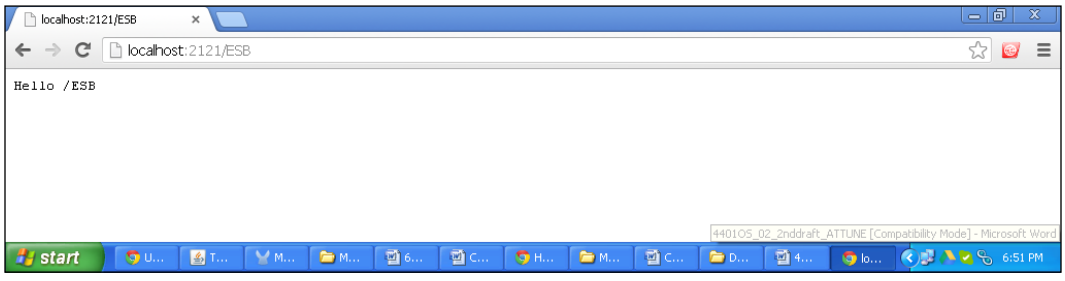


8. Copy `http://localhost:2121` and paste it in your browser to see the output.



How it works...

By putting the URL in the browser, you can see the following output. The word, `Hello`, is called from the `Greeting` class through the HTTP request-response exchange pattern. When a request is received, the Java component simply returns whatever was sent as part of the request.



Configuring the IMAP Endpoint to retrieve e-mails

IMAP stands for **Internet Message Access Protocol**. The IMAP/POP3 connector allows you to receive e-mail messages from a mail server using IMAP/POP3. The IMAP Endpoint is configured as Inbound with a one-way exchange pattern. POP3 is similar to IMAP by functionality. **POP3** stands for **Post Office Protocol Version 3**. The POP3 Endpoint can be configured as a one-way exchange pattern.

Getting ready

In this section, you will learn how to configure the namespace and schema location in the Mule configuration XML file. When you are dealing with XML documents in Mule, you need to declare any namespaces used by the document. You can specify a namespace globally so that it can be used by XPath expressions across Mule. You can declare a namespace locally also in filters and routers using the `<namespace>` element.

The namespace for the IMAP XML namespace is:

```
xmlns:imap "http://www.mulesoft.org/schema/mule/imap"
```

The syntax for the IMAP XML namespace is:

```
xmlns:imaps "http://www.mulesoft.org/schema/mule/imaps"
```

The IMAP transport can be used for receiving messages from IMAP inboxes using the `javax.mail` API. When we use IMAP, we will have to use the following schema location:

```
http://www.mulesoft.org/schema/mule/imap/3.1/mule-imap.xsd
```

The IMAPS transport uses secure connections over SSL/TLS. When we use IMAPS, we will have to use the following schema location:

```
http://www.mulesoft.org/schema/mule/imaps/3.1/mule-imaps.xsd
```

The namespace for the POP3 XML namespace is:

```
xmlns:pop3 "http://www.mulesoft.org/schema/mule/pop3"
```

The syntax for the POP3 XML namespace is:

```
xmlns:pop3s http://www.mulesoft.org/schema/mule/pop3s
```

The POP3 transport can be used for receiving messages from POP3 inboxes using the following XML schema location:

```
http://www.mulesoft.org/schema/mule/pop3/3.1/mule-pop3.xsd
```

The POP3S transport connects to POP3 mailboxes using the `javax.mail` API using the following XML schema location:

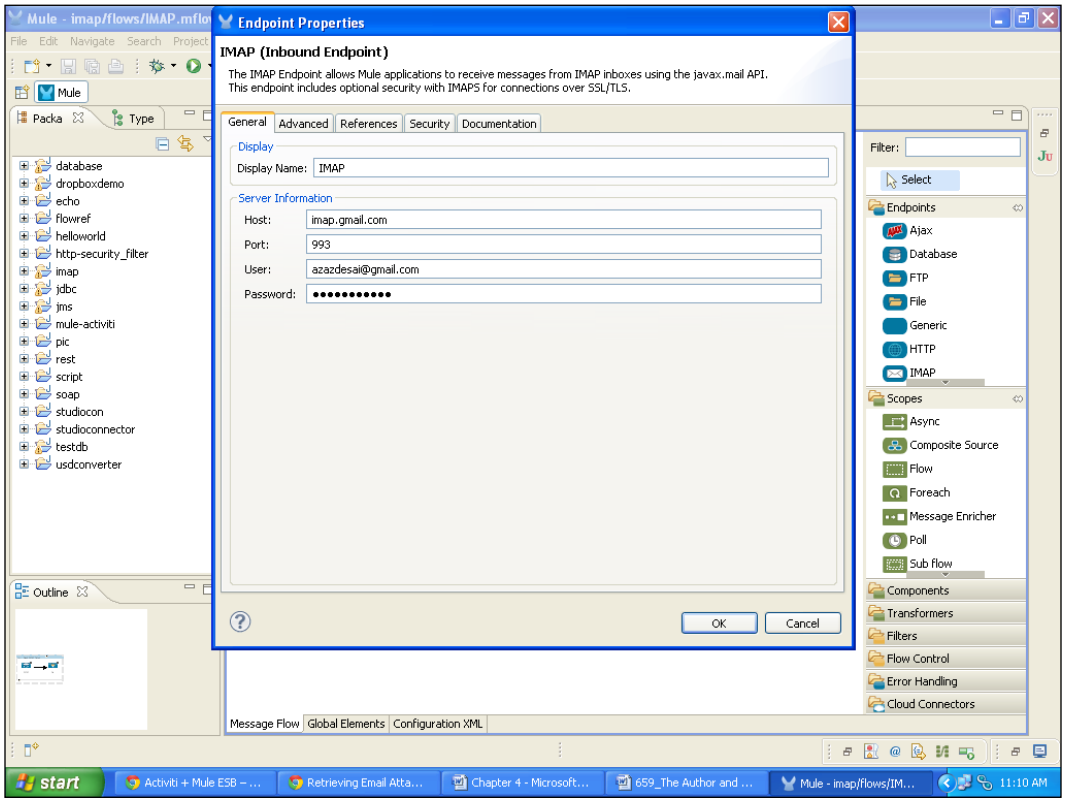
```
http://www.mulesoft.org/schema/mule/pop3s/3.1/mule-pop3s.xsd
```

How to do it...

Drag the **IMAP** Endpoint from the palette and drop it on the canvas. Double-click on it and fill the **User:**, **Host:**, **Port:**, and **Password:** fields, which are mandatory. For IMAP, the default port number is **143**. For IMAPS, the default port number is **993**. If you are using IMAP, your hostname will be `imap.gmail.com`, and if you are going with POP, your hostname will be `pop.gmail.com`. The default port number is **110** for POP3S, and for POP3 the default port number is **995**.

```
<imap:inbound-endpoint user="azazdesai@gmail.com" password="XXXXXX"
                       host="imap.gmail.com" doc:name="IMAP" port="993"/>
```

In the following screenshot, you will see how to configure the IMAP Endpoint:



How it works...

Internet Message Access Protocol (IMAP) is used for retrieving e-mails on a mail server from multiple computers and devices. For this, you must have configured the **IMAP** Endpoint.

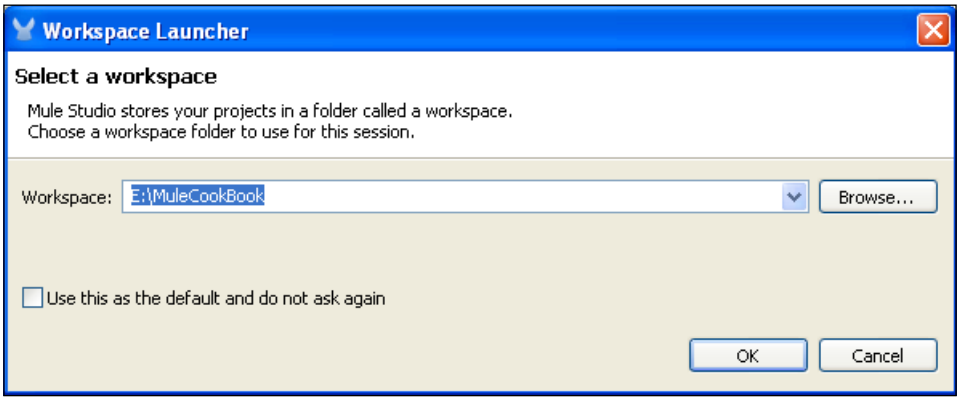
Using the JDBC Endpoint to connect to the database

The JDBC Endpoint is used to communicate with the database. It's used for retrieving, updating, deleting, and inserting database records. We will see an example on how it works. In this example, we will retrieve data from the database and store it in a file on our local hard drive.

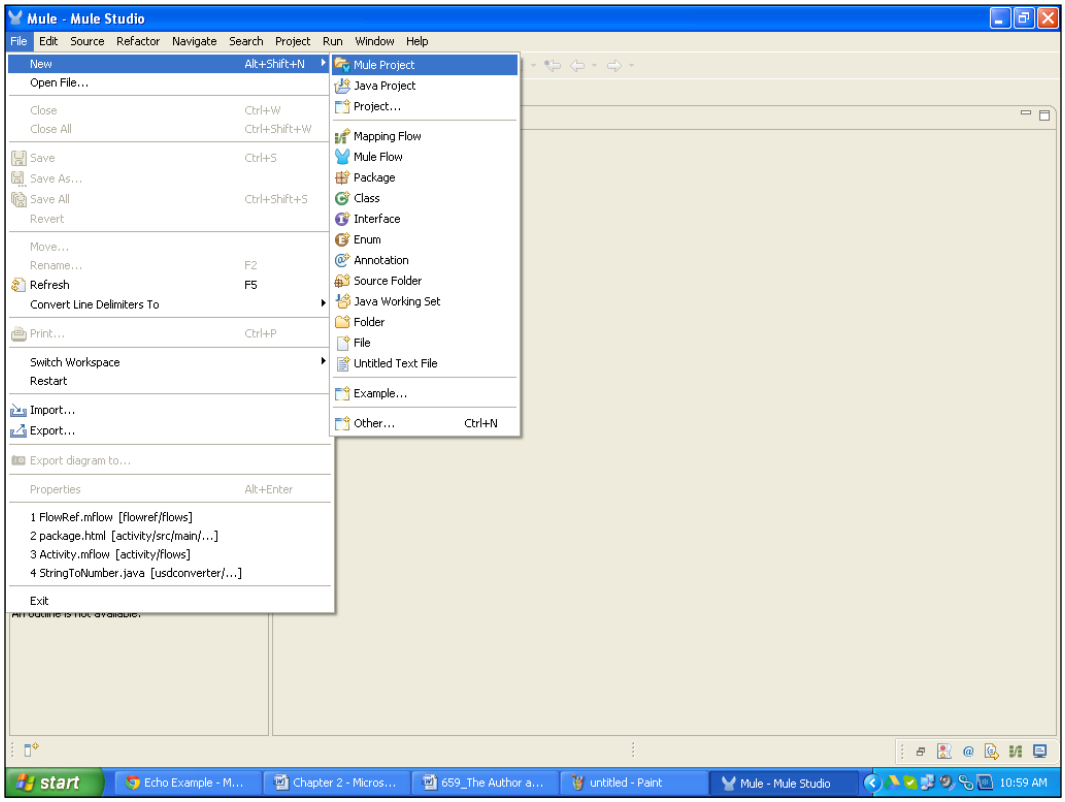
Getting ready

In this section, we will see how to use and configure the JDBC component in Mule Studio. In this example you will use three components: the JDBC Endpoint, the Object-to-String transformer, and the File Outbound Endpoint.

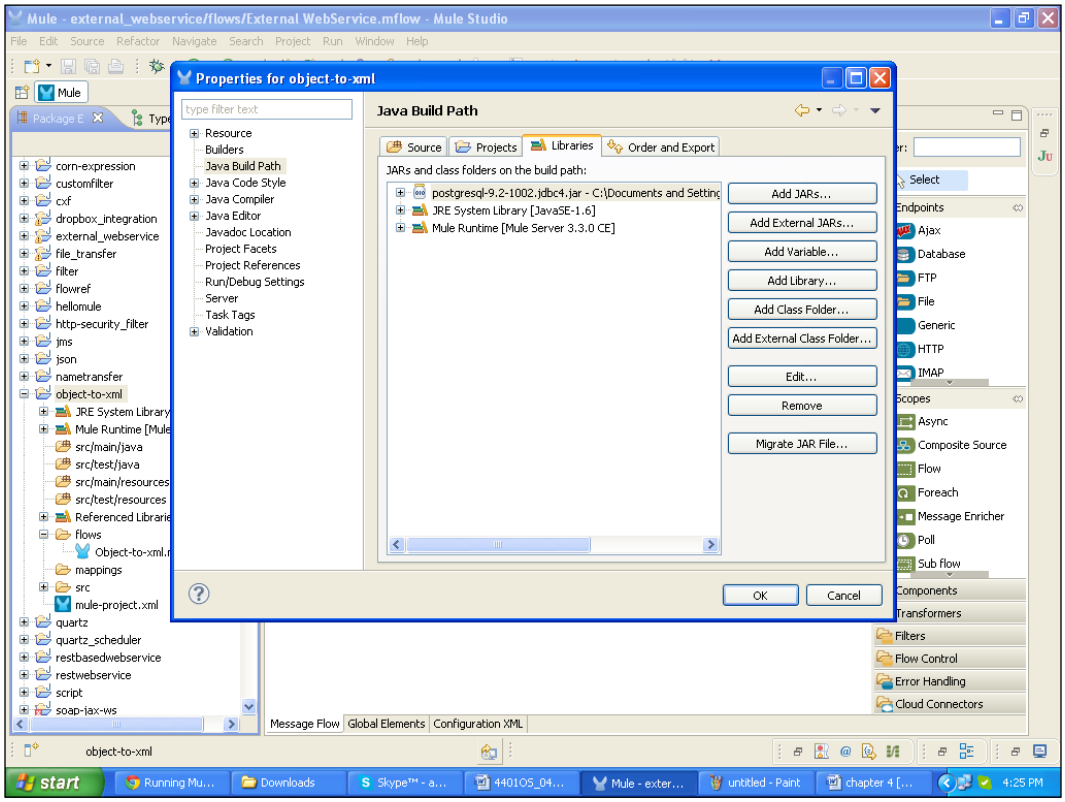
1. We will use one JDBC Endpoint, a data source, and the File Outbound Endpoint. Retrieve the record from the MySQL database and store it on the local hard drive.



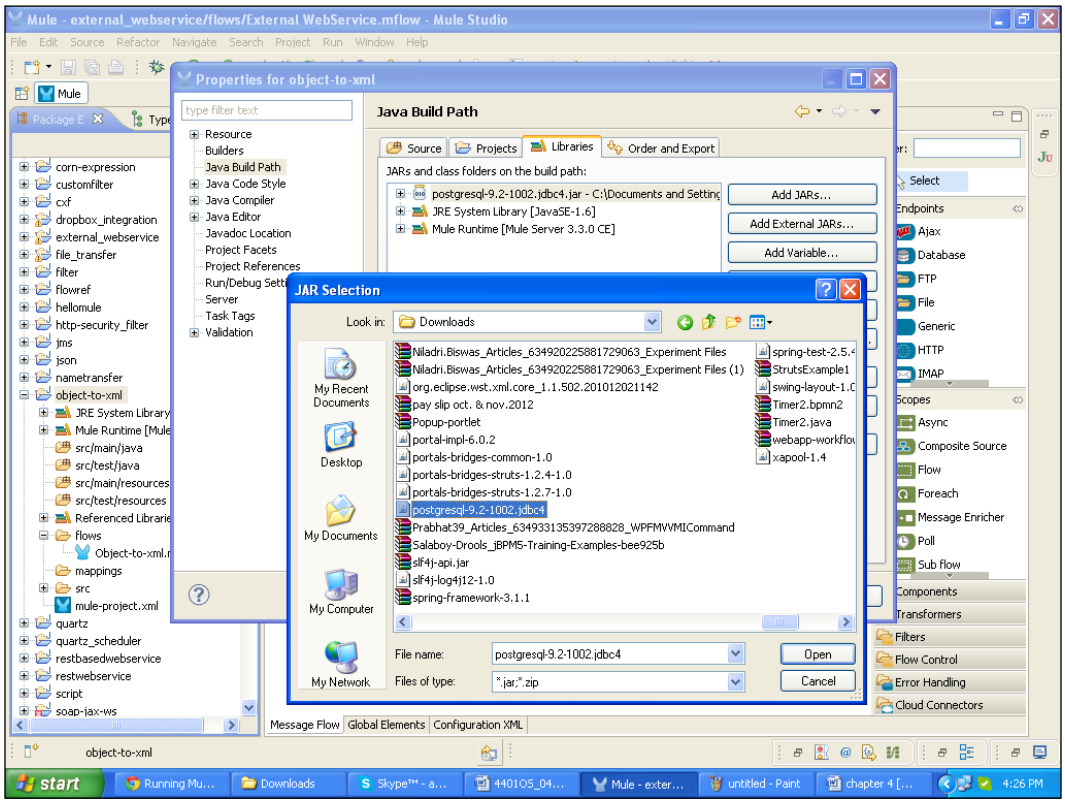
2. To create a new project, go to **File** | **New** | **Mule Project**. Enter the project name called Database and click on **Next** and then on **Finish**. Your new project is created. Now you can start the implementation.



3. Download the `postgresql-9.2-1002.jdbc4.jar` file from the source code on the Packt Publishing website. First, you have to add the `postgresql` JAR file in your class path. Right-click on your project, select **Properties**, go to the Java build path, and click on the **Add External JARs..** button.



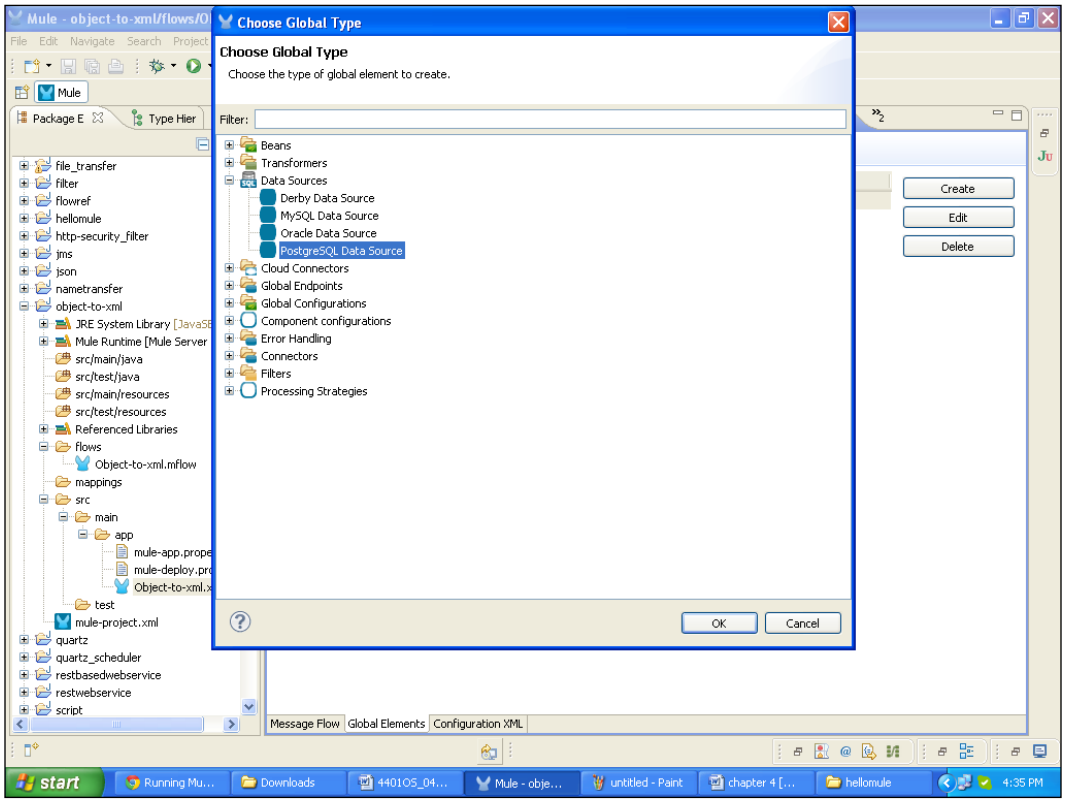
4. Select the JAR file, `postgresql-9.2-1002.jdbc4`, and click on the **OK** button.



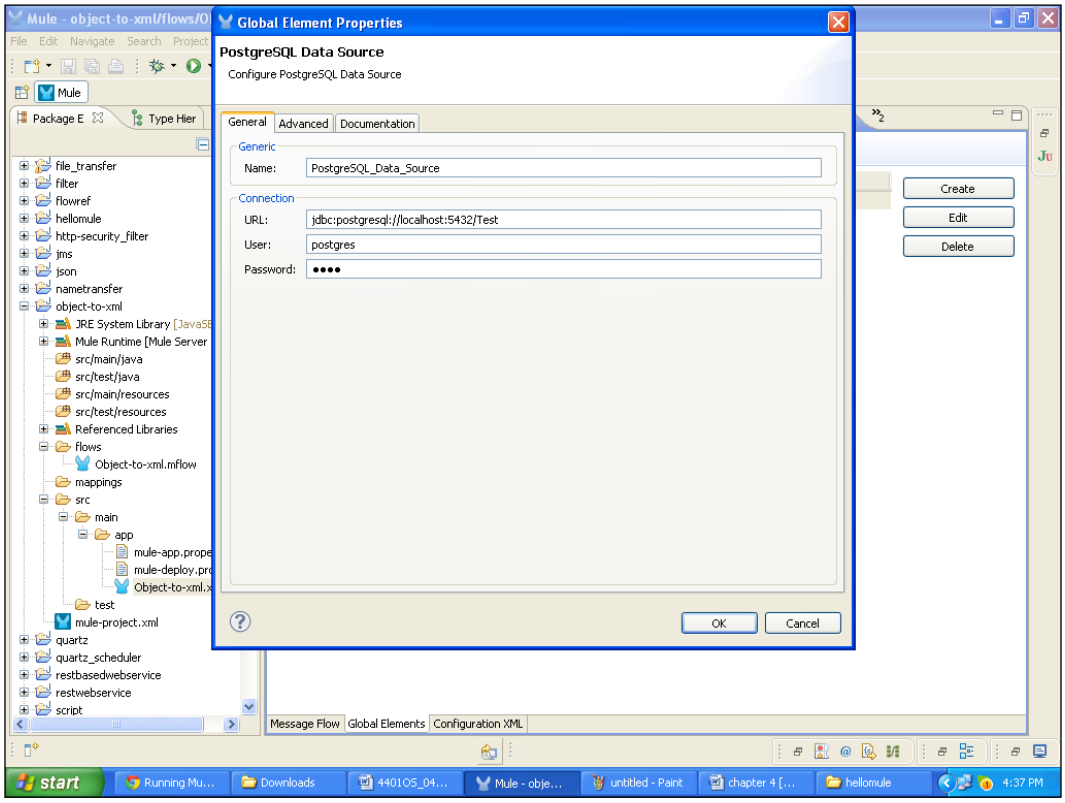
How to do it...

In this section we will configure the PostgreSQL database in Mule Studio and learn how to use this JDBC Endpoint in a flow.

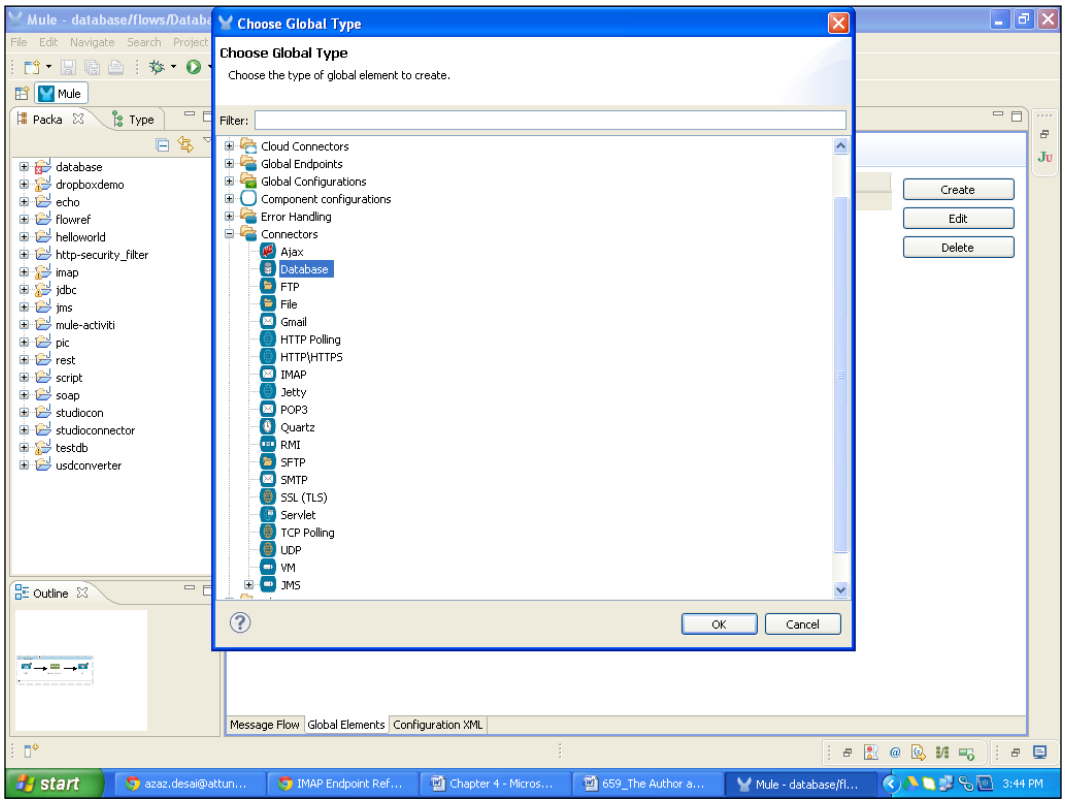
1. To create a flow, click on `Database.mflow` and go to **Global Elements** | **Create** | **Data Sources** | **PostgreSQL Data Sources**.



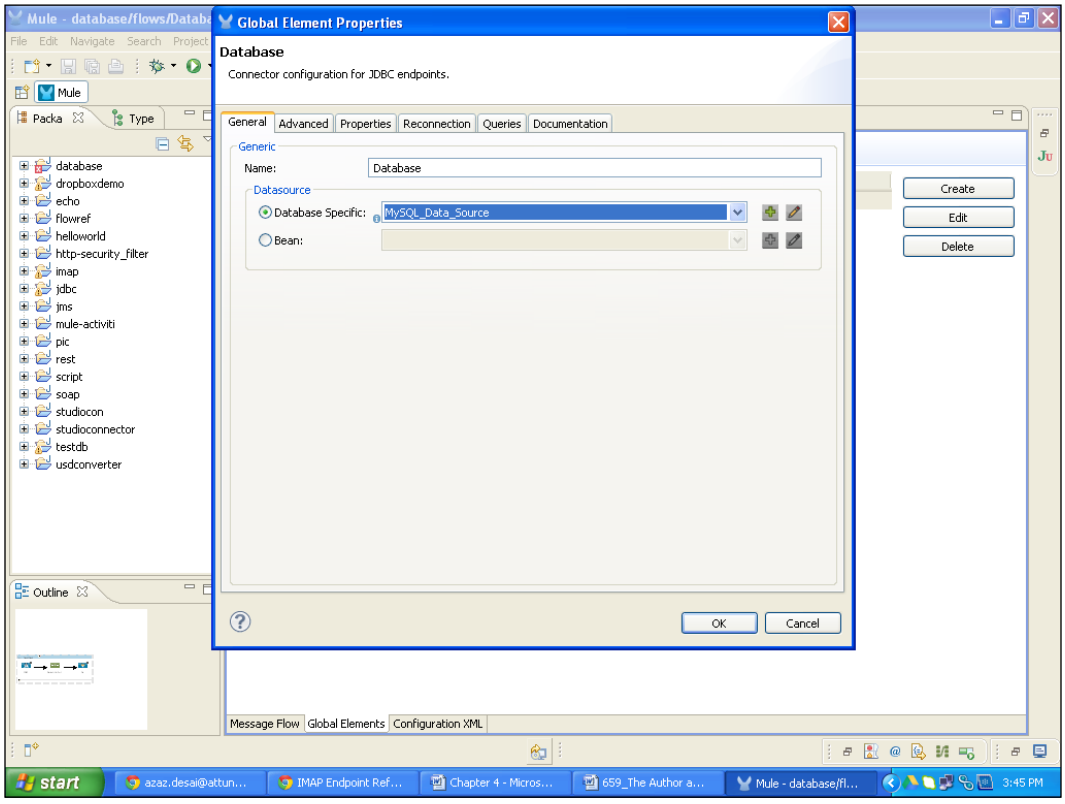
2. You have to configure **PostgreSQL Data Sources**. In the URL textbox, enter `jdbc:postgresql://localhost:5432/Test` as the value. `Test` is our database, which is created in PostgreSQL. In the end, enter the PostgreSQL user credentials.



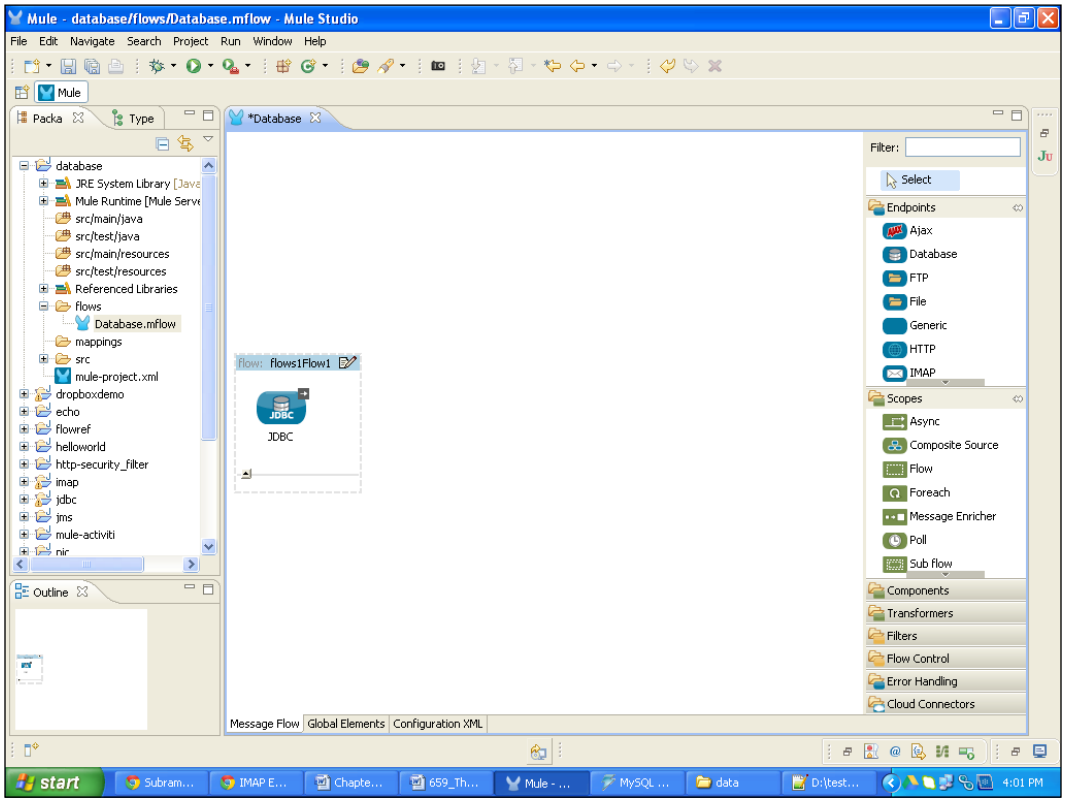
3. Go to **Connectors** | **Database**.



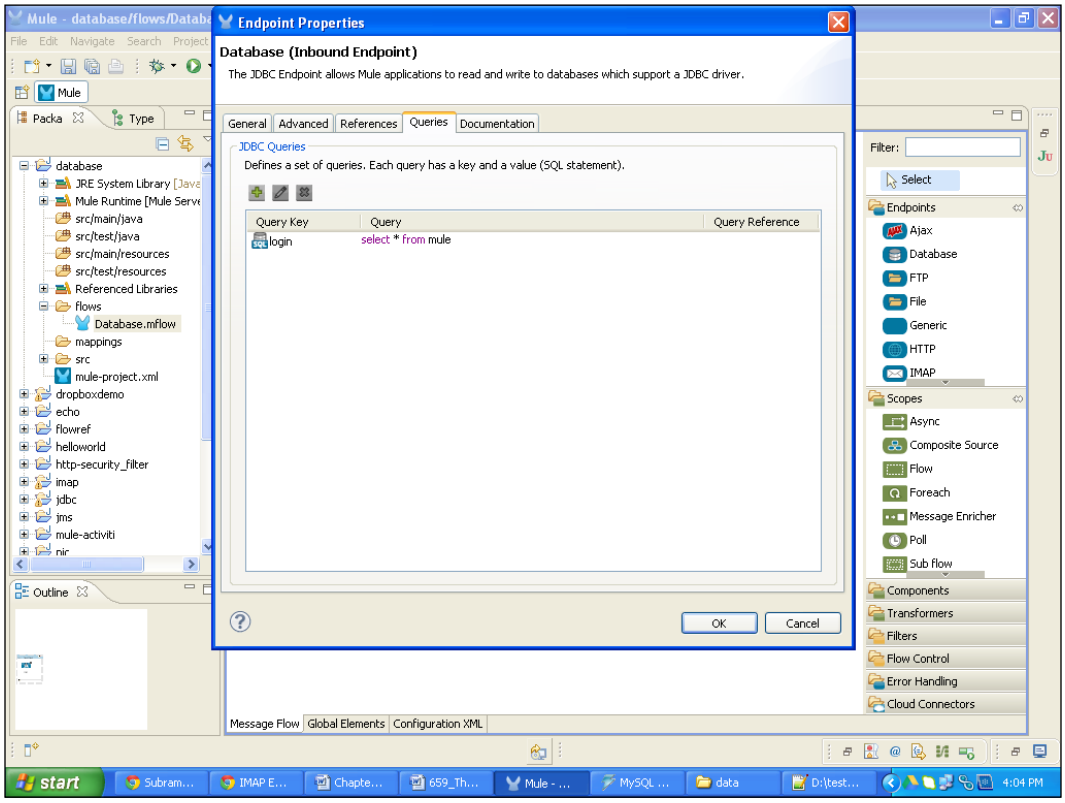
4. To configure the Database connector, select the **Database Specific:** name, **PostgreSQL_Data_Sources**, that was created before and click on **OK**.



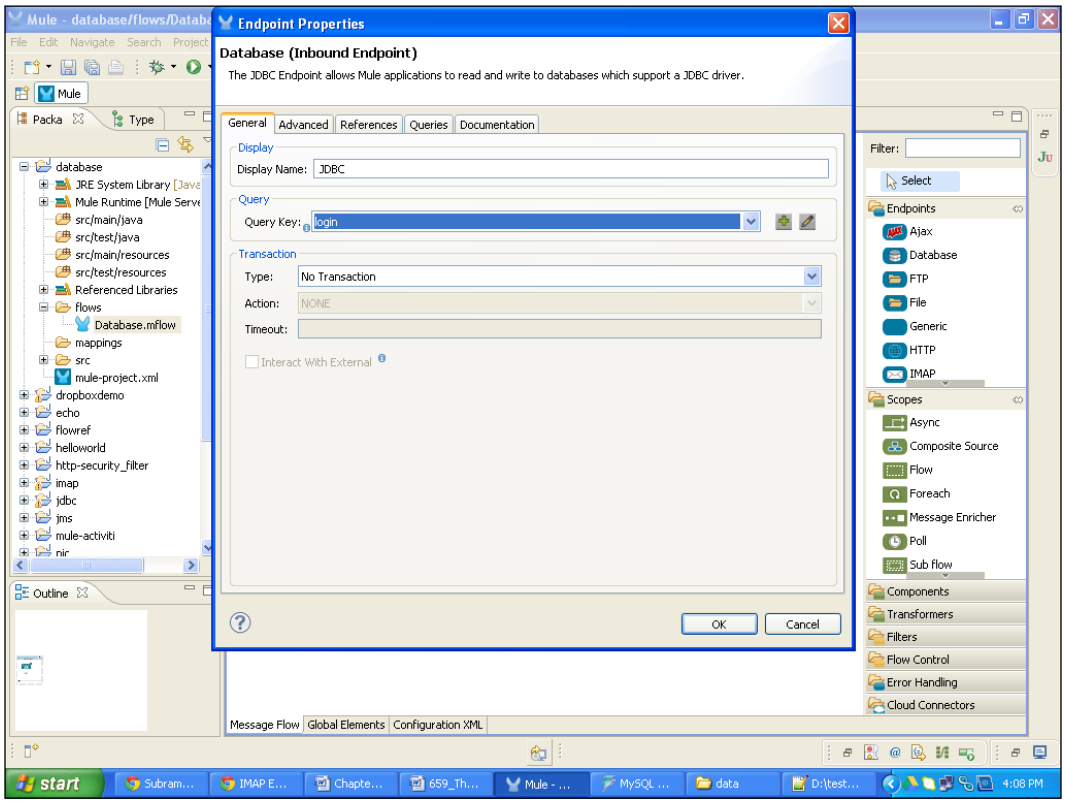
5. To create a flow, click on the **Message Flow** tab, drag the **JDBC** Endpoint from the palette, and drop it on the canvas.



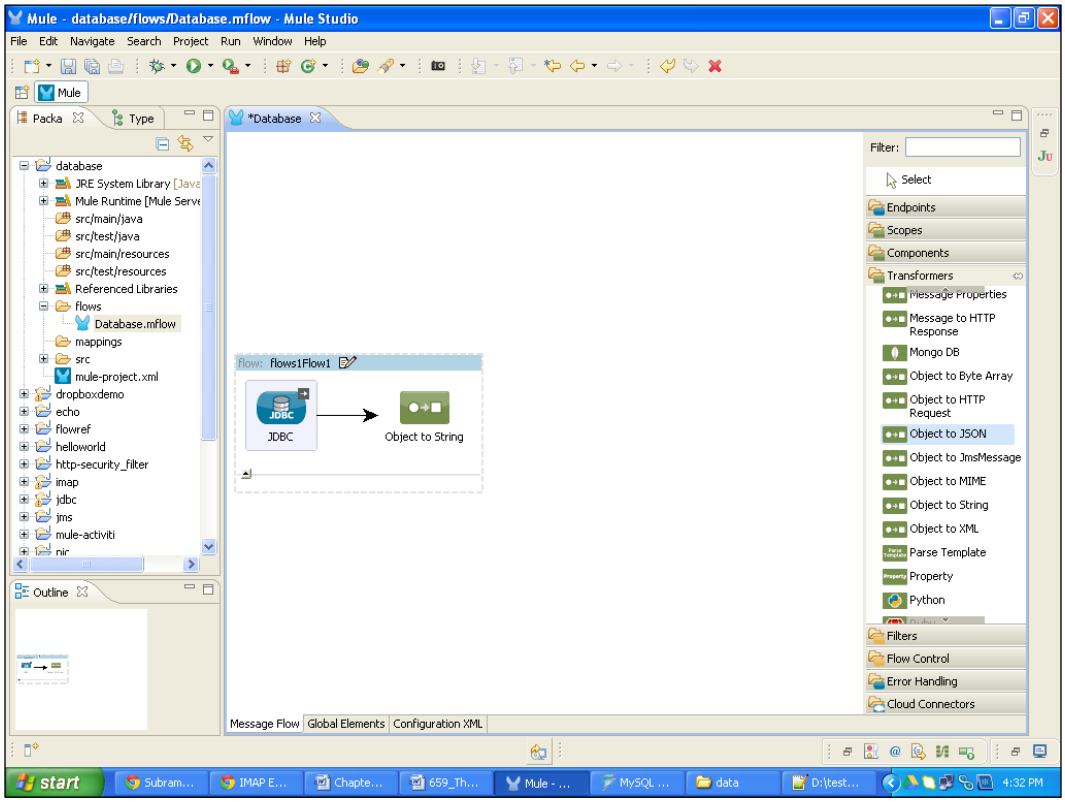
6. To configure the JDBC Endpoint, double-click on it. Click on the **Queries** tab and click on the plus icon, as shown in the following screenshot. Now enter the query key, `login`, and enter the query in the query box; for example, `select * from mule`. Here, `mule` is our table name, which is created in PostgreSQL.



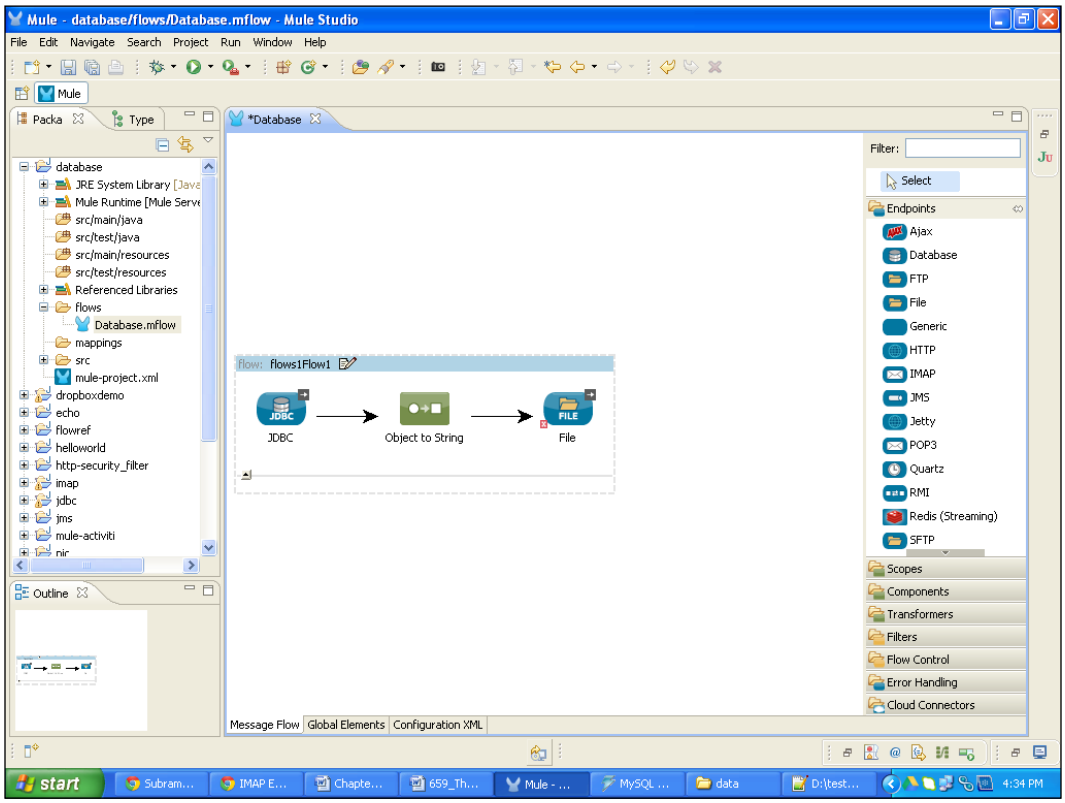
7. Click on the **General** tab, select the query key, `login`, and click on **OK**.



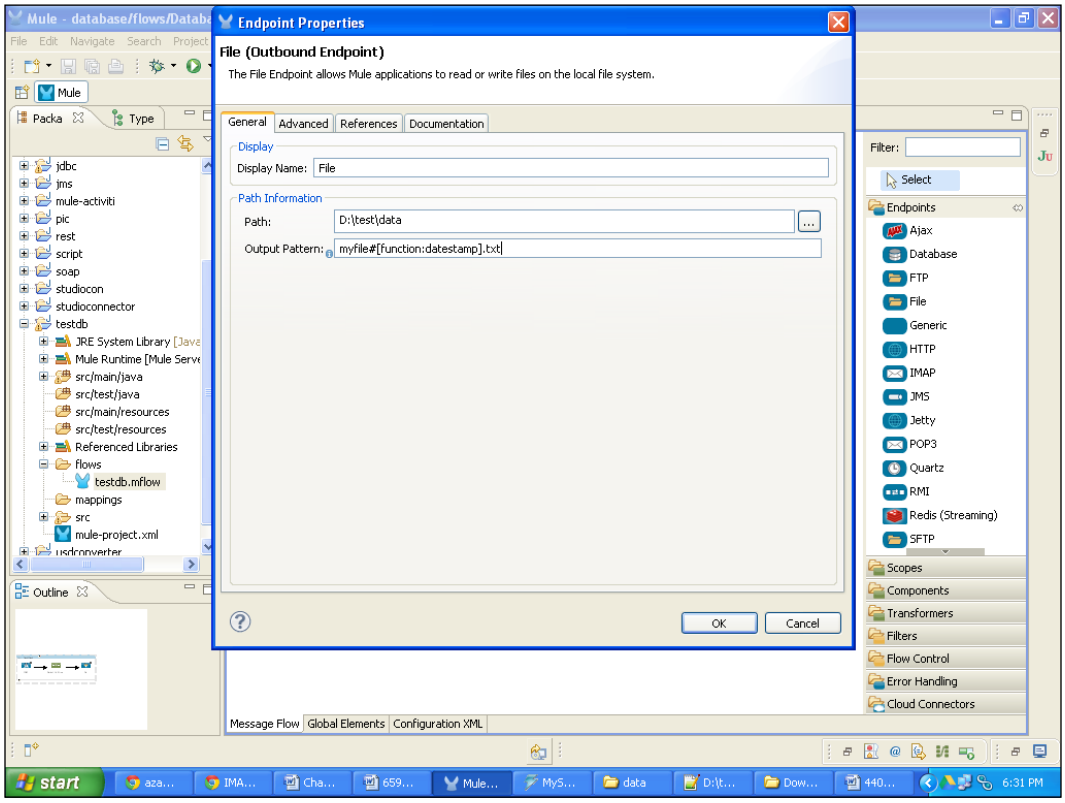
8. Drag the **Object to String** transformer from the palette and drop it on the canvas. No need to configure this transformer because it converts the data object to string.



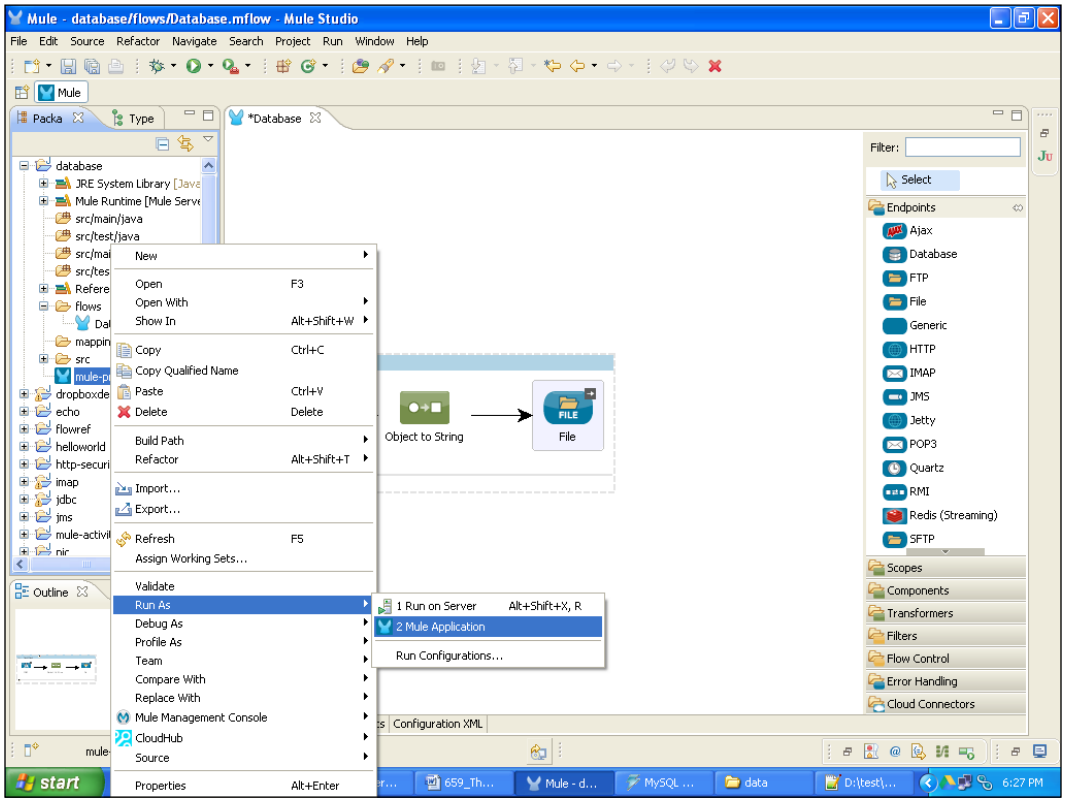
9. Drag the **File Outbound** Endpoint from the palette and drop it on the canvas.



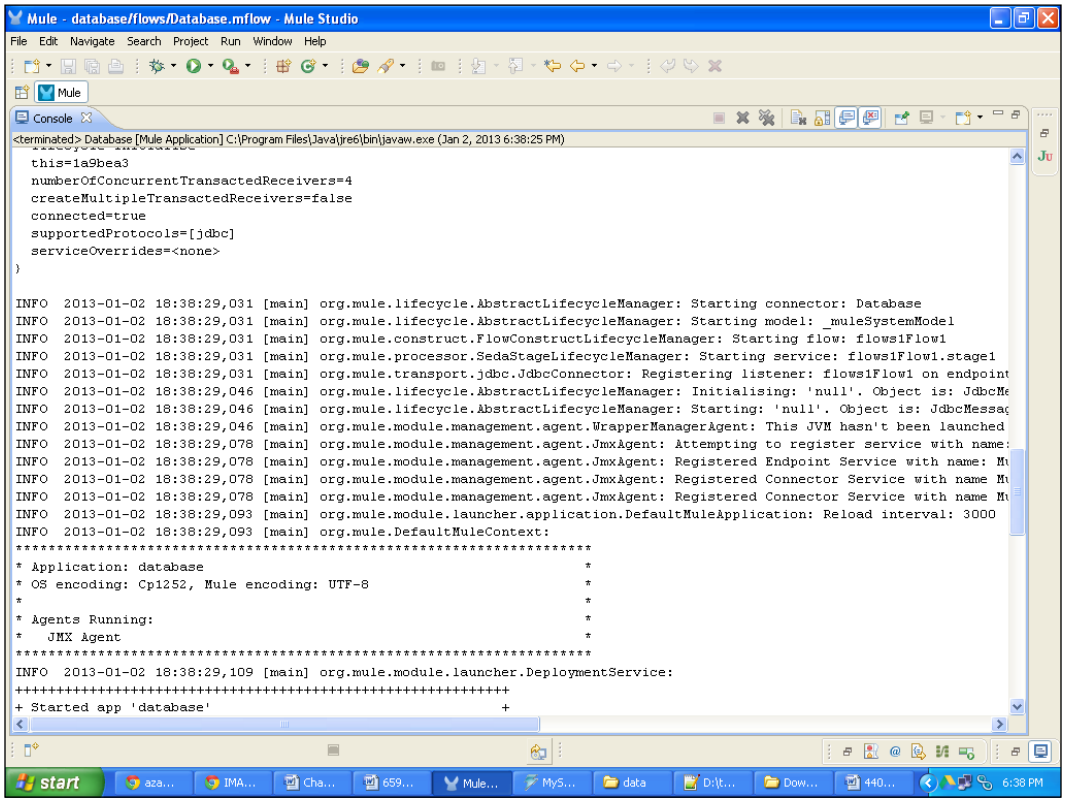
10. To configure the **File Outbound** Endpoint, double-click on it. Select a system path and output pattern, which is `myfile#[function:datestamp].txt`. Now, enter the filename and the expression. Through the expression you can display the current date and time, and here the file format is `.txt`. Now click on the **OK** button.



11. Now we are ready for the deployment. If you haven't saved your application code, first save it. After saving your project, right-click on the `Database.mflow` file and go to **Run As** | **Mule Application**.



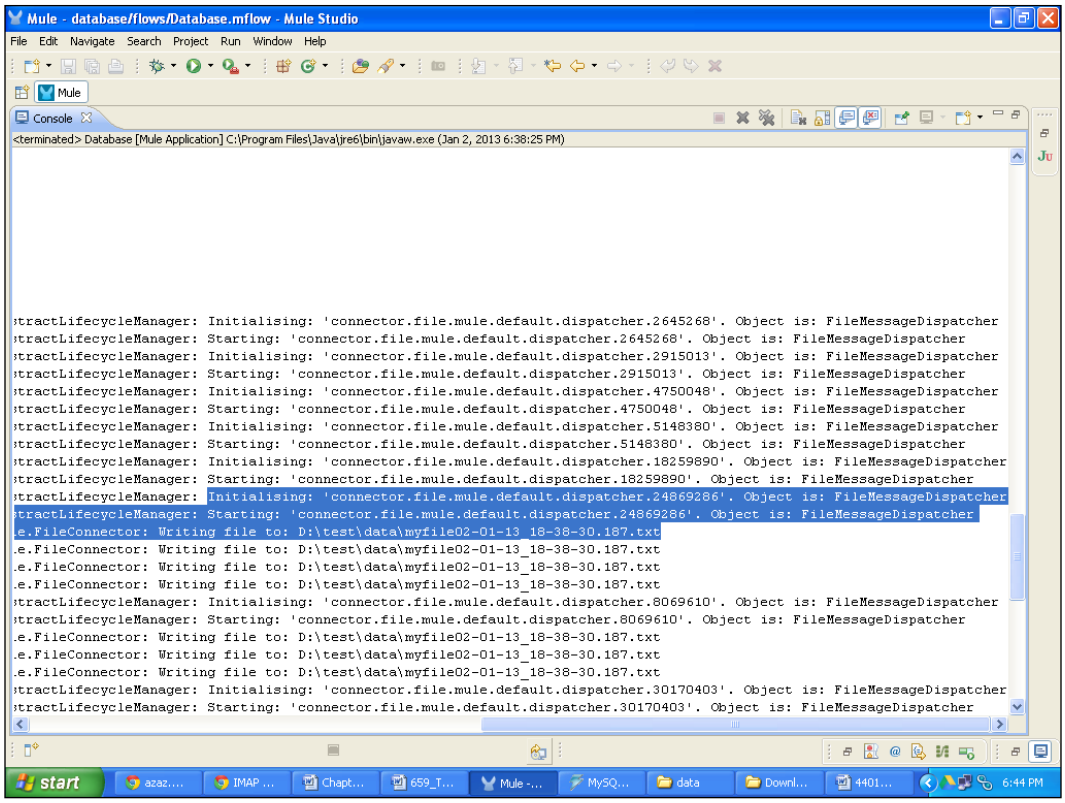
12. If your application code is successfully deployed, you will see the message `Started app 'database'` on the console.



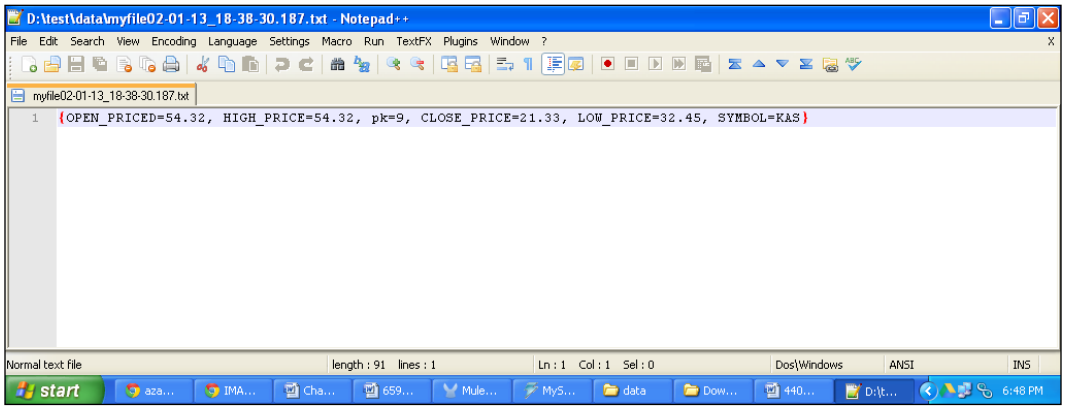
How it works...

Once your deployment is done successfully, you can see the log on the console. It will show that the files are stored in the particular drive. When you see the log on the console, you will see that the files are transferred to the destination path.

1. You can see that the database records are stored in a particular file. Here, the files are stored into the `D:/`.



2. The file in Notepad++ looks like the following screenshot. This data is retrieved from the database in the `mule` table, which was created in the PostgreSQL database.



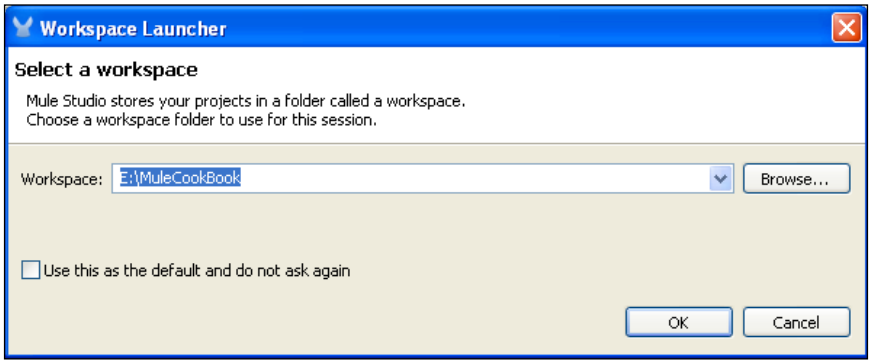
Implementing the File Transport channel using the File Endpoint

The File Endpoint is a transport channel. We can transfer files from one directory to another through the File Endpoint. The File Inbound Endpoint is used for setting the source path, and the File Outbound Endpoint is used to set the destination path. You can define the File Endpoint globally as well. If you declare the File Endpoint globally, you can set the reference tab to assign the name of the File Endpoint.

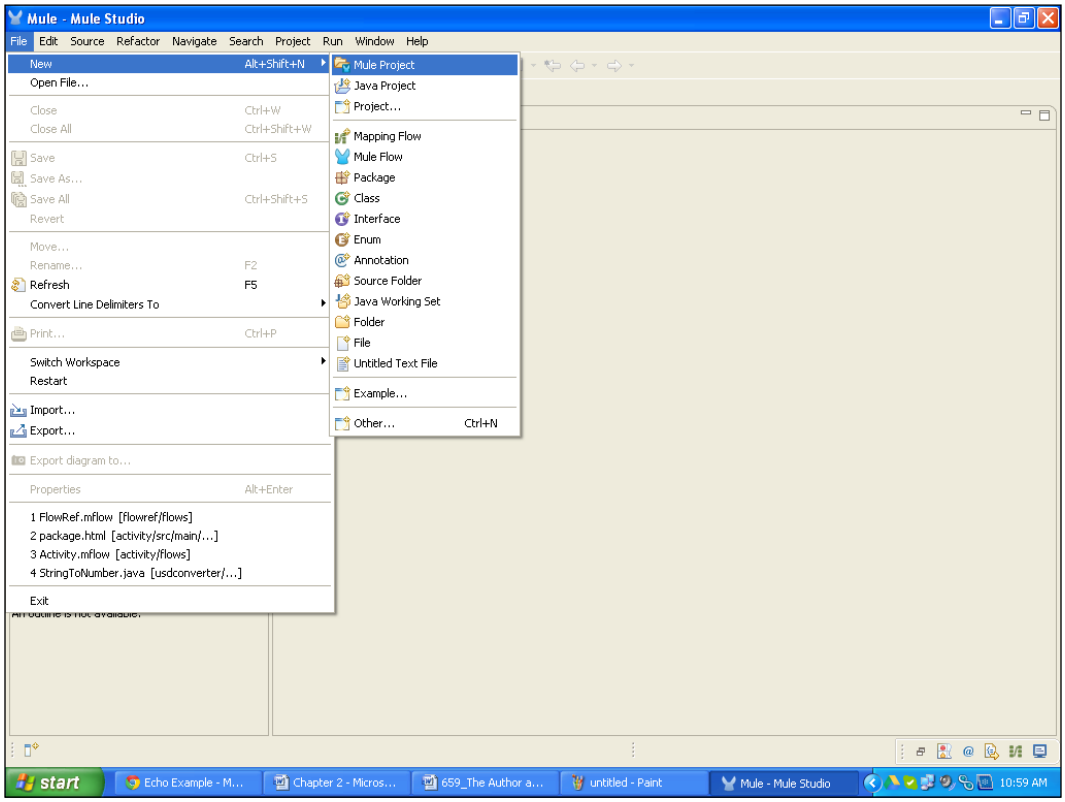
Getting ready

In this section, you will see how to use the File Endpoint and how to transfer files from one location to another using it.

1. Use the File Endpoint, a Choice Router, and the Echo component for transferring files from one location to another. Open Mule Studio and enter the workspace name as shown in the following screenshot:



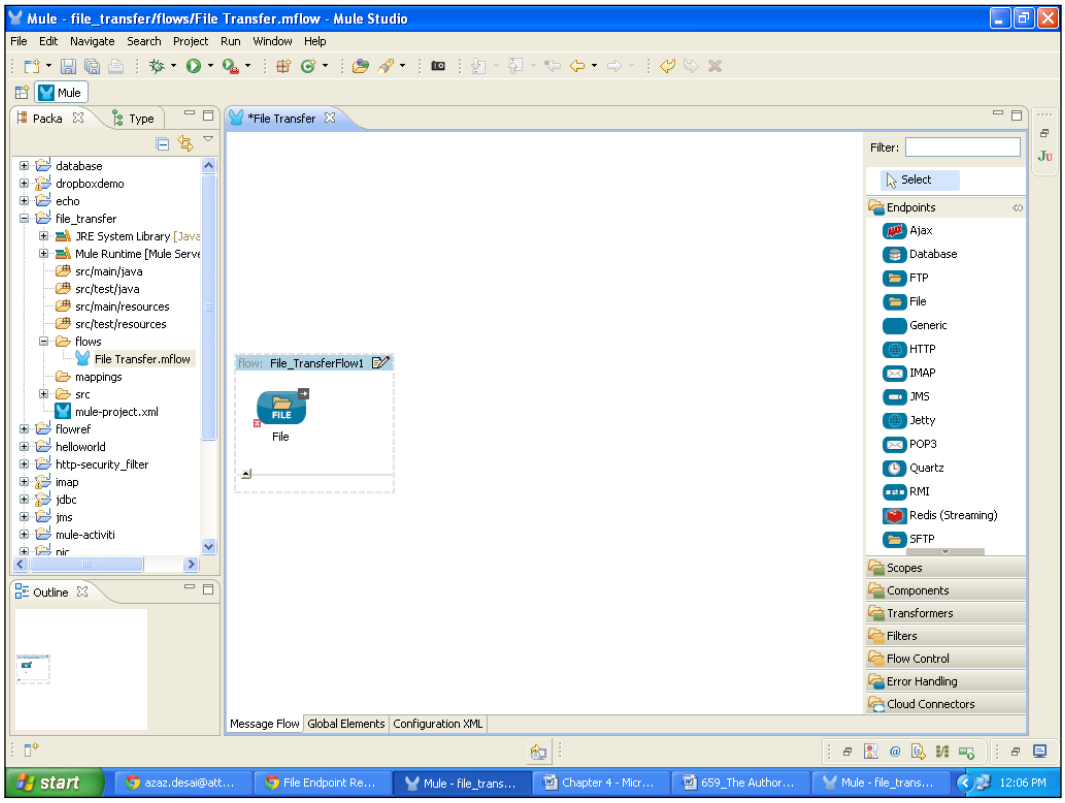
2. To create a new project, go to **File** | **New** | **Mule Project**. Enter the project name, `File Transfer`, and click on **Next** and then on **Finish**. Your new project is created. Now you can start the implementation.



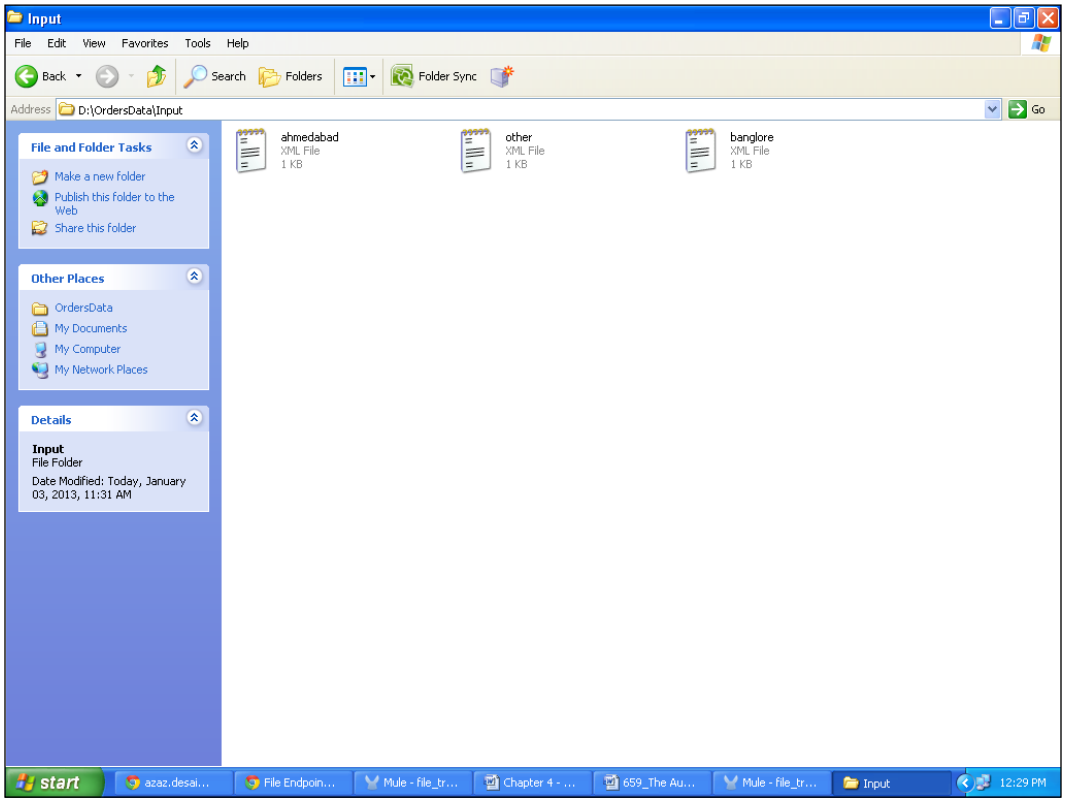
How to do it...

In this section you will see how to create a flow in Mule Studio and how it works using the File Endpoint. In this example, you will use four components: the File Inbound Endpoint, the Choice Router, the Echo component, and the File Outbound Endpoint.

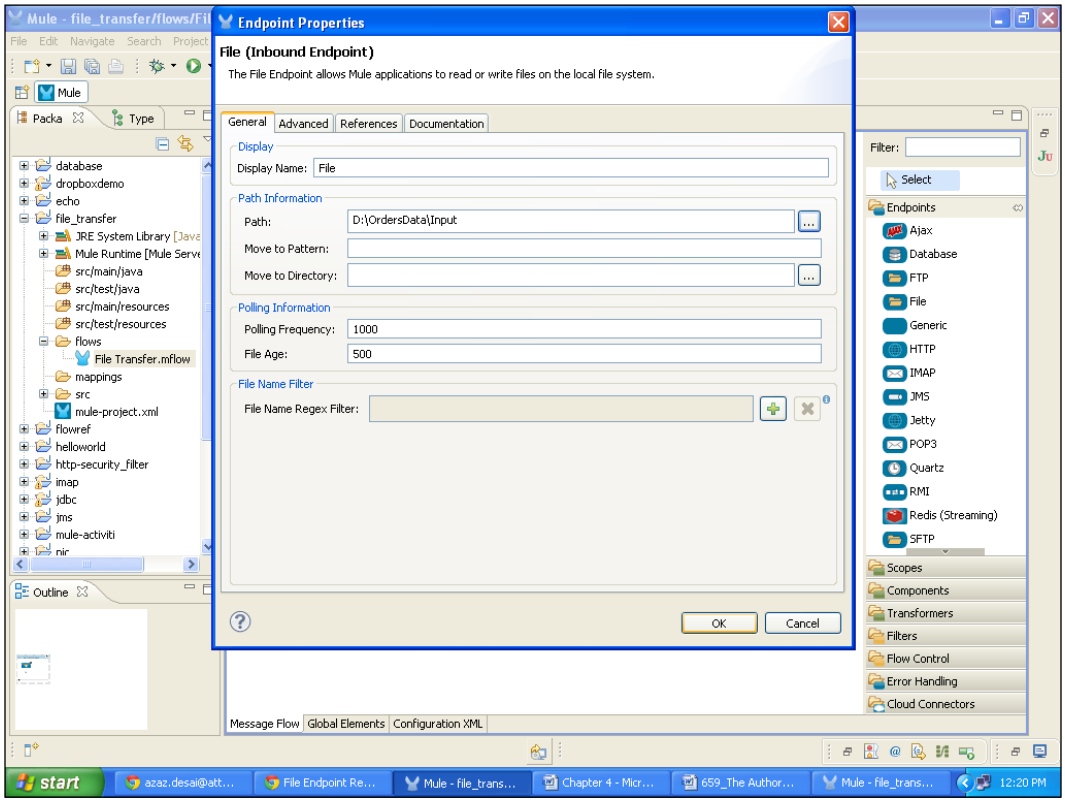
1. To create a flow, go to the file `Transfer.mflow` file. Firstly, you have to drag the **File** Endpoint from the palette and drop it on the canvas.



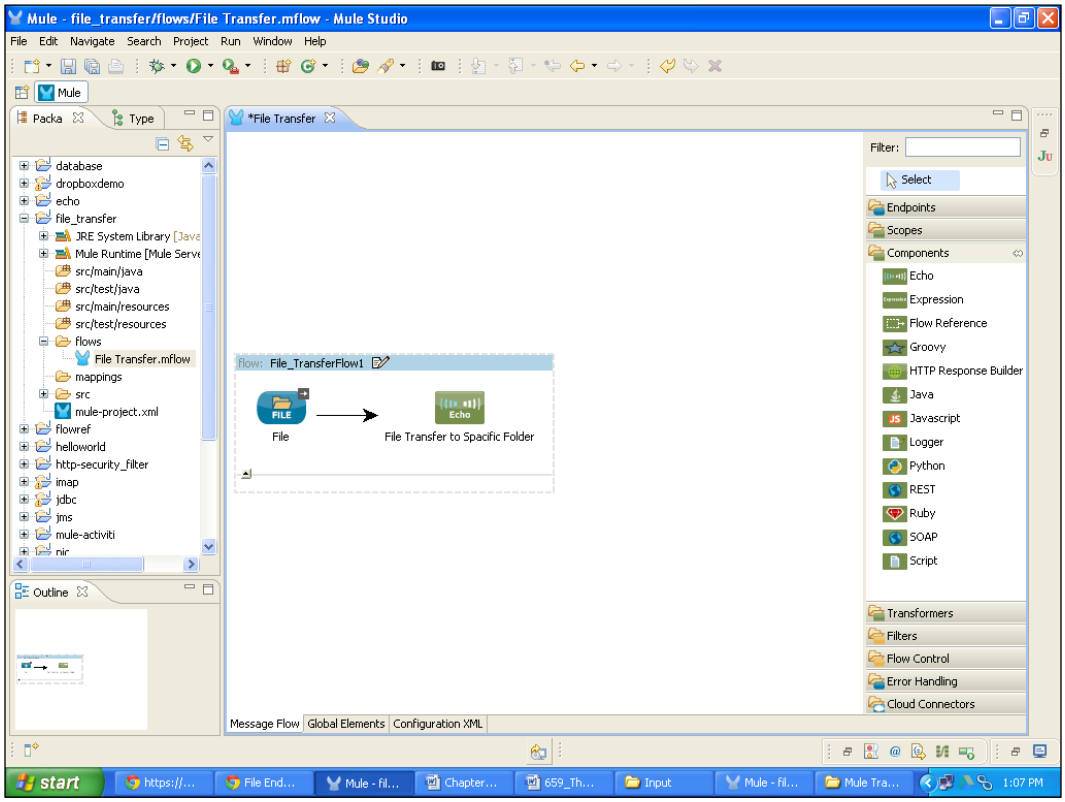
2. To configure the **File** Endpoint, double-click on it and set the source path shown in the following screenshot. In the `D:\`, create a folder called `Order data`. In this folder, create two folders: an `Input` folder and an `Output` folder. In the `Input` folder, place three XML files and these three XML files will be transferred to a specific folder (the `Output` folder).



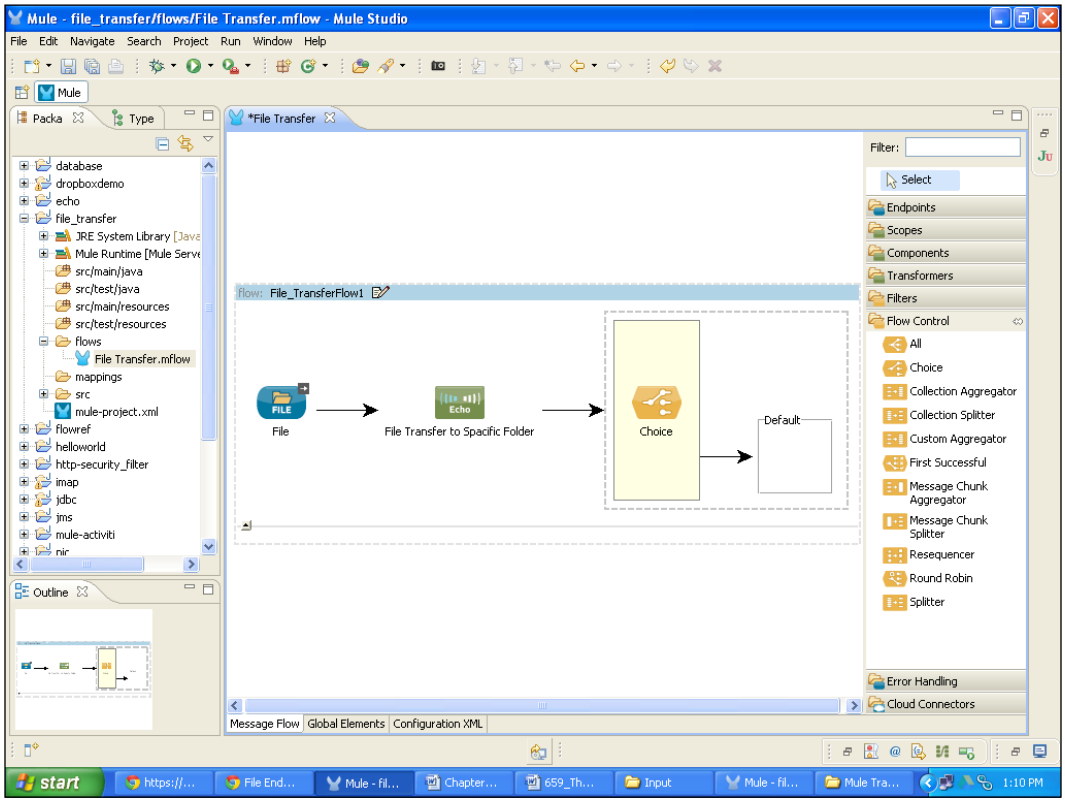
Here, you can configure the File Inbound Endpoint. Once you configure it, click on the **OK** button.



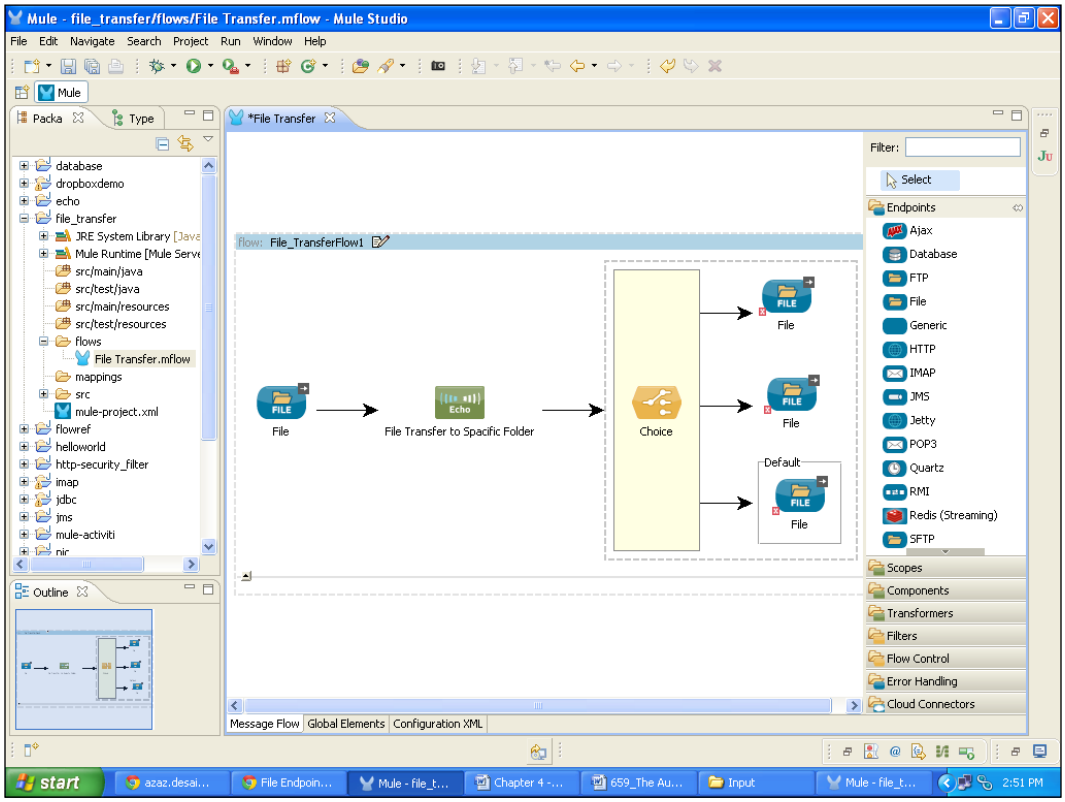
3. For identification or display purposes, we drag the **Echo** component from the palette and drop it on the canvas. Change the **Echo** component name to **File Transfer to Specific Folder**.



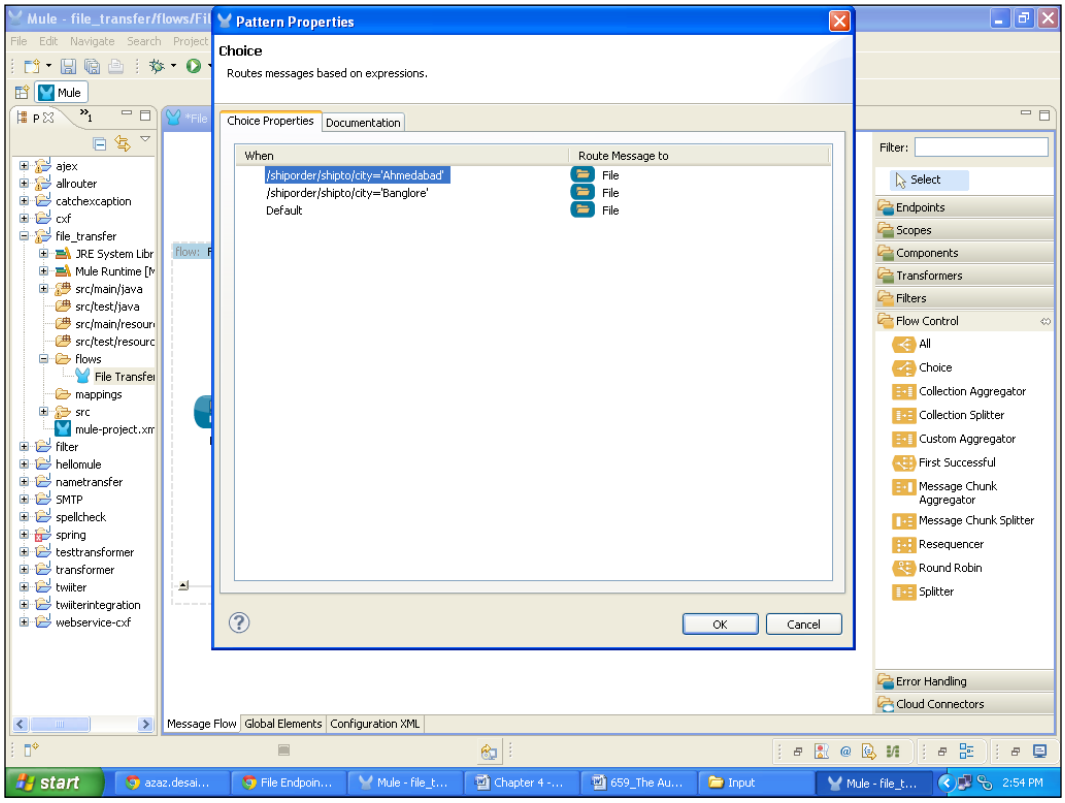
4. To transfer a file to a specific folder, drag the **Choice** Router or Flow Control. In the **Choice** Router, you can define the condition. If the condition matches with the expression, the flow will be executed.



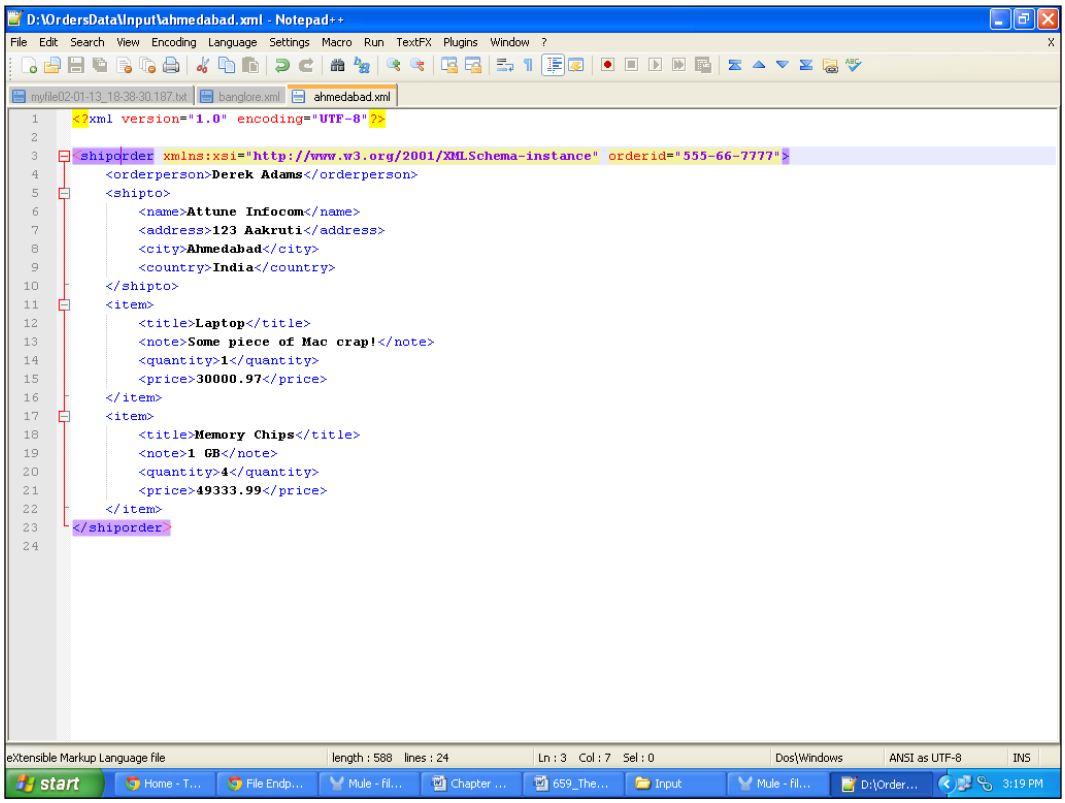
5. To set the destination target, drag three **File** Endpoints from the palette and drop them on the canvas, as shown in the following screenshot:



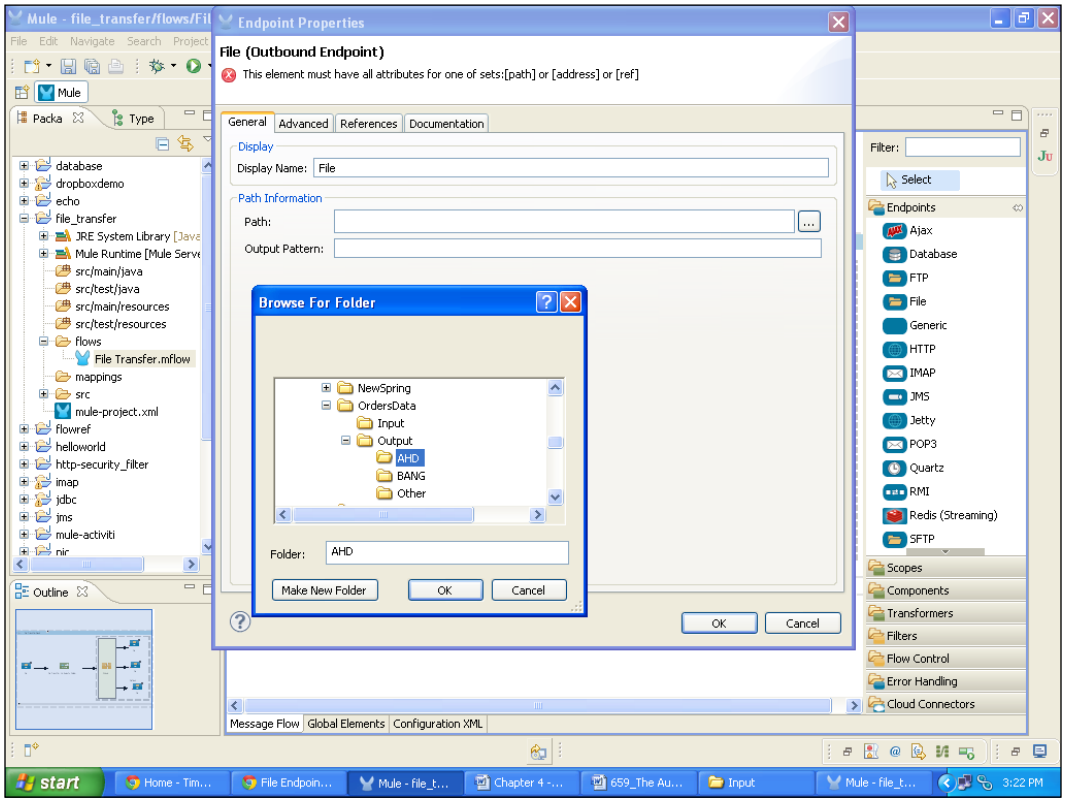
6. To configure the **Choice** Router or Flow Control, double-click on it. Here, you can see the **When** partition; this is our condition area. If the condition is true, it will execute; for example, in the condition `/shiporder/shipto/city='Ahmedabad'`, `shiporder` is a tag which is used in the XML file shown in the following screenshot:



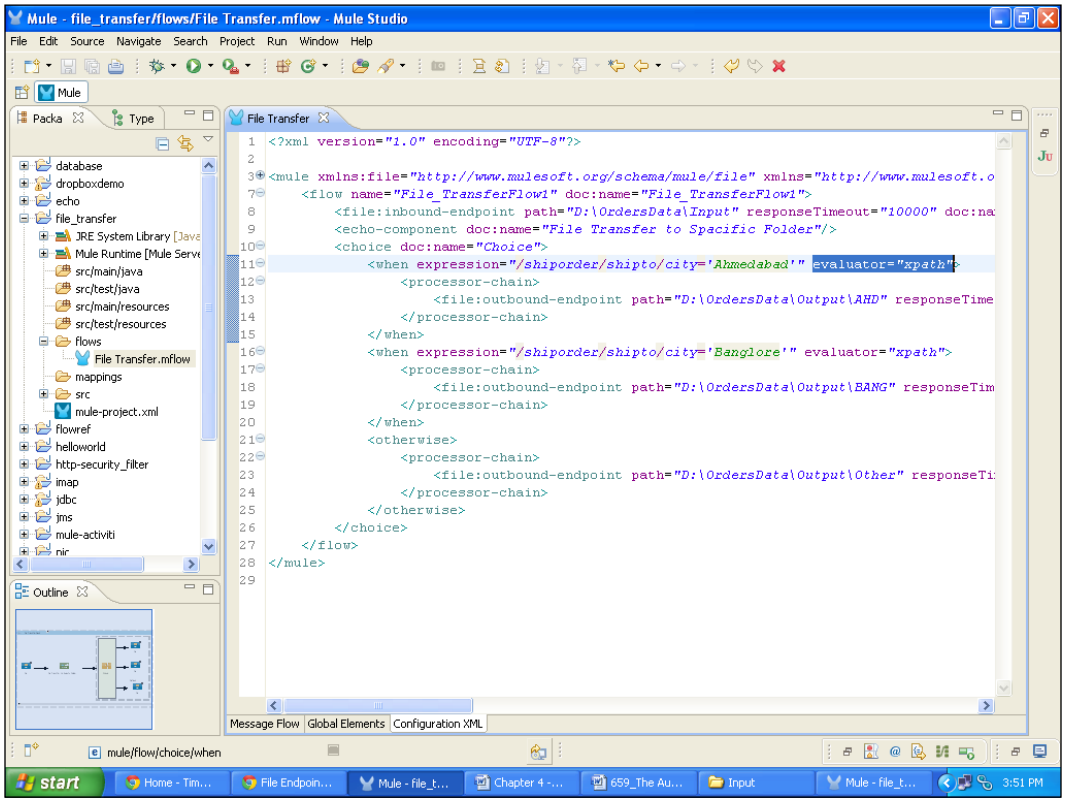
7. This is an XML file, which is used in the `Input` folder.



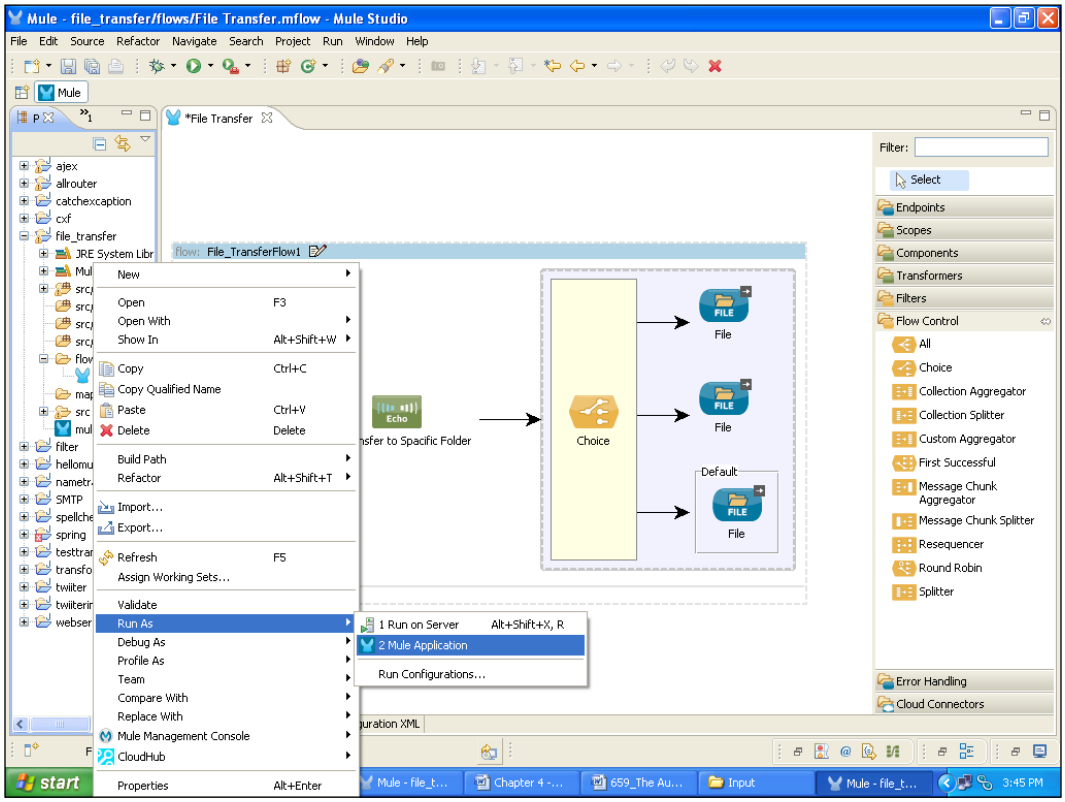
8. Double-click on the first **File** Endpoint to configure it. In the same way, you can configure the other two File Outbound Endpoints, one for the `BANG` folder and the other for the `Other` folder.



9. In the **Choice** Router, you need to add an attribute `evaluator="xpath"` after the expression attribute to read the XML tag, as shown in the following screenshot:



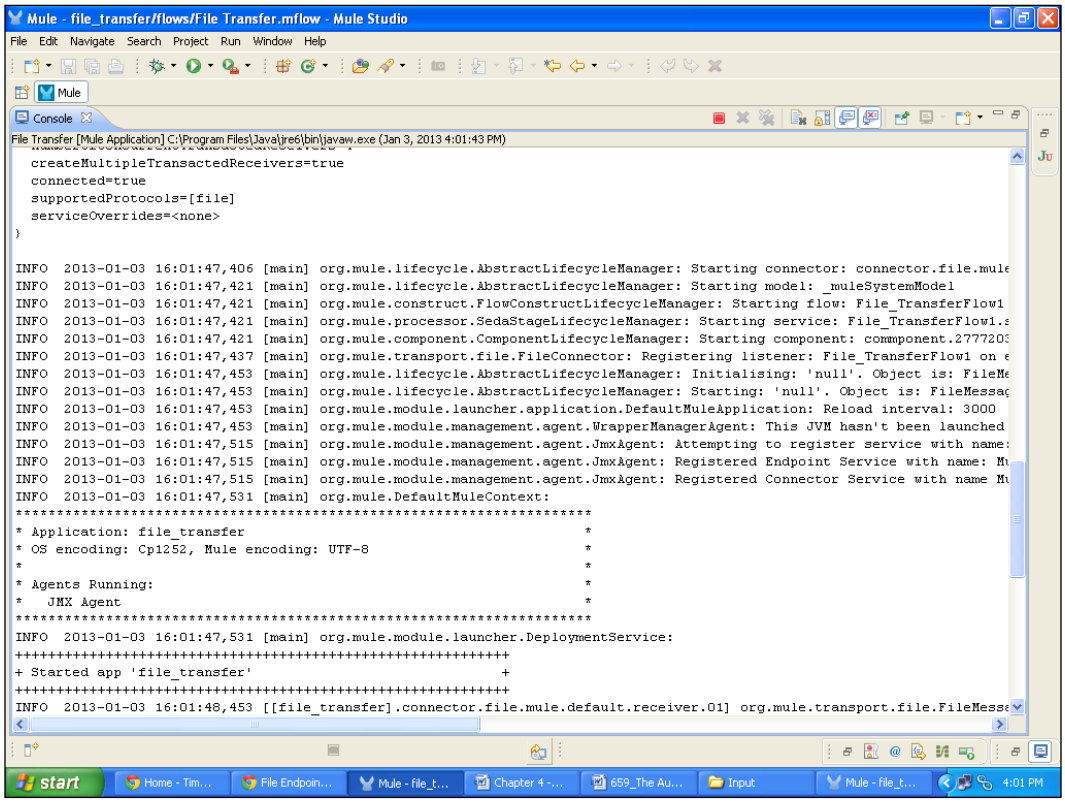
10. To deploy the application code in the Mule server, go to **Run As** | **Mule Application**, and the Mule Server will deploy your application.



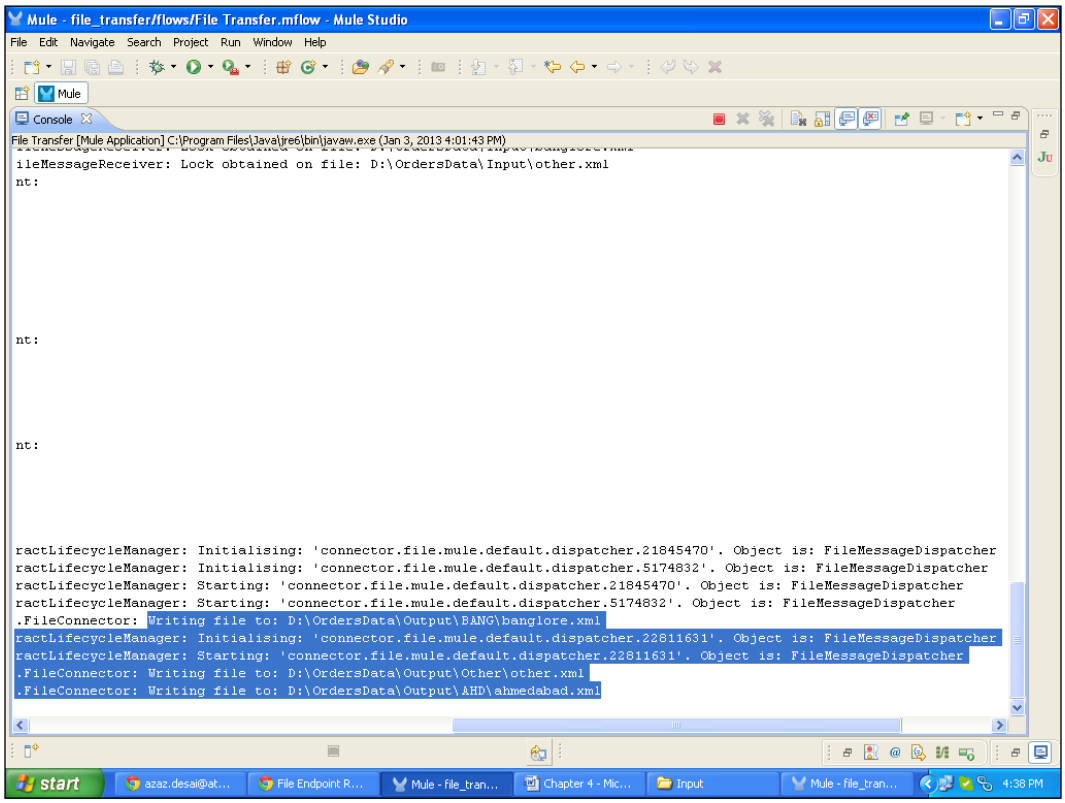
How it works...

Once your application is successfully deployed, you can see the log on the console. All files are transferred to that particular directory.

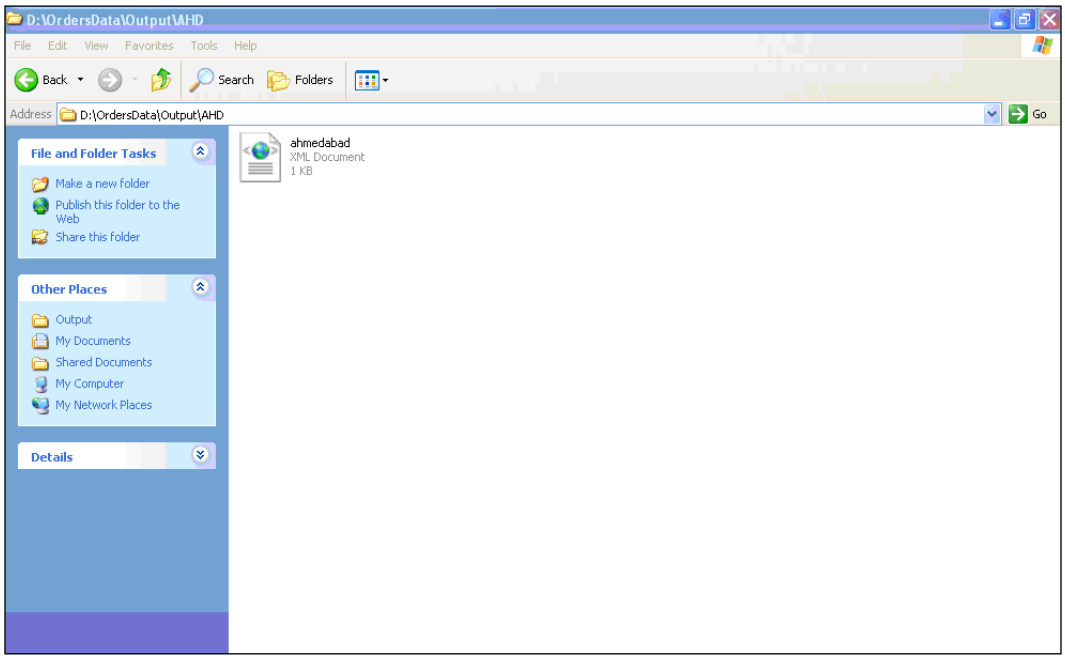
If your application code is successfully deployed, you will see the message `Started app 'file transfer'` on the console.



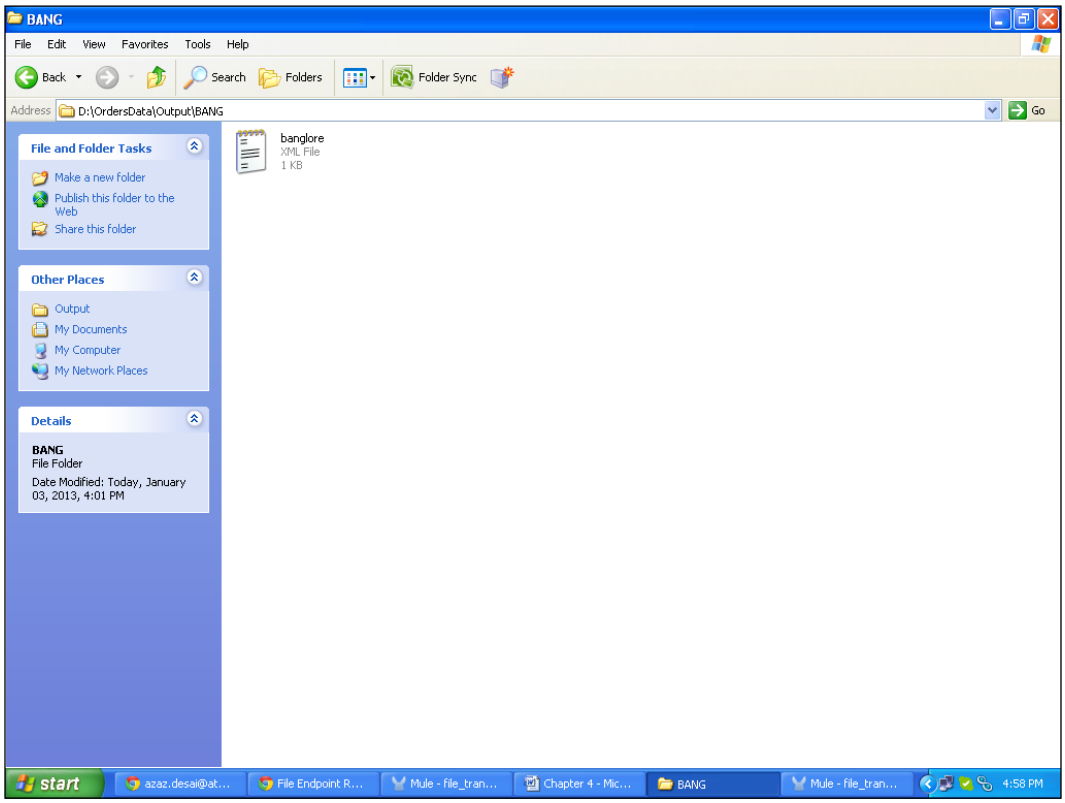
You can see that each file is transferred to a specific folder in the following screenshot:



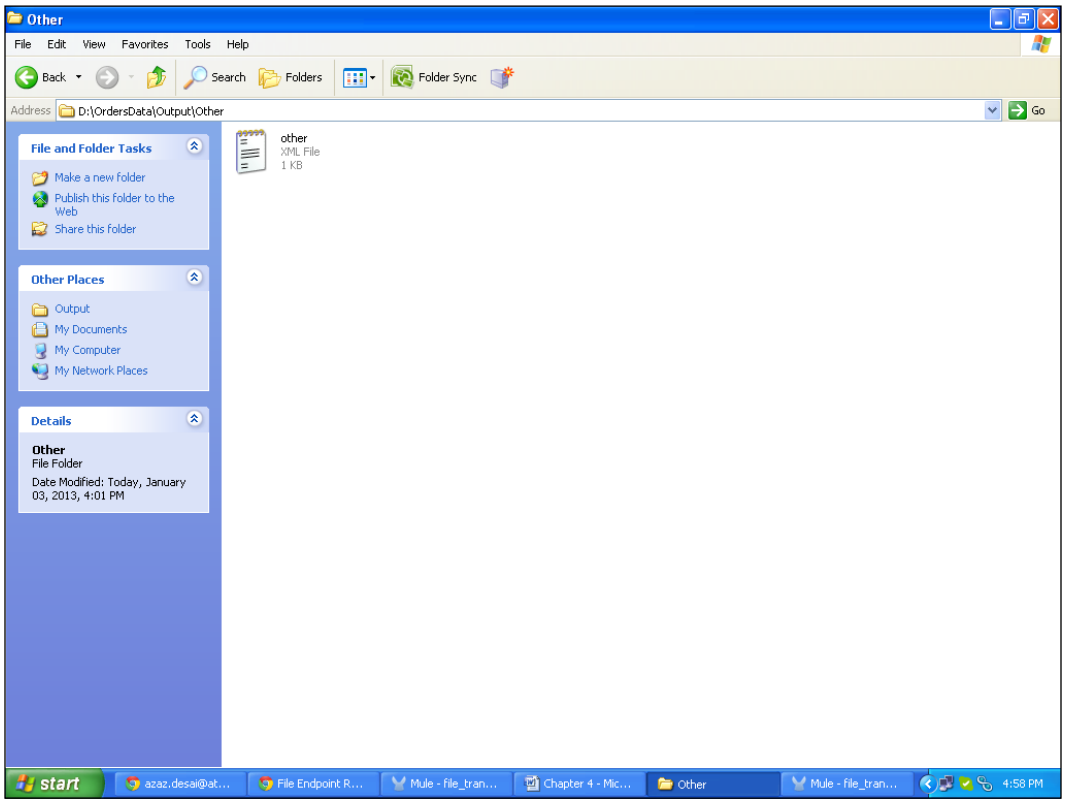
You can see that the first XML file is transferred to the first destination folder:



The second XML file is transferred to the second destination folder:



And, the third XML file is transferred to the third destination folder:



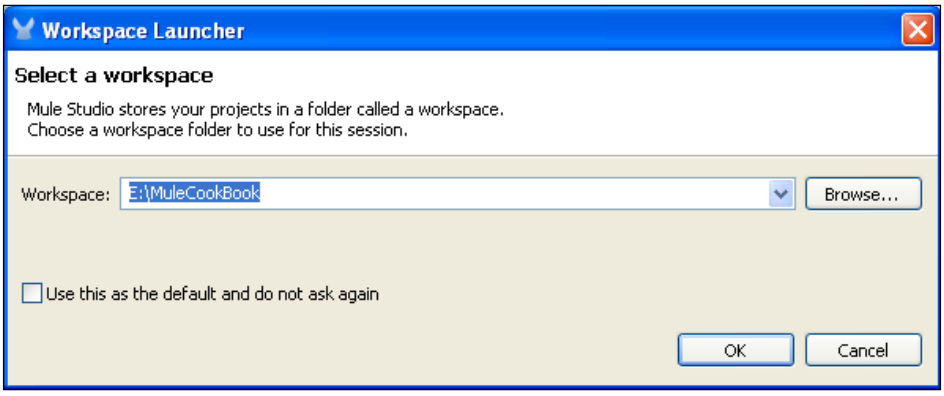
Sending messages asynchronously using the AJAX Endpoint

AJAX stands for **Asynchronous Java and XML**. The AJAX connector allows Mule actions to be sent and received asynchronously to and from the web browser. We will see how to use AJAX Endpoint in this recipe.

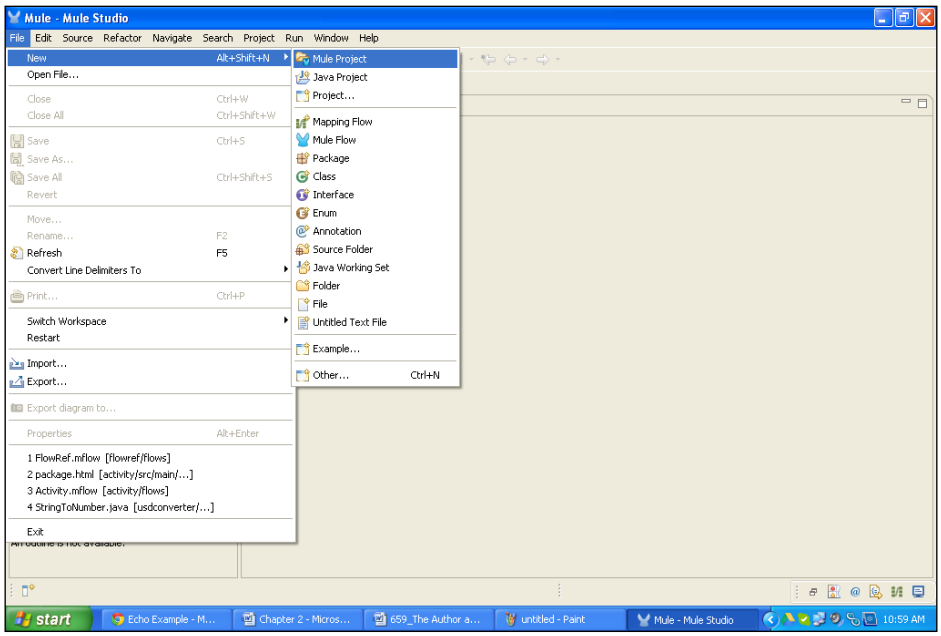
Getting ready

In this section, you will see a Google spell check example and how it works. Here, we use the HTTP Endpoint, the AJAX Endpoint, the Echo component, an Object to XML transformer, and the XSLT transformer. In this example you will see how to configure the AJAX component.

1. Open Mule Studio and enter the workspace name as shown in the following screenshot:



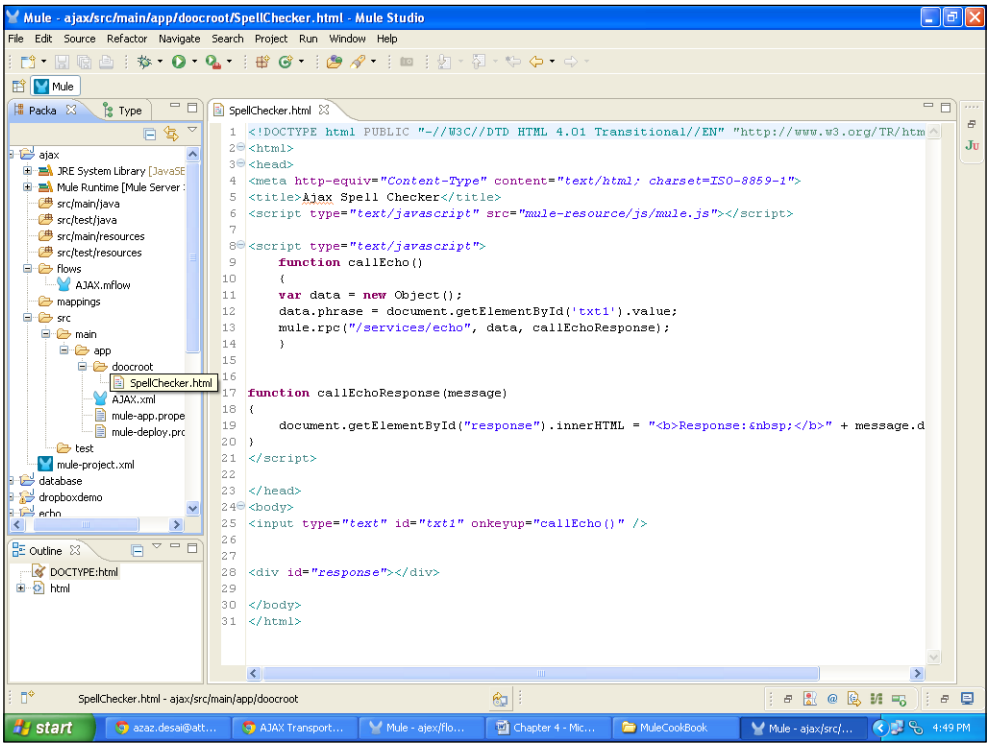
2. To create a new project, go to **File** | **New** | **Mule Project**. Enter the project name called `AJAX` and click on **Next** and then on **Finish**. Your new project is created. Now you can start the implementation.



How to do it...

In the following section, you'll see how the spell check example works, and you will also see how to configure each component in flow.

1. Create a folder inside `src/main/app` called `docroot` as shown in the following screenshot. Inside that folder, create an XML file called `SpellCheck.html`. In this file you have to create an HTML page, which will be displayed on the browser. We call the JavaScript function, `callEcho`, through the `onkeyup` function.

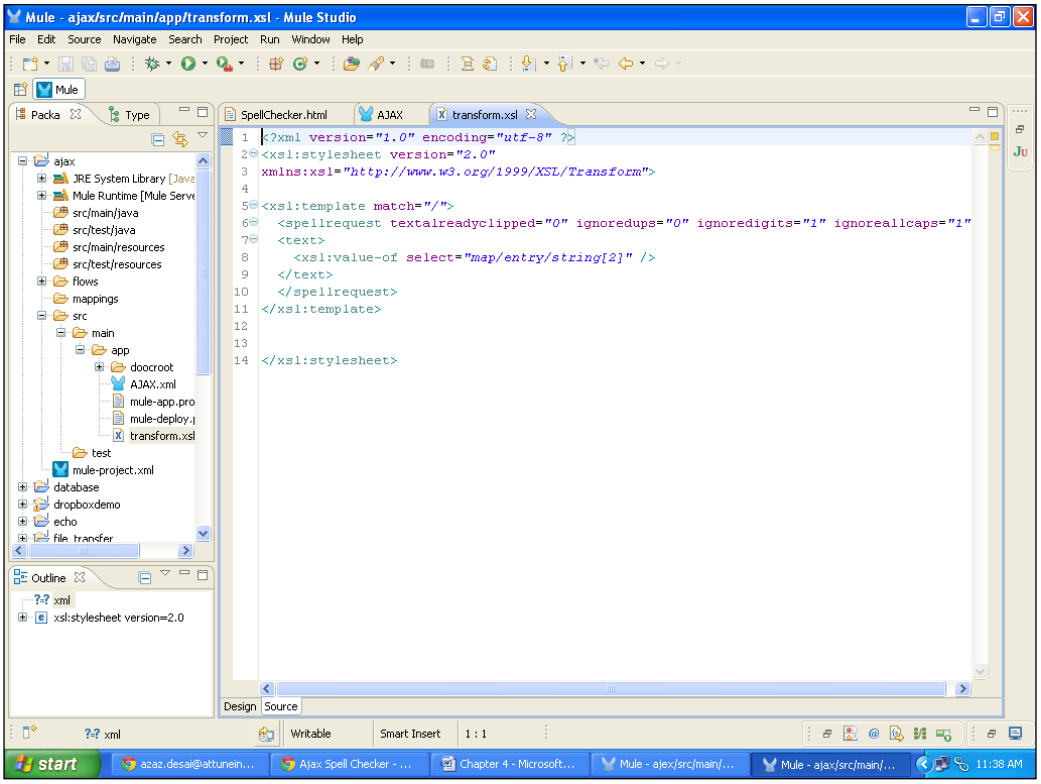


This is a `SpellCheck.html` file; we have to put this file inside the `src/main/app/docroot` location:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">
<title>Ajax Spell Checker</title>
<script type="text/javascript" src="mule-resource/js/mule.js"></
script>
```

```
<script type="text/javascript">
  function callEcho()
  {
    var data = new Object();
    data.phrase = document.getElementById('txt1').value;
    mule.rpc("/services/echo", data, callEchoResponse);
  }
function callEchoResponse(message)
{
    document.getElementById("response").innerHTML =
"<b>Response: </b>" + message.data + "\n";
}
</script>

</head>
<body>
<input type="text" id="txt1" onkeyup="callEcho()" />
   <div id="response"></div>
   </body>
</html>
```

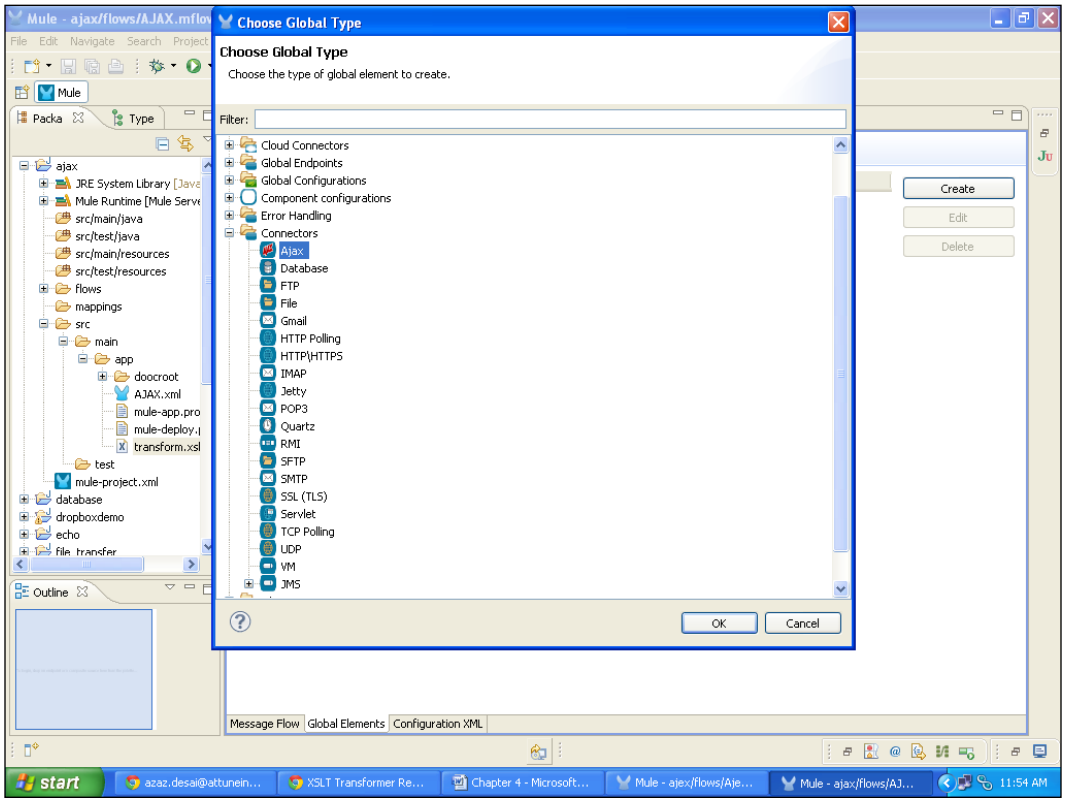
2. Create a `transform.xsl` file inside `src/main/app`; this is how it will look:



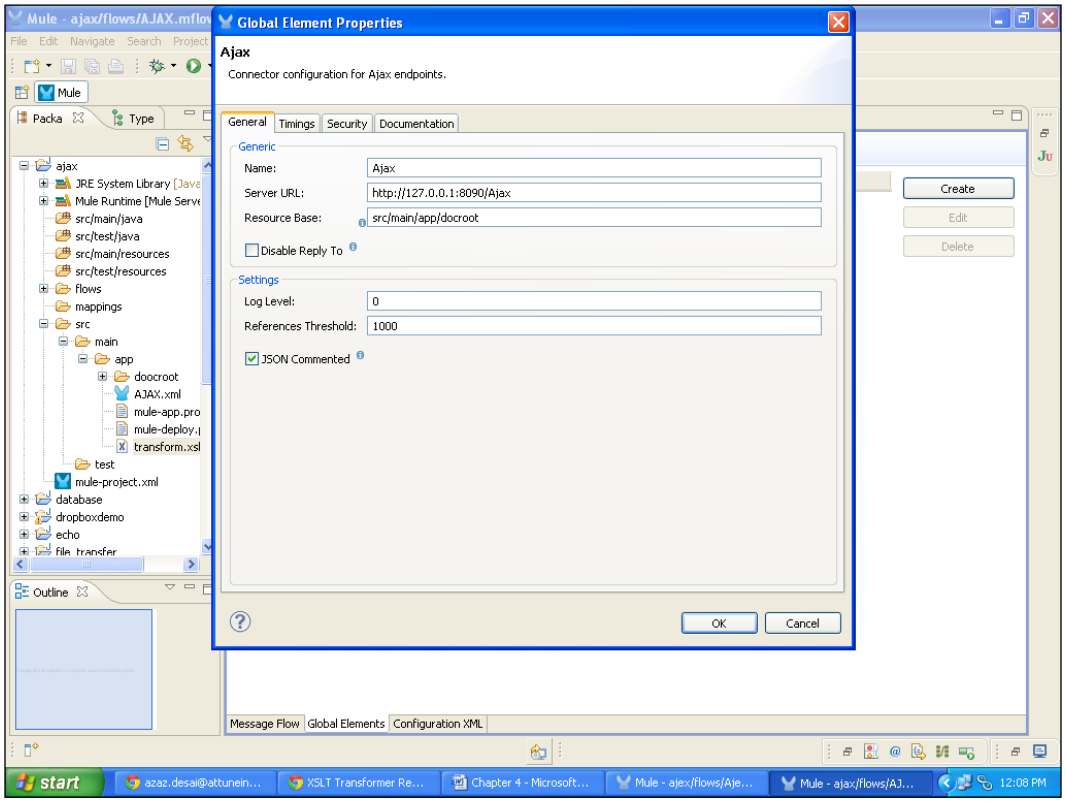
The `transform.xsl` file provides instructions for extracting data from incoming messages and translating that data into a form that applications can digest.

```
<?xml version="1.0" encoding="utf-8" ?>
<xsl:stylesheet version="2.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
  <spellrequest textalreadyclipped="0" ignoredups="0"
ignoredigits="1" ignoreallcaps="1">
  <text>
    <xsl:value-of select="map/entry/string[2]" />
  </text>
  </spellrequest>
</xsl:template>
</xsl:stylesheet>
```

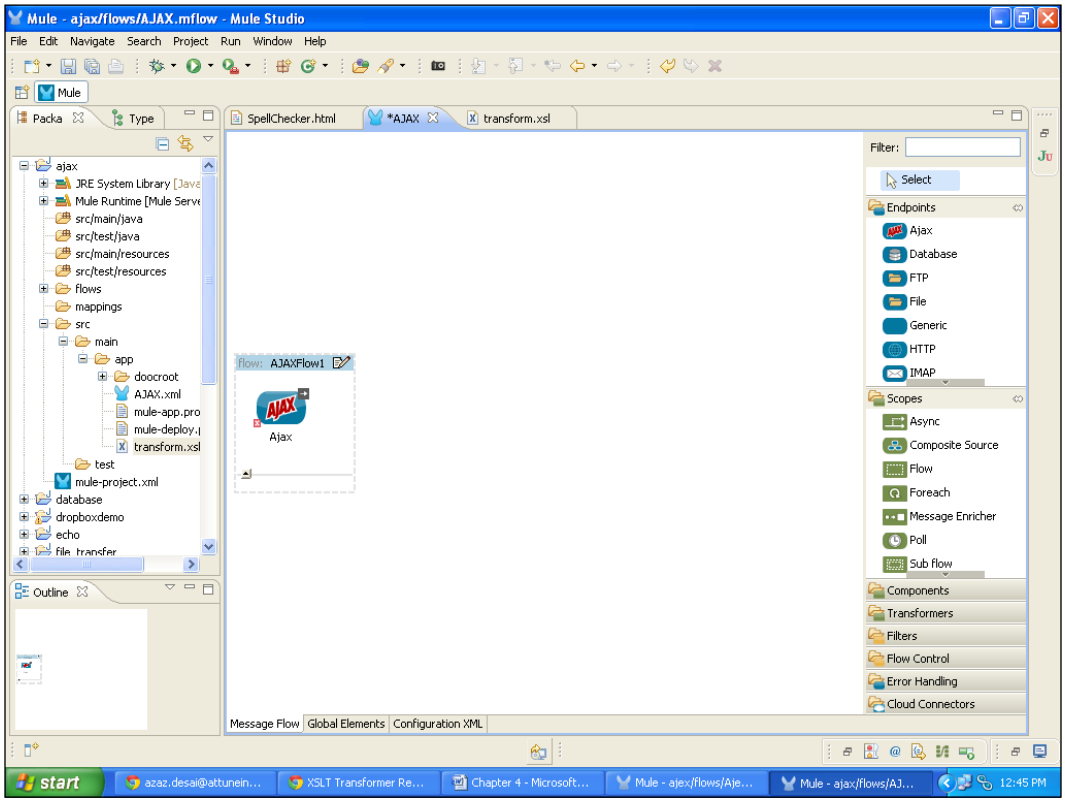
3. To create a flow, go to the `AJAX.mflow` file. Then, go to the **Global Elements** tab; click on **Create** and go to **Connectors** | **Ajax**, and then click on **OK**.



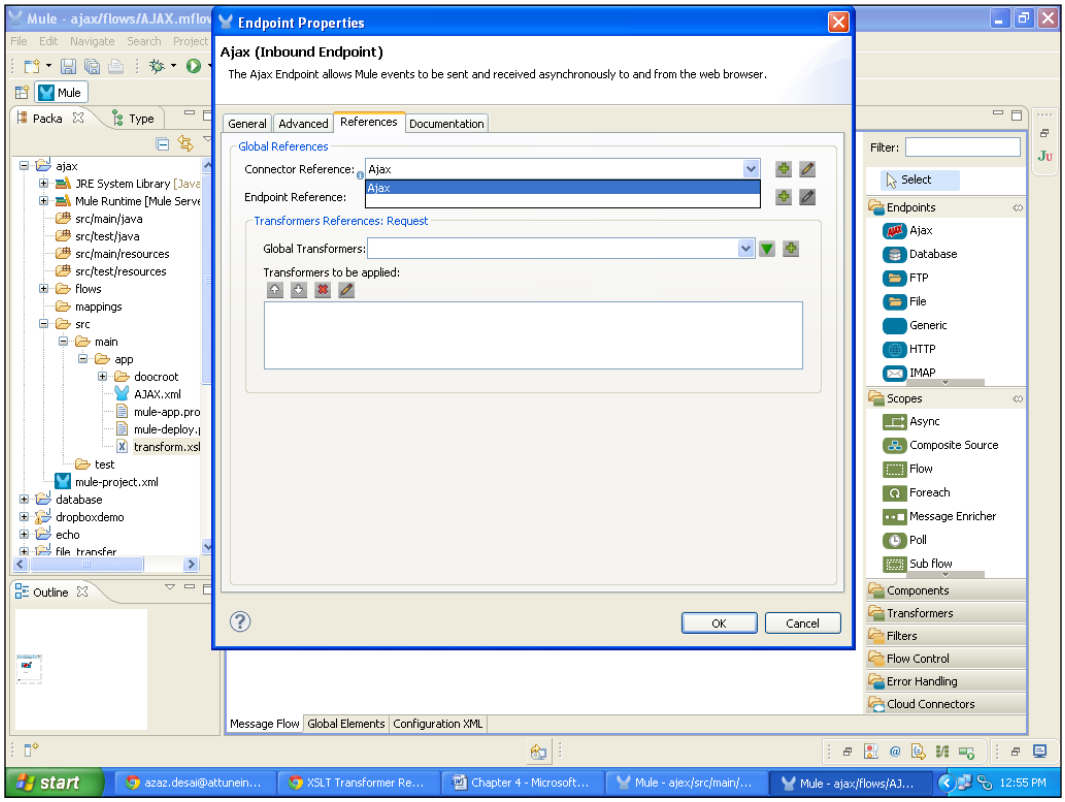
4. To configure the AJAX connector, enter the server URL `http://127.0.0.1:8090/Ajax` in the **Server URL:** field. In **Resource Base**, enter the `.html` file path, which is located at `src/main/pp/docroot`.



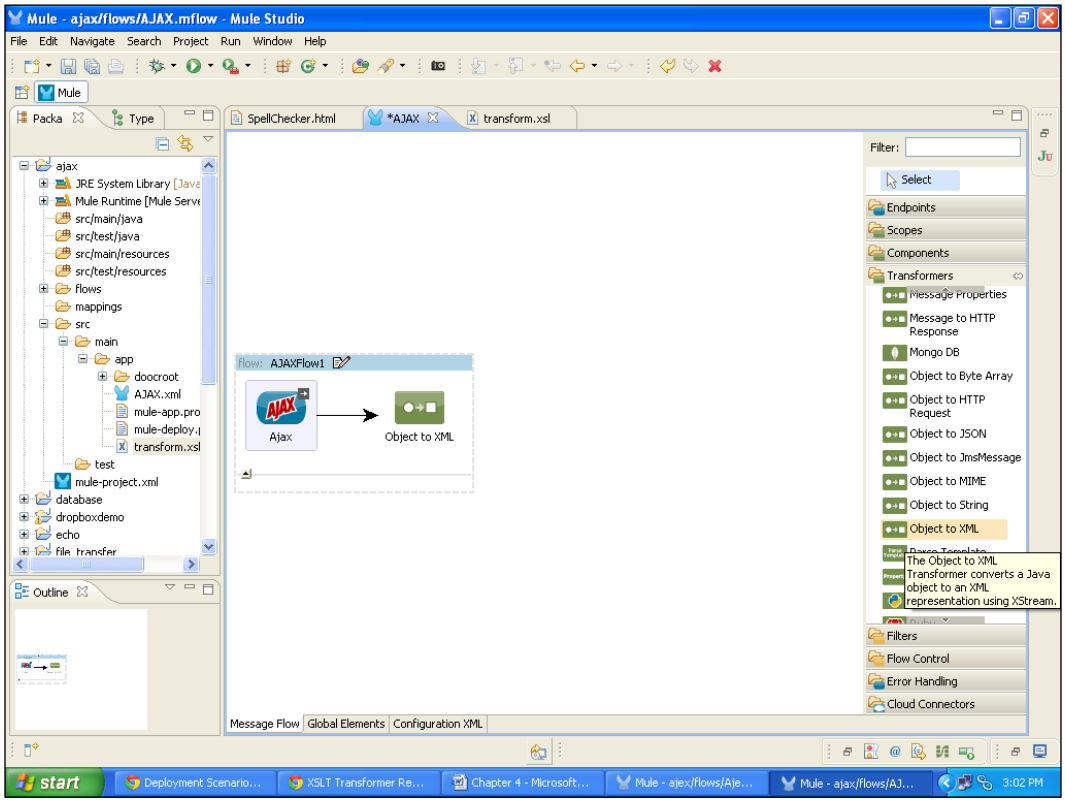
5. Go to the message flow, drag the **AJAX** Endpoint from the palette, and drop it on the canvas.



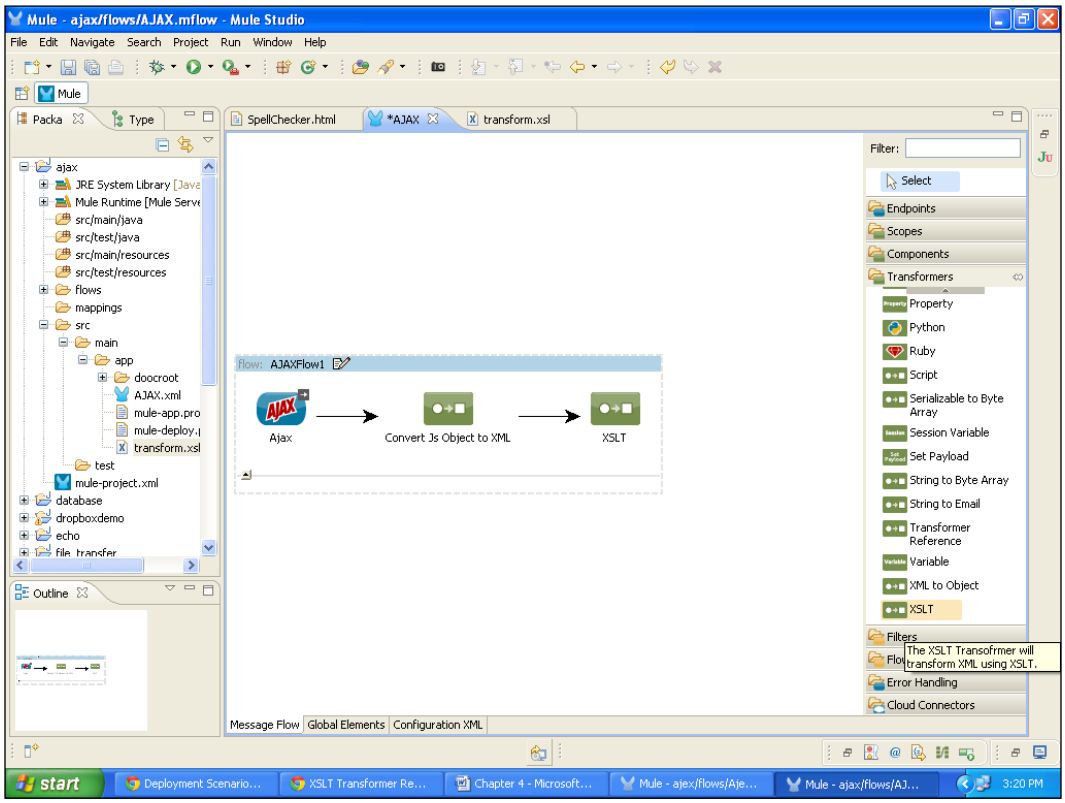
6. Double-click on the **Ajax** Endpoint to configure it. You have to enter a channel path, `/services/echo`, the same path which you have assigned in the `SpellCheck.html` file. Go to the **Reference** tab, select the connector reference name and click on the **OK** button.



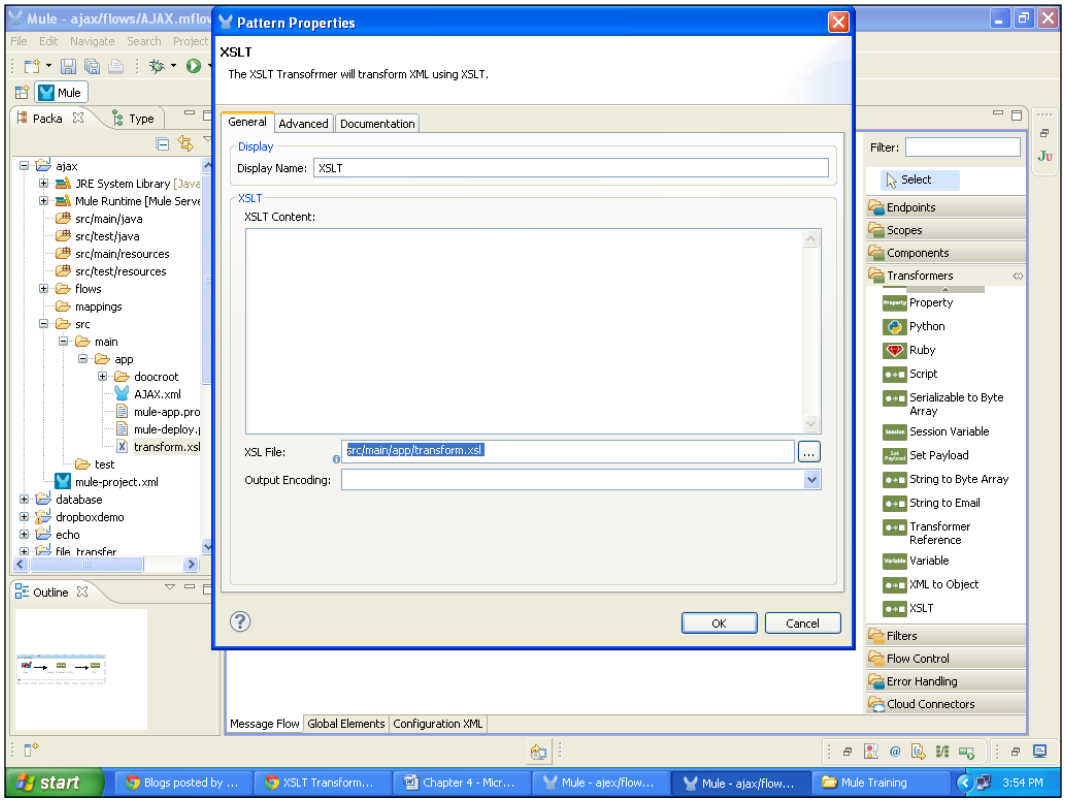
7. Drag the **Object to XML** transformer from the palette and drop it on the canvas. It is used to convert a JavaScript object to XML.



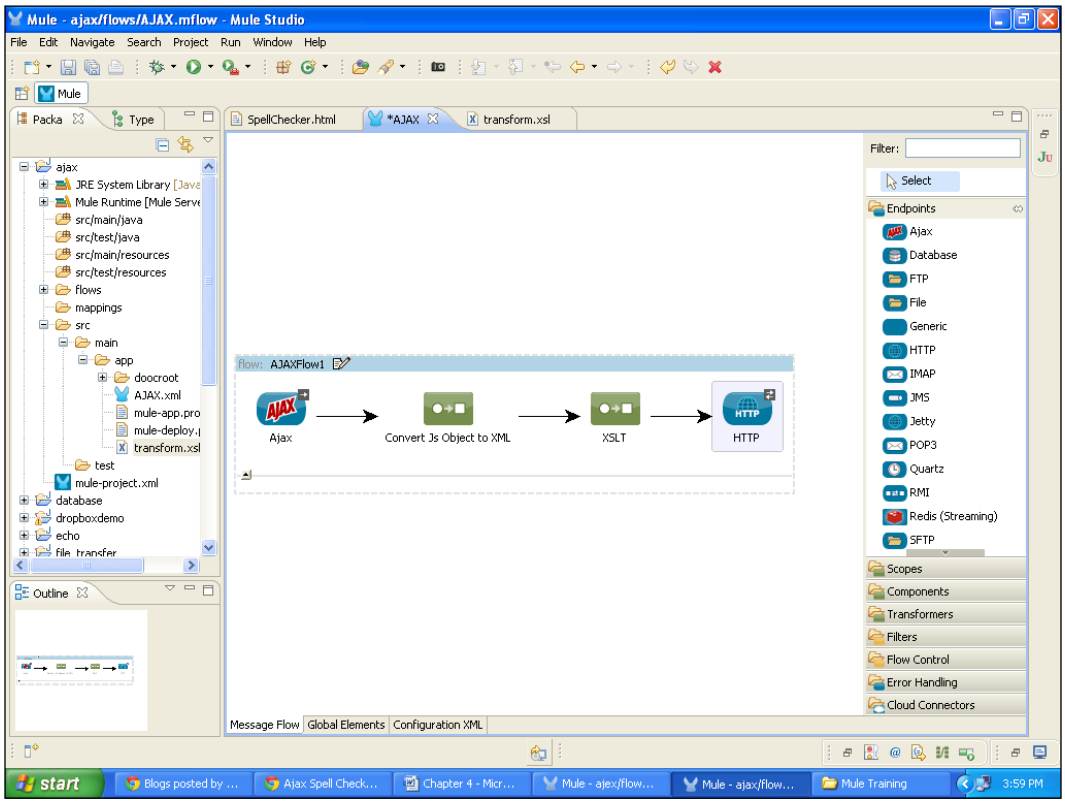
8. Drag the **XSLT** transformer from the palette and drop it on the canvas.



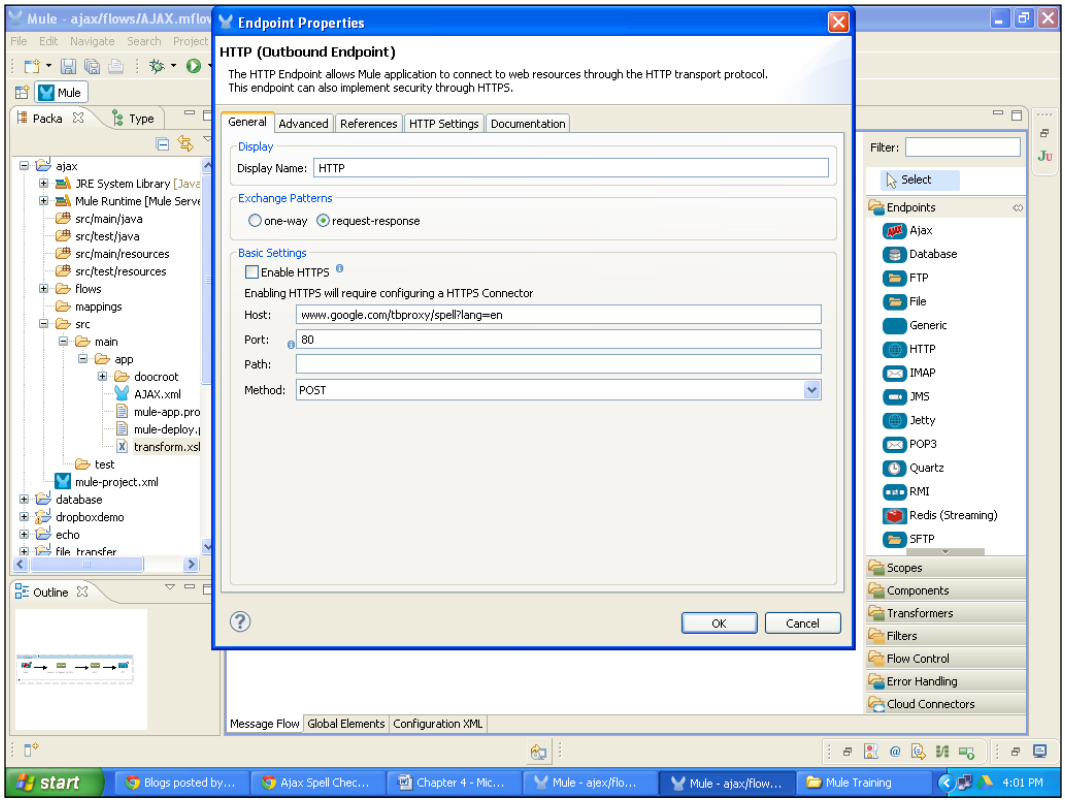
9. Double-click on the **XSLT** transformer to configure it. In **XSL File:**, enter the path of the `.xsl` file, which is located in `src/main/app` and click on the **OK** button.



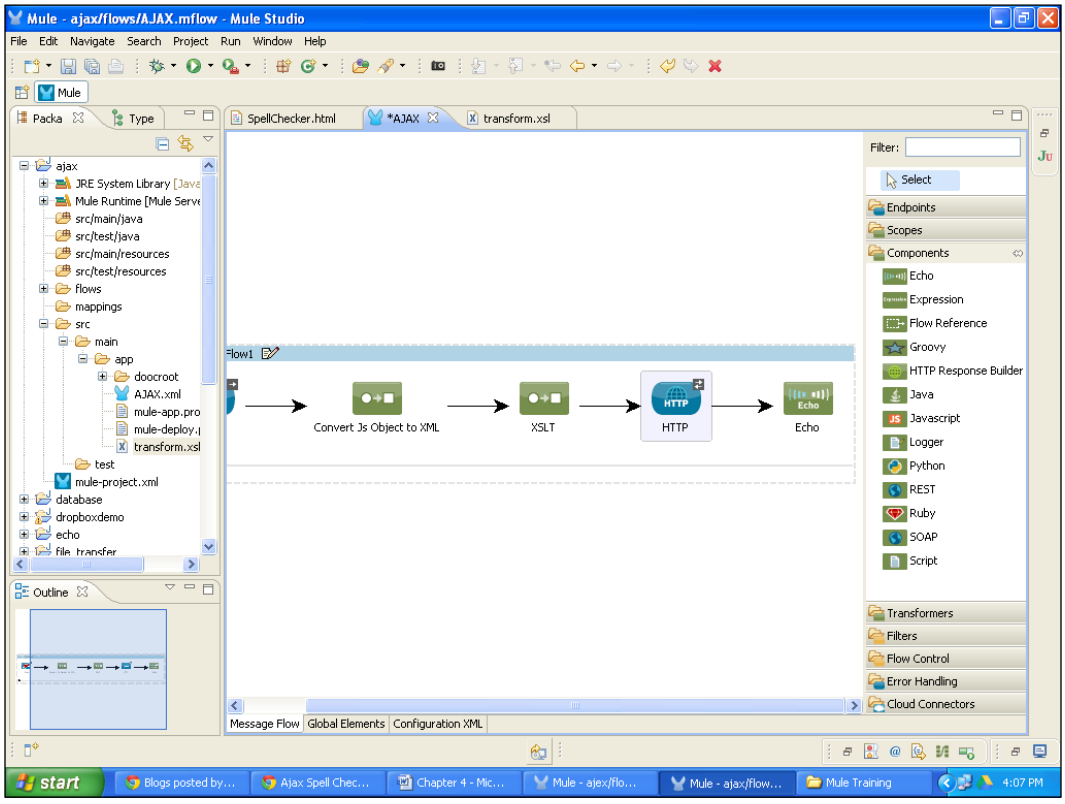
10. Drag the **HTTP** Endpoint from the palette and drop it on the canvas.



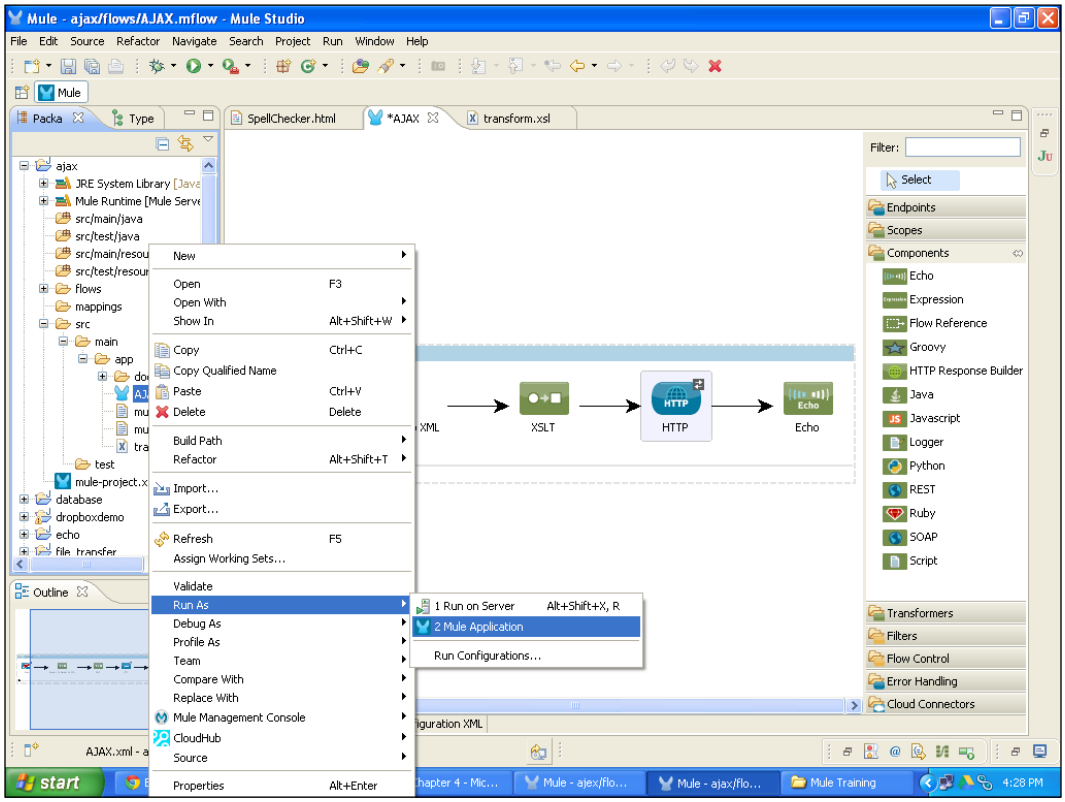
11. Double-click on the **HTTP** Endpoint to configure it, and in the **Host:** field enter the spell check API link. Select the **POST** method.



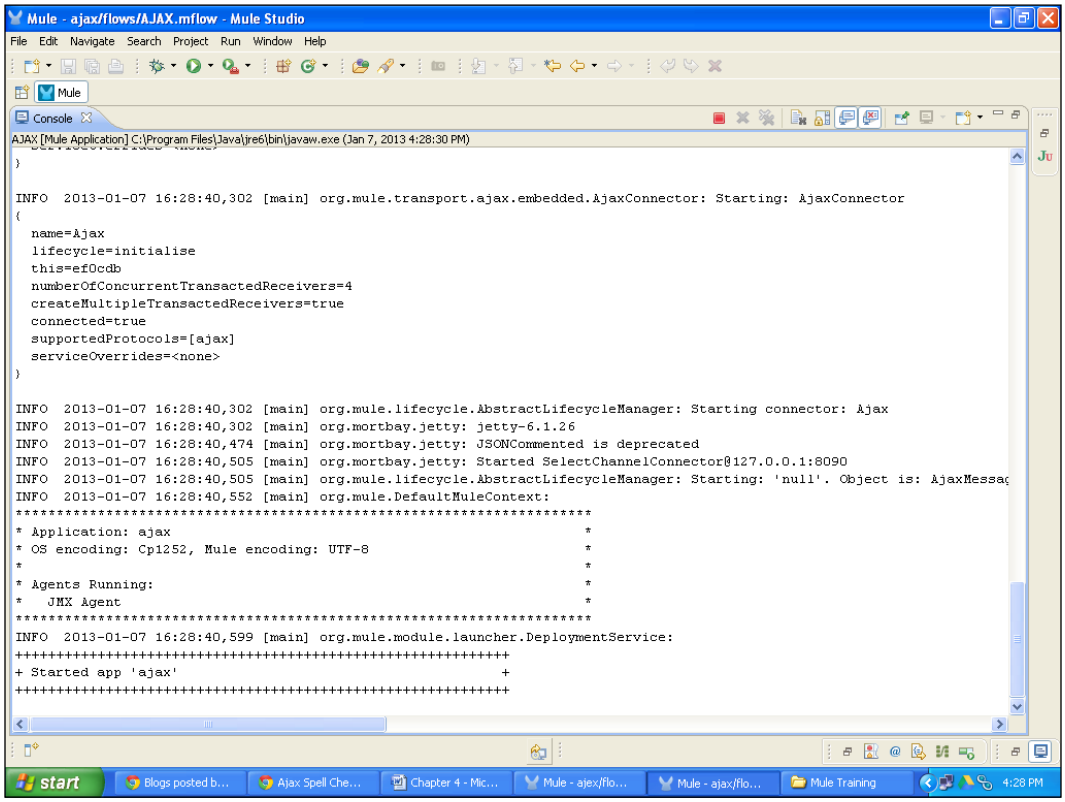
12. For display purposes, drag the **Echo** component from the palette and drop it on the canvas.



13. Now we are ready for the deployment. If you haven't saved your application code, first save it. After saving your project, right-click on the `AJAX.mflow file` and go to **Run As** | **Mule Application**.

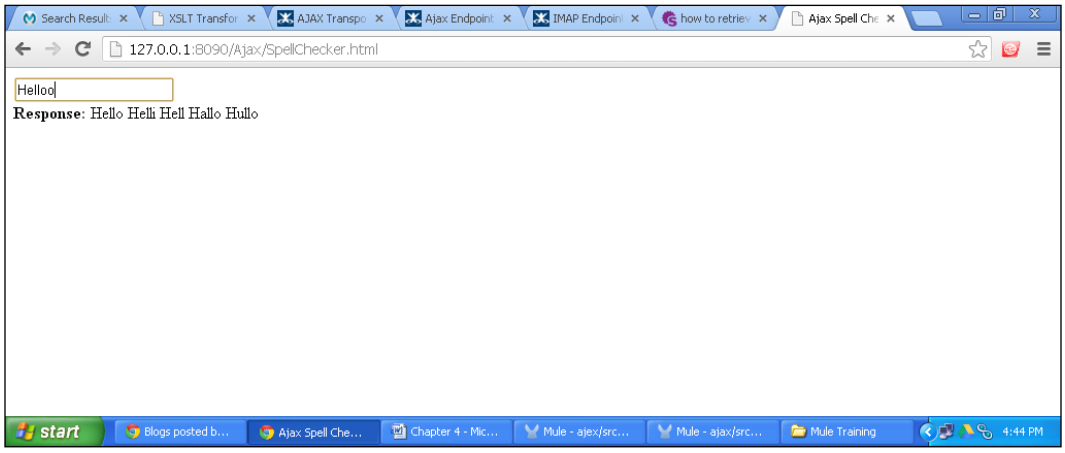


14. If your application code is successfully deployed, you will see the message `Started app 'ajax '` on the console.



How it works...

Paste this URL in your browser: `http://127.0.0.1:8090/Ajax/SpellChecker.html`, and type any word into the textbox. You can see the related search results through the HTTP response by the Google spell check API.



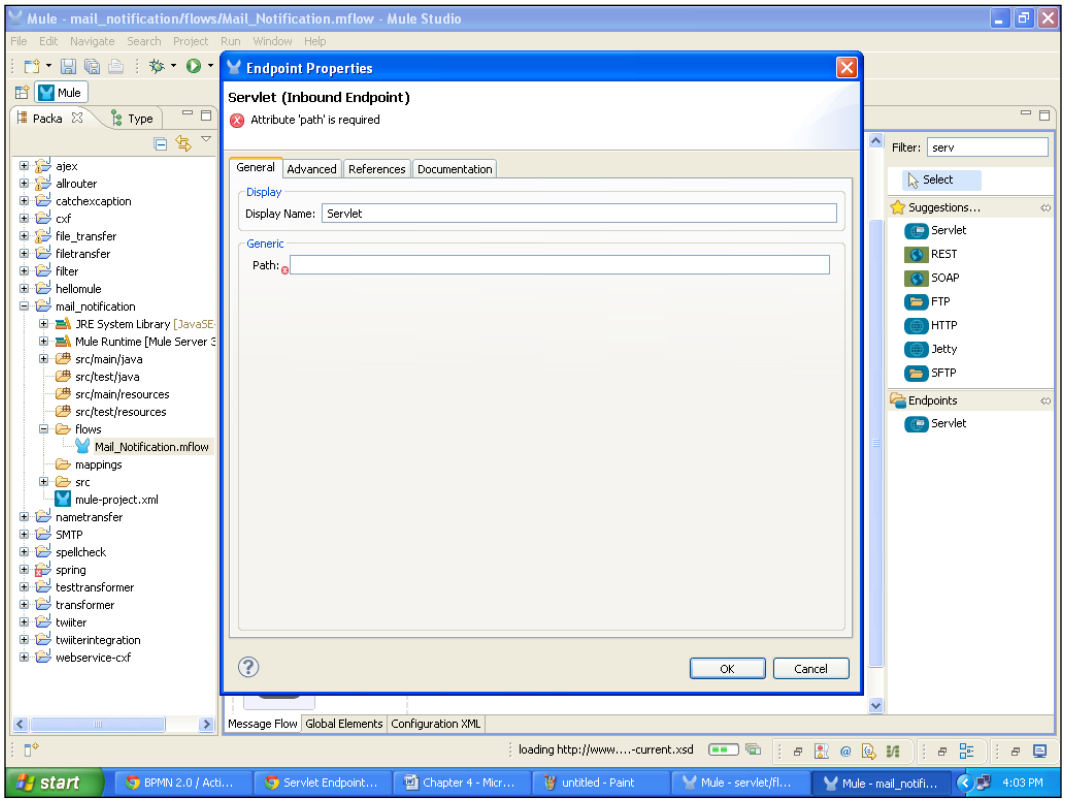
Using the Servlet Endpoint to listen to events or messages from servlet requests

Identify the path to the servlet through which the event or message is received. The Path property can be set from the General tab. We will see each tab that is used in the Servlet Endpoint. The Servlet Endpoint contains four tabs:

- **General**
- **Advanced**
- **References**
- **Documentation**

Getting ready

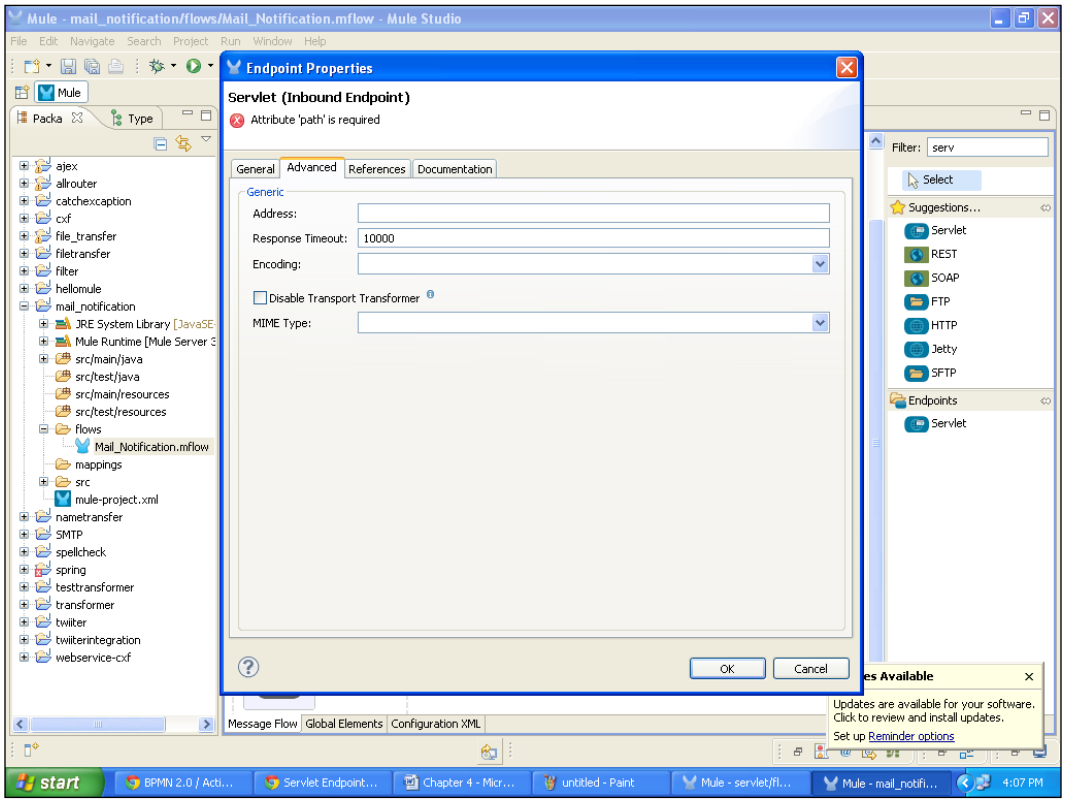
The **General** tab contains two fields: one is **Display Name:** and the other is **Path:**. The display name is used for identity purposes, that is, to identify the channel over which your Servlet Endpoint is going to communicate with the client web page. Here you have to assign the path, and use the same path for the browser.



How to do it...

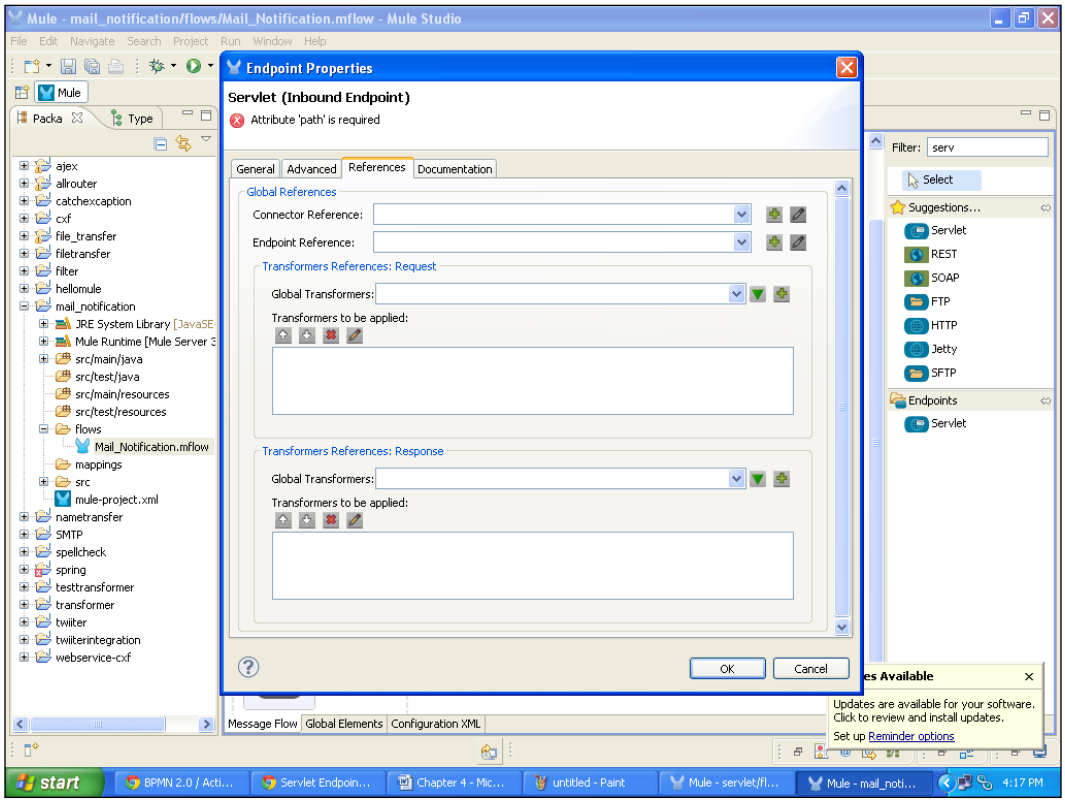
In this section, you will see how to configure the **Servlet** Endpoint and how to use it.

In the **Advanced** tab, enter the address for the Endpoint; for example, `localhost:8080`. Response timeout identifies how long the Endpoint must wait for a response. The default is **10000** ms.



How it works...

The **References** tab is used when you are creating a global element. If you use a global element to configure the **Server** Endpoint, you have to configure the **References** tab.



5
Transformers

In this chapter, we will cover the different types of transformers. You will learn about the following topics:

- Configuring the JSON-to-Object transformer
- Configuring the Object-to-XML transformer
- Configuring the Message and Variable transformers
- Creating the custom transformer
- Understanding the DataMapper transformer

Introduction

A **transformer** is used for converting a message from one format to another. You can create a custom transformer as well, which we will see later. There are different types of transformers, such as the **Java** transformer, the **DataMapper** transformer, the **XSLT** transformer, and the **Append String** transformer. Standard transformers are easy to use compared to custom transformers. For example, we use the Append String transformer to append a string text to the original message payload with a single line of code, as follows:

```
<append-string-transformer name="Greeting" message=", How are you?"/>
```

If the original string message is `Hello`, it will convert it to `Hello, How are you?`.

Configuring the JSON-to-Object transformer

The JSON-to-Object transformer is used to transform a JSON string to object data and vice versa. Using this transformer, we convert JSON data into an object type.

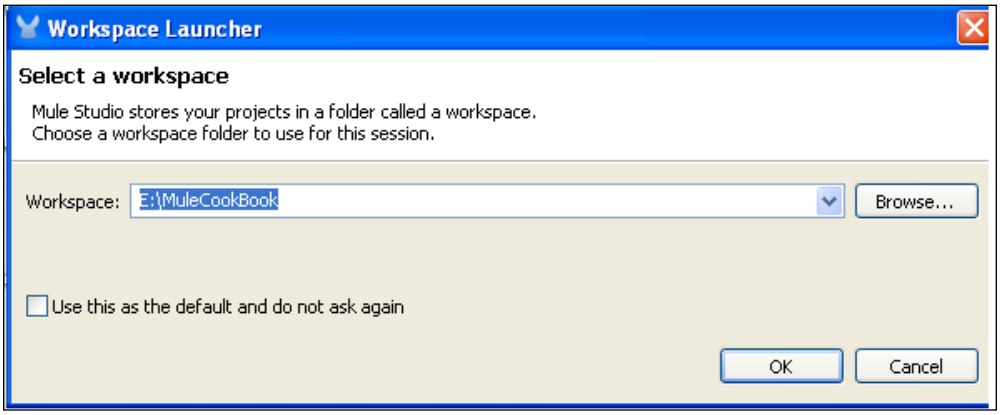
Getting ready

In this example, you will see how to store data in the database using the JSON-to-Object transformer. We need four components for this: the File Endpoint, the JSON-to-Object transformer, the Collection Splitter, and the Database Endpoint. You have to install PostgreSQL on your system.

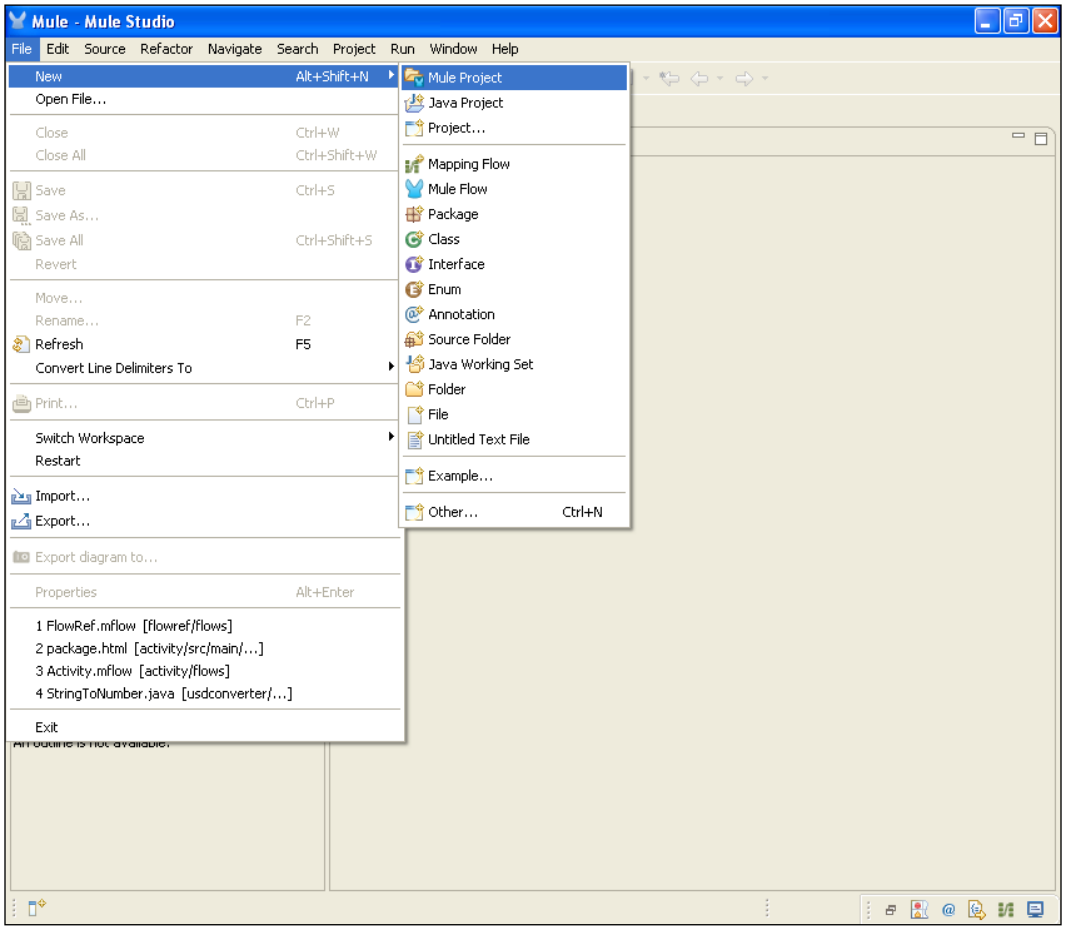
How to do it...

In this section, you will see how to use the JSON-to-Object transformer in Mule Studio.

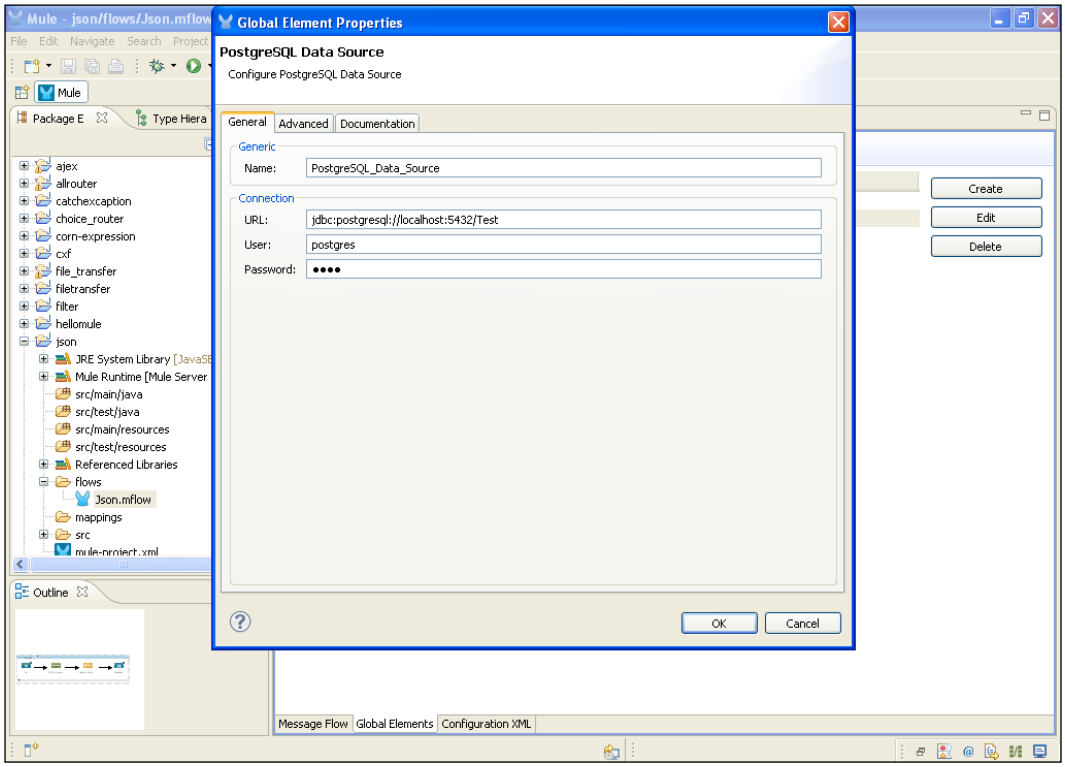
1. Open Mule Studio and enter a name for the workspace as shown in the following screenshot:



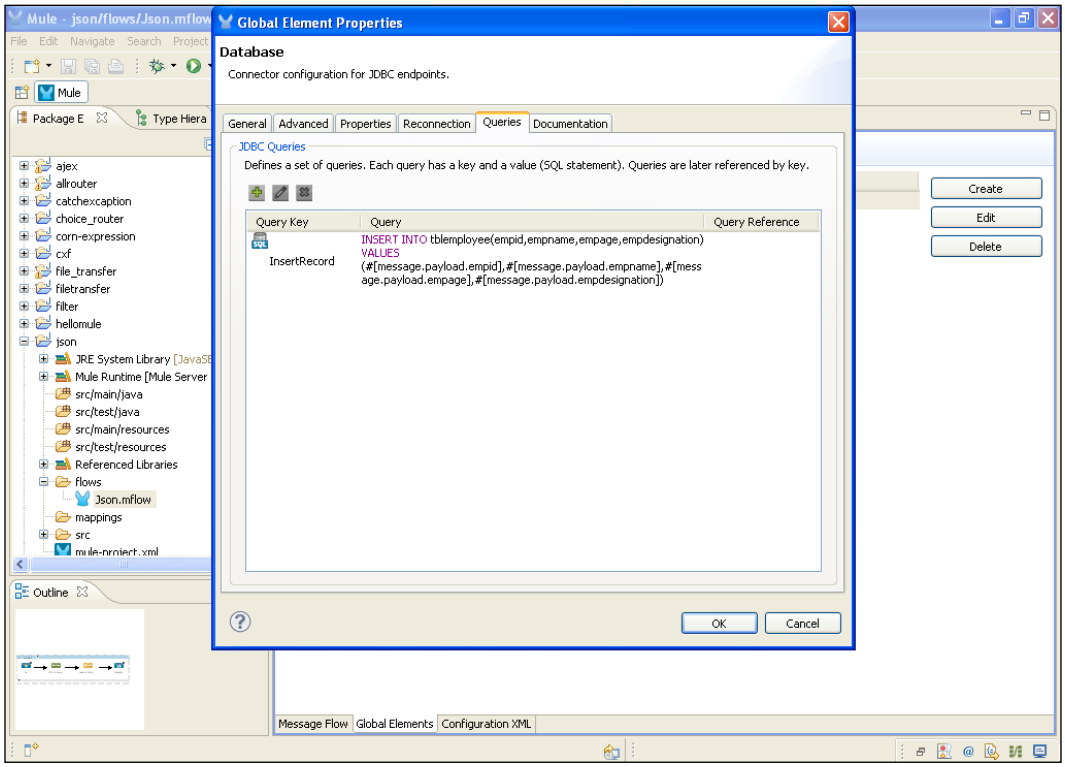
2. To create a new project, go to **File** | **New** | **Mule Project**. Enter the project name as `JSON` and click on **Next** and then on **Finish**. Your new project is created and now you have to start the implementation.



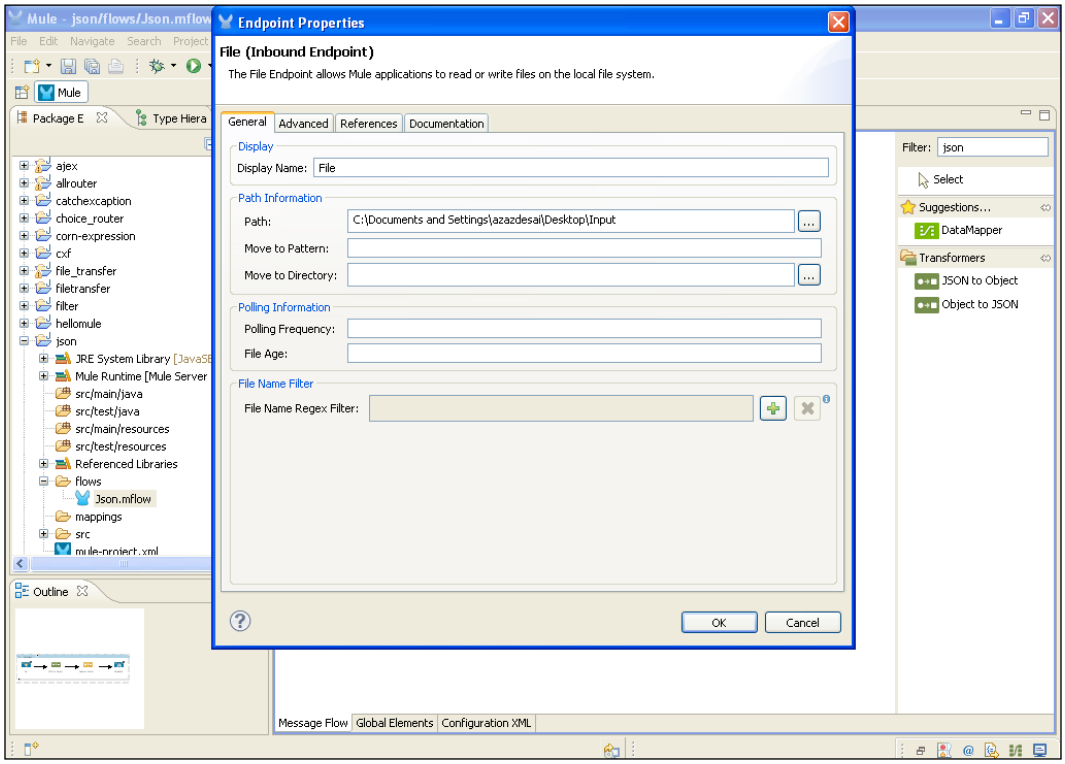
3. Go to the `Json.mflow` file, navigate to the **Global Elements** tab, and click on the **Create** button. Go to **Data Sources** | **PostgreSQL Data Sources**. In the **URL:** textbox, enter the database name that was created in PostgreSQL. Here, we entered `Test`. Enter the PostgreSQL username and password and click on the **OK** button.



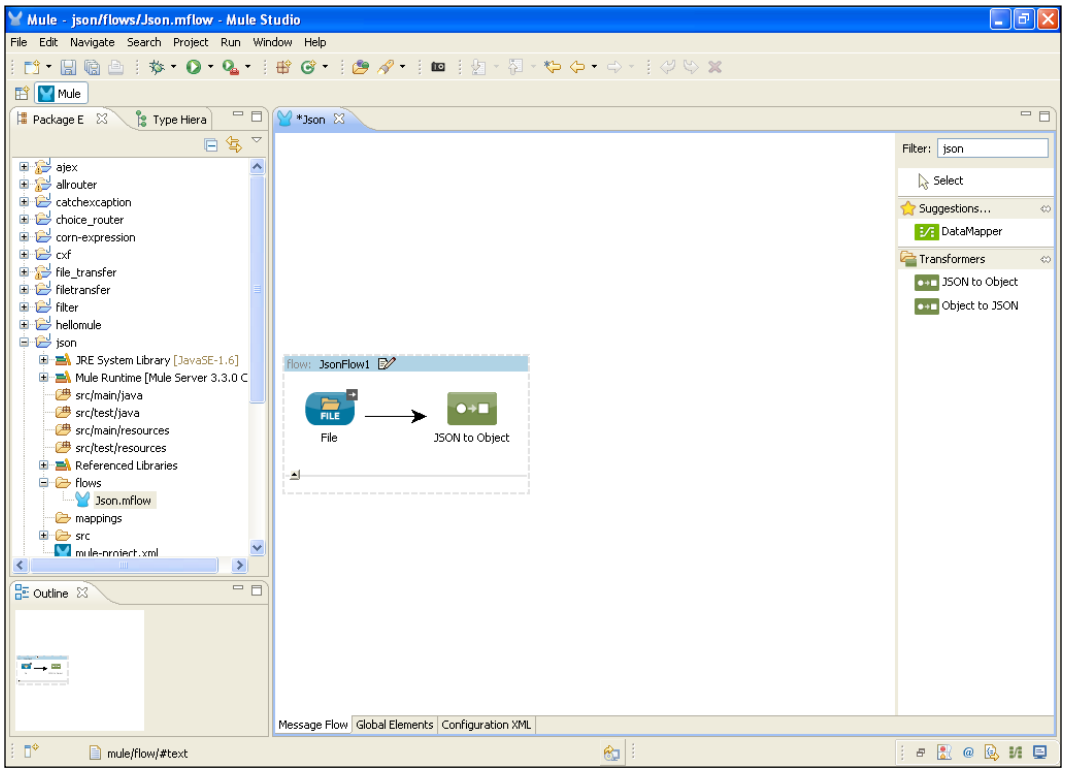
4. Click on the **Create** button again and go to **Connectors** | **Database**. Go to the **Queries** tab and enter the query key `InsertRecord` and the query `INSERT INTO tblemployee(empid,empname,empage,empdesignation) VALUES(#[message.payload.empid],#[message.payload.empname],#[message.payload.empage],#[message.payload.empdesignation])`. After that, go to the **General** tab, change the database name, and select the data-specific name in that box. Click on the **OK** button.



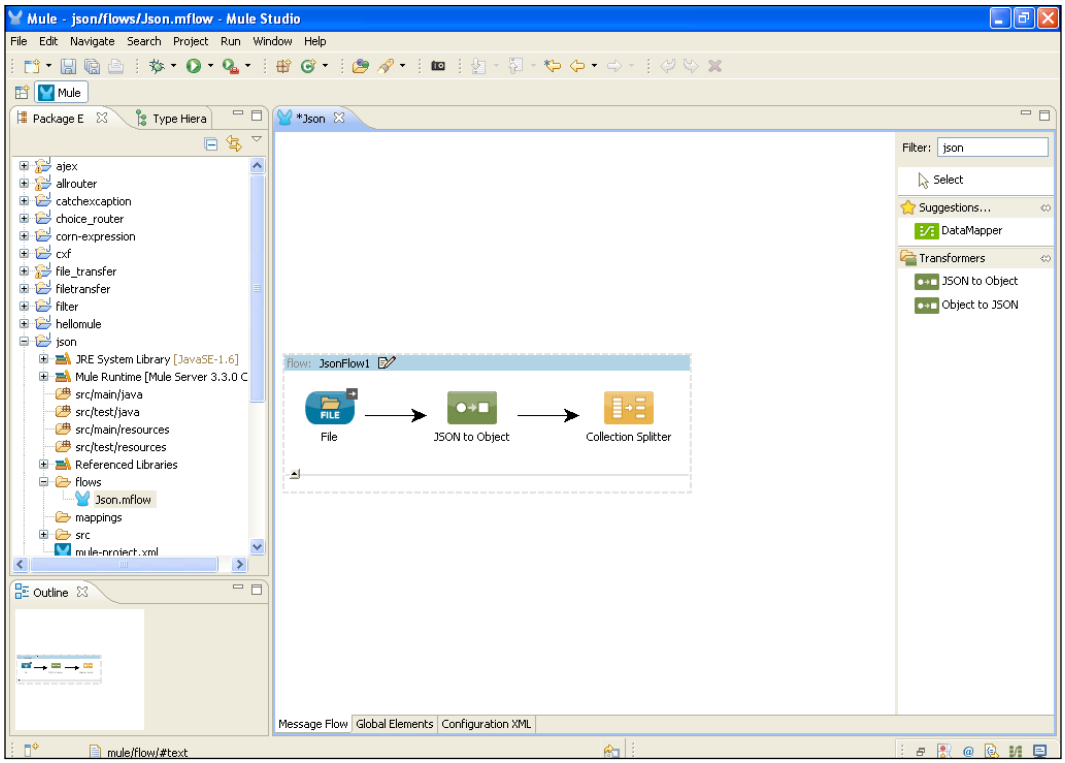
5. Go to **Message Flow**. Drag the **File** Endpoint onto the canvas and then double-click on it. Here, you will have to enter the path of the JSON file.



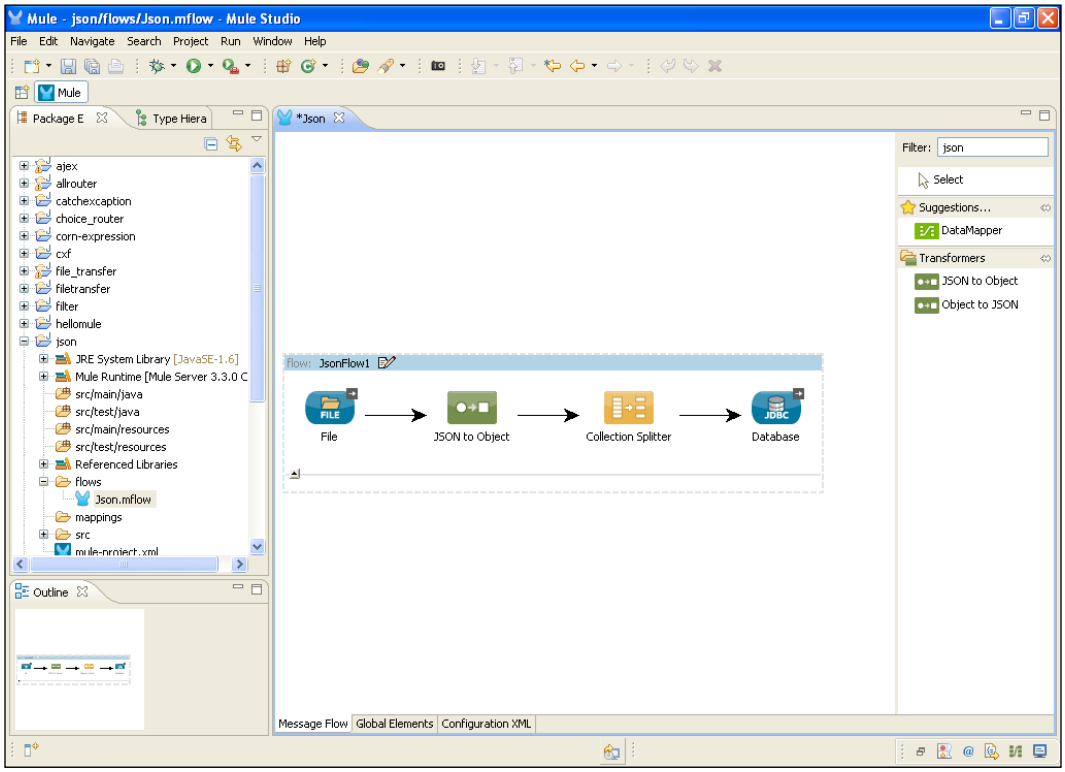
6. To convert the JSON format to object format, drag the **JSON to Object** transformer.



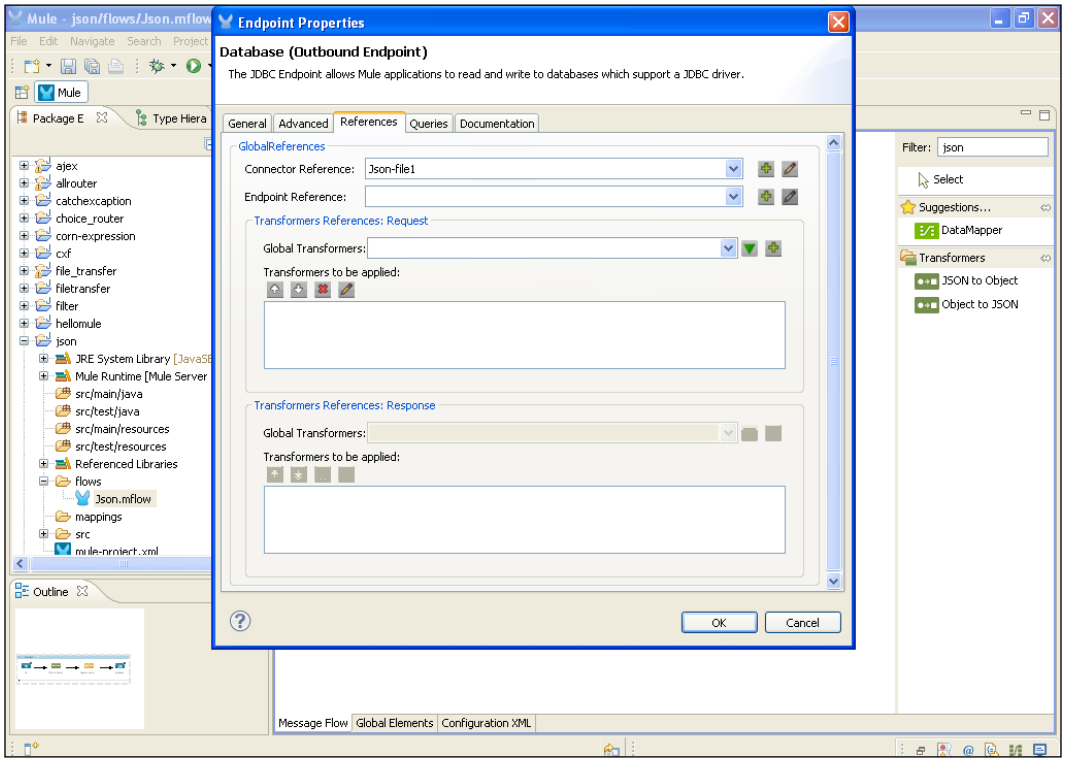
7. Drag the **Collection Splitter** onto the canvas, which will transform the JSON data into `Java.Util.List`, and then divide it into several `Java.Util.Maps` items. Finally, we will write it to the database.



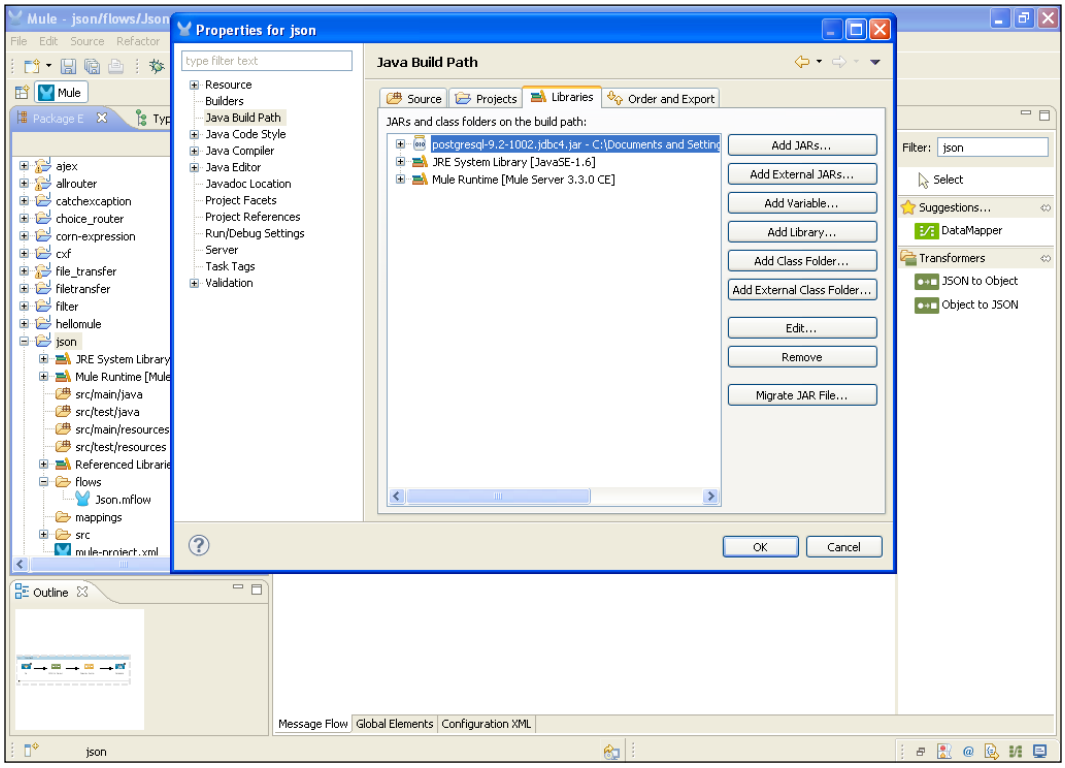
8. To configure the database, drag the **Database** Endpoint onto the canvas. We can now configure the **Database** Endpoint.



9. For configuring the **Database** Endpoint, click on the **References** tab and select the connector reference name. Select `JSON-file1` and then click on the **General** tab and select the query key that was created in the database connector.



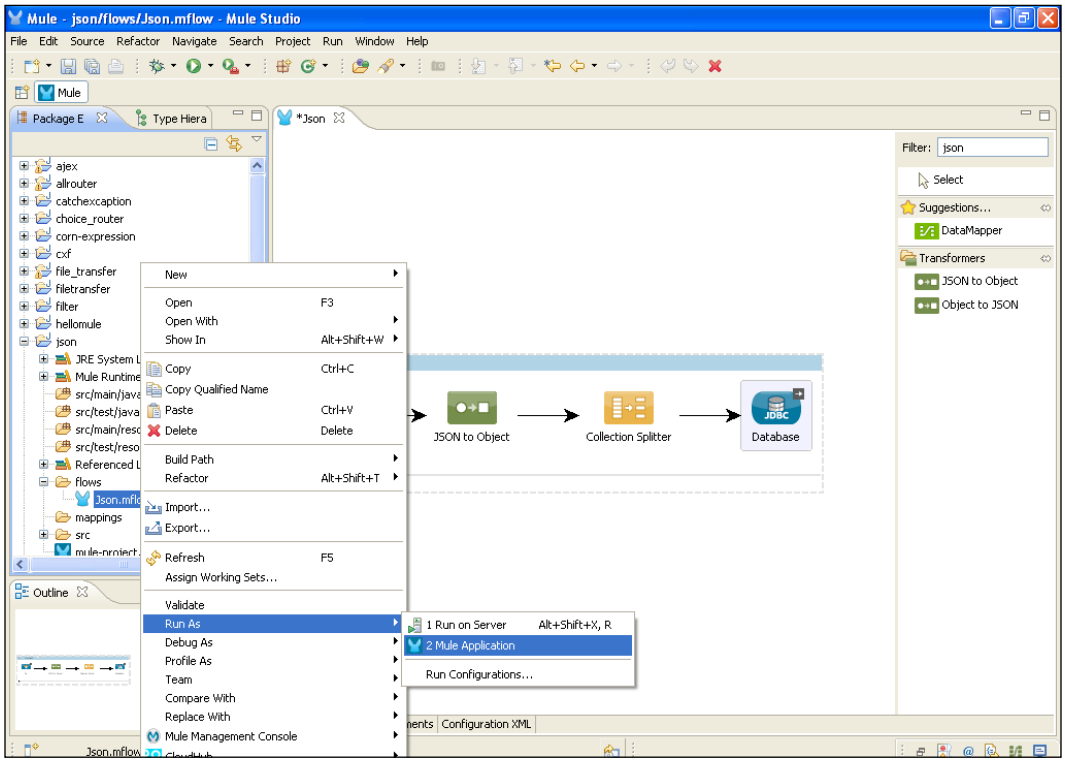
10. To import a JAR file for the PostgreSQL database, right-click on your project and click on **Properties**. Here, you can import the `jdbc` connector JAR file to PostgreSQL.



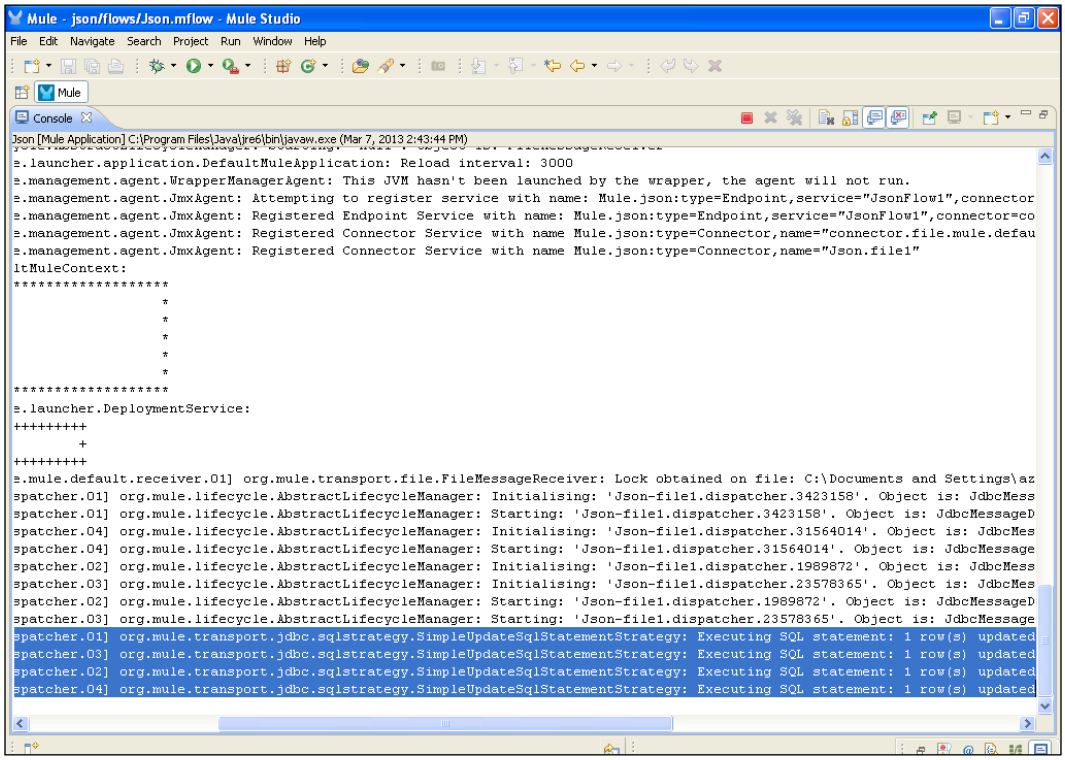
How it works...

In this section, you will learn how to deploy your application using Mule Studio. After deploying the application, here's how to run this example:

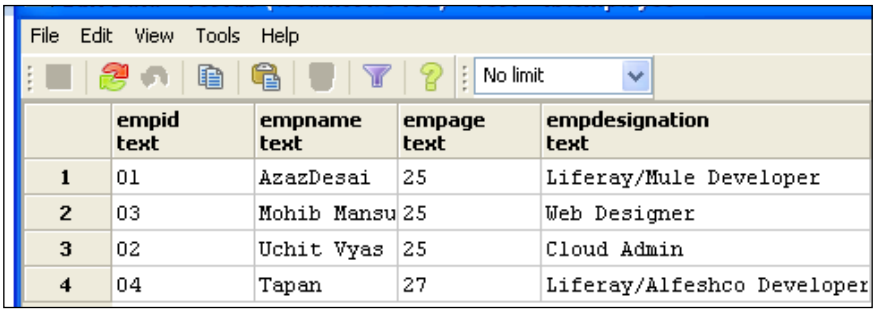
1. To deploy the application code in the Mule server, go to **Run As** | **Mule Application**. The Mule server will deploy your application.



2. In the console output window, you can see that four rows are updated in the database, as seen in the following screenshot:



3. Open the PostgreSQL database and have a look at the `tblemployee` table. You will see that all the data has been inserted into this table:



	empid text	empname text	empage text	empdesignation text
1	01	AzazDesai	25	Liferay/Mule Developer
2	03	Mohib Mansu	25	Web Designer
3	02	Uchit Vyas	25	Cloud Admin
4	04	Tapan	27	Liferay/Alfeshco Developer

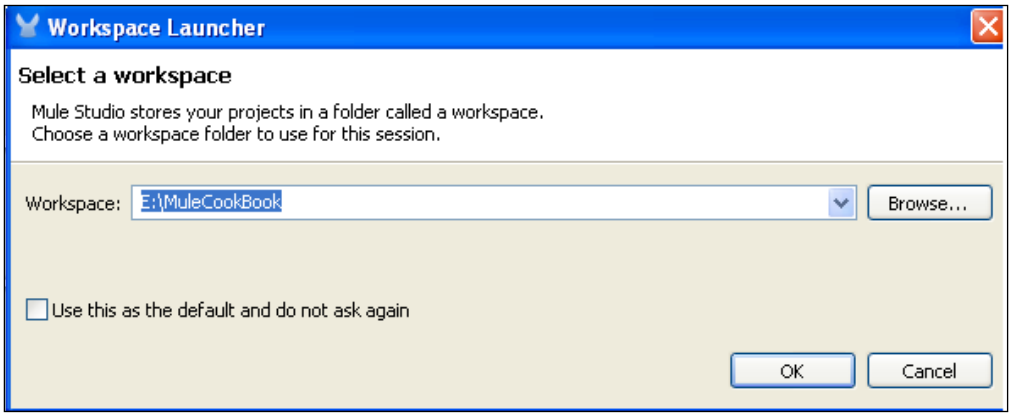
Configuring the Object-to-XML transformer

The **Object-to-XML** transformer is used to transform a Java object to XML data. In this example, you will see how to retrieve data from the database and how to store data in a particular XML file through the Object-to-XML transformer. Here, you will use the table from the previous example. We will use three components: the Database Endpoint, the Object-to-XML transformer, and the File Outbound Endpoint.

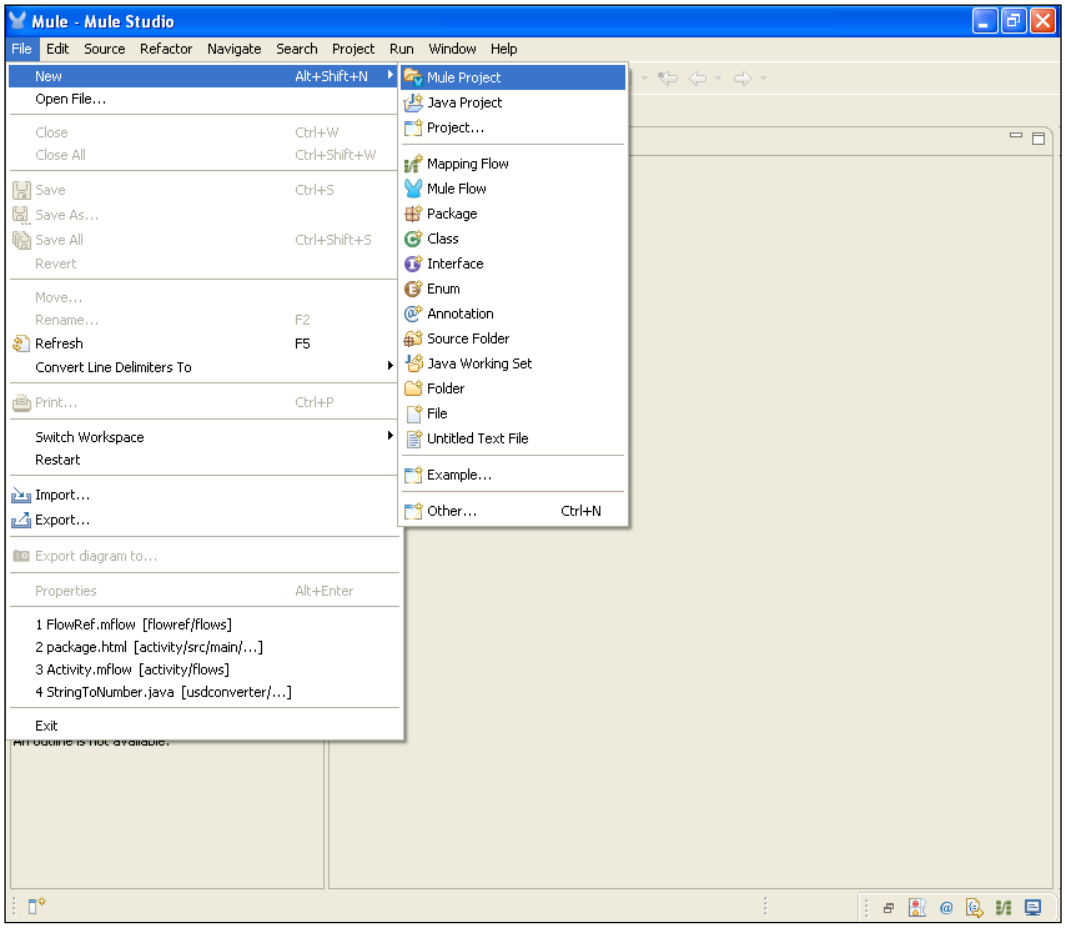
Getting ready

In this section, you will see how to configure the Database Endpoint in Mule Studio.

1. Open Mule Studio and enter a name for the workspace as shown in the following screenshot:



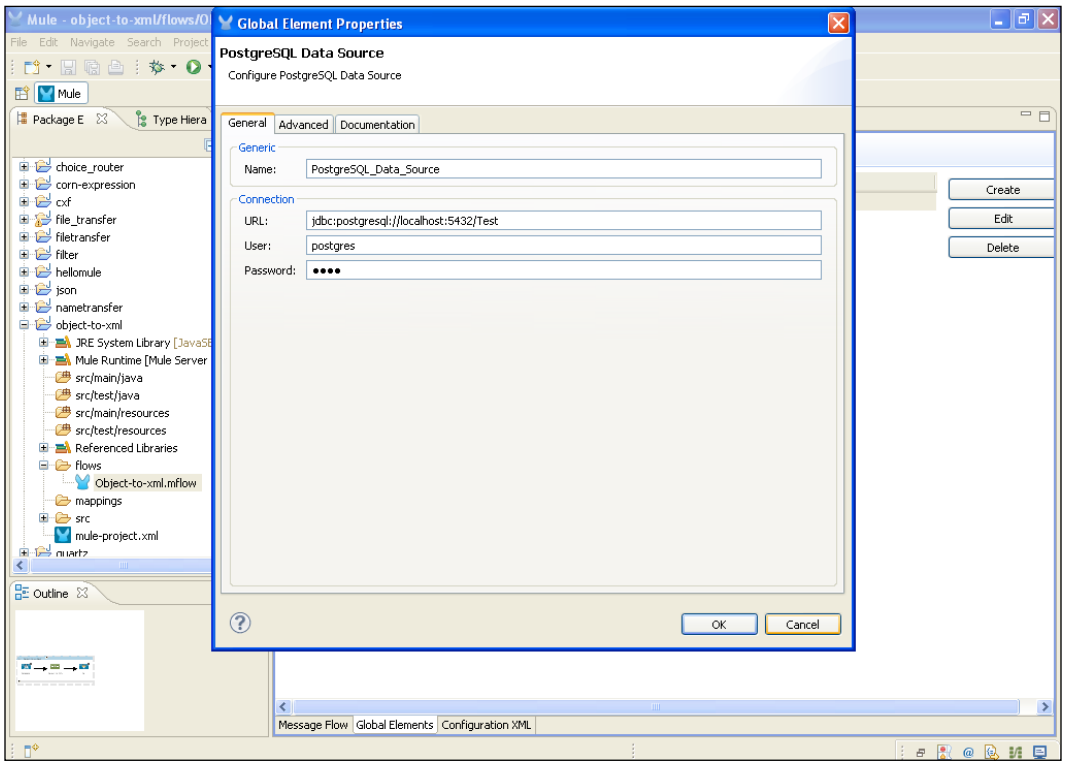
2. To create a new project, go to **File** | **New** | **Mule Project**. Enter the project name as `Object-to-xml` and click on **Next** and then on **Finish**. Your new project is created. You can now start implementing it.



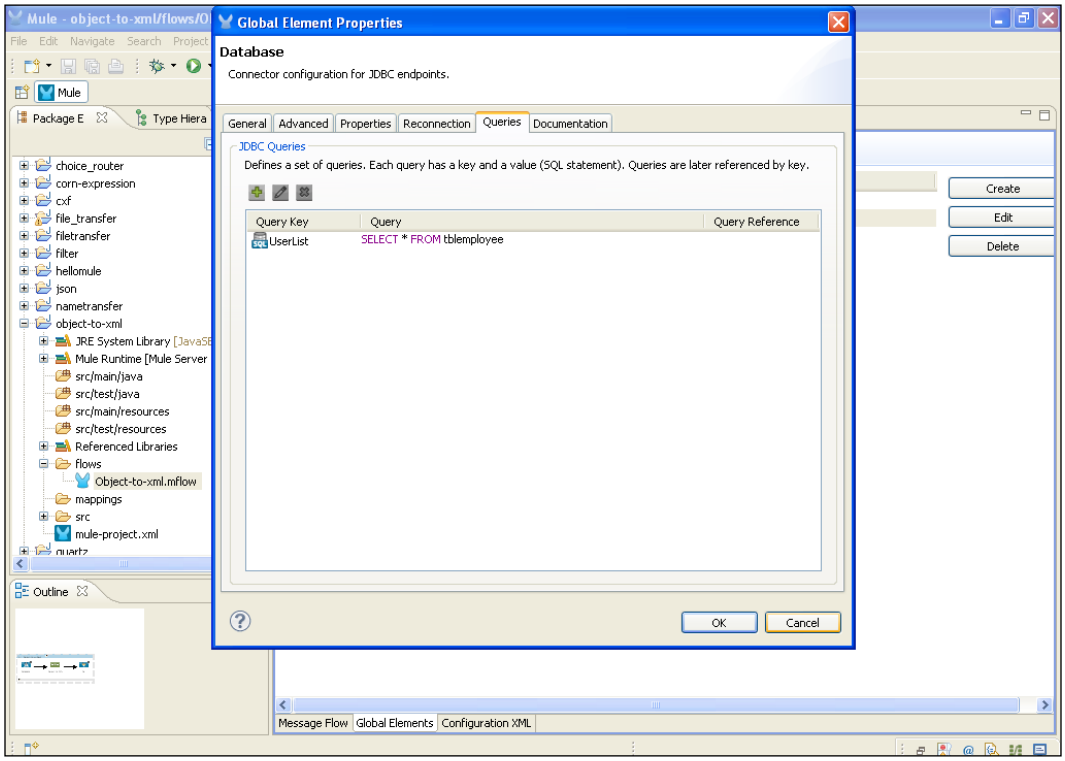
How to do it...

In this section, you will learn how to configure the Object-to-XML transformer and the File Outbound Endpoint.

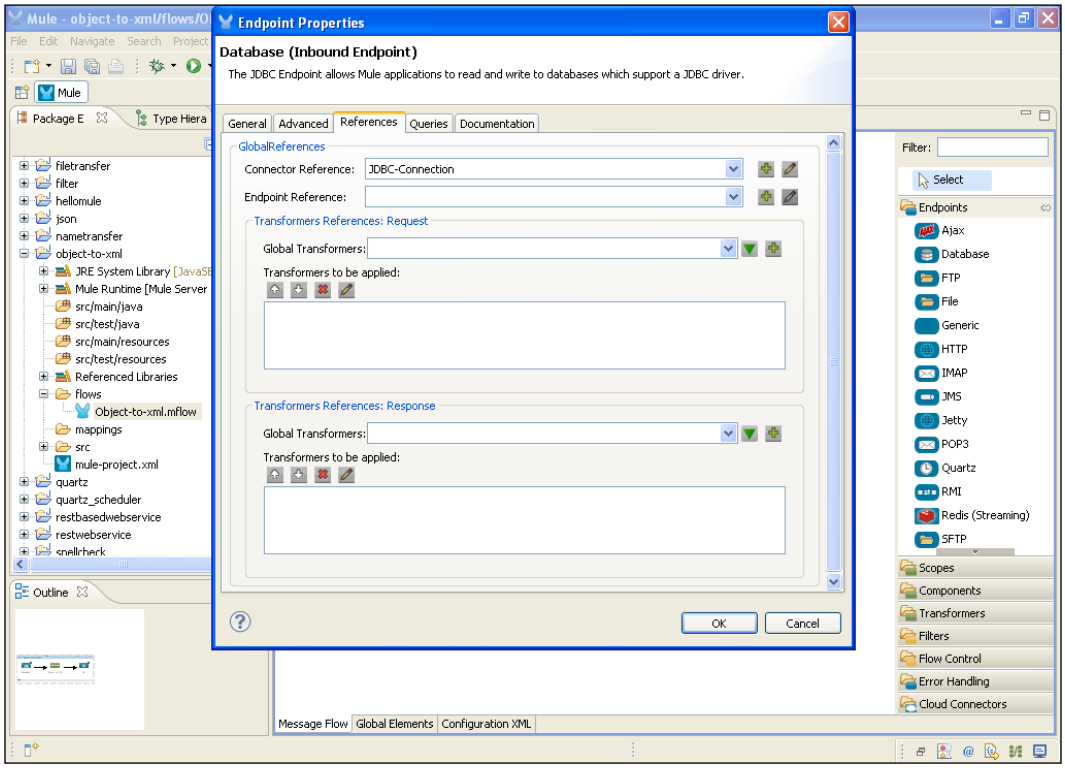
1. Navigate to the `Object-to-xml.mflow` file. To configure the database, go to the **Global Elements** tab, click on the **Create** button, and go to **Data Sources** | **PostgreSQL Data Sources**. In the **URL:** textbox, enter the name of the database that was created in PostgreSQL. Here we enter `Test`. Enter the username and password for PostgreSQL and click on the **OK** button.



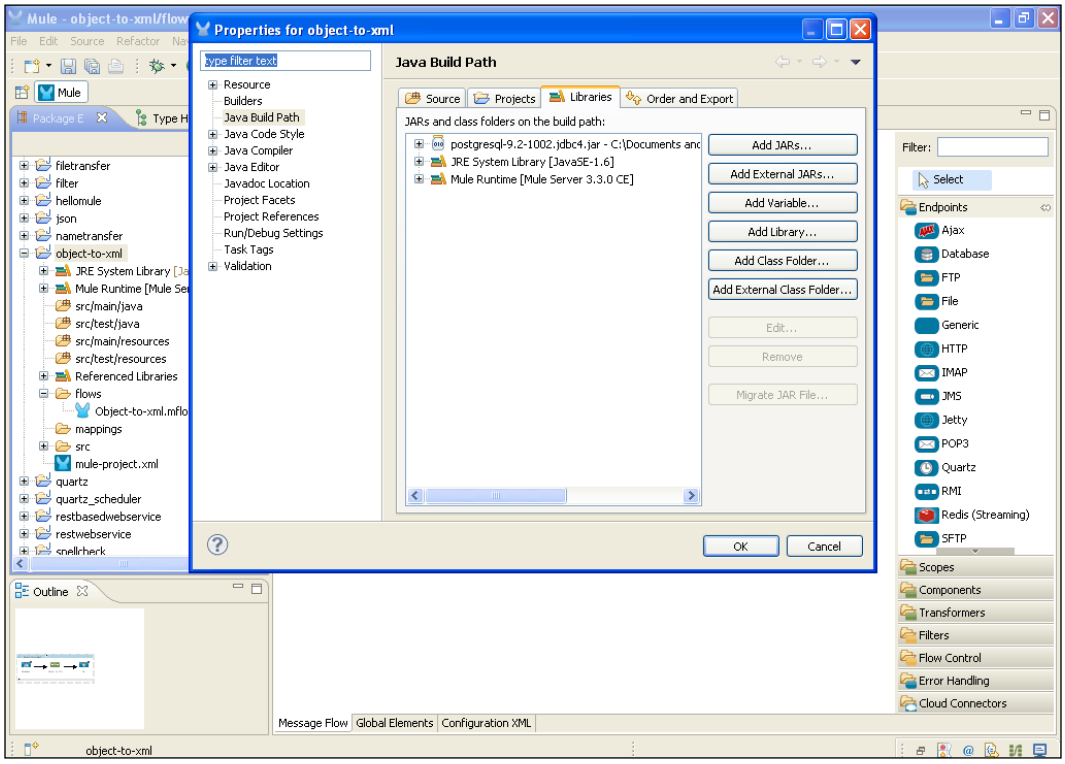
2. Click on the **Create** button and go to **Connectors** | **Database**. Go to the **Queries** tab, enter the query key name `Update List` and the query `SELECT * FROM tblemployee`. After this, go to the **General** tab, change the database name, and select a data-specific name in the box. Click on the **OK** button.



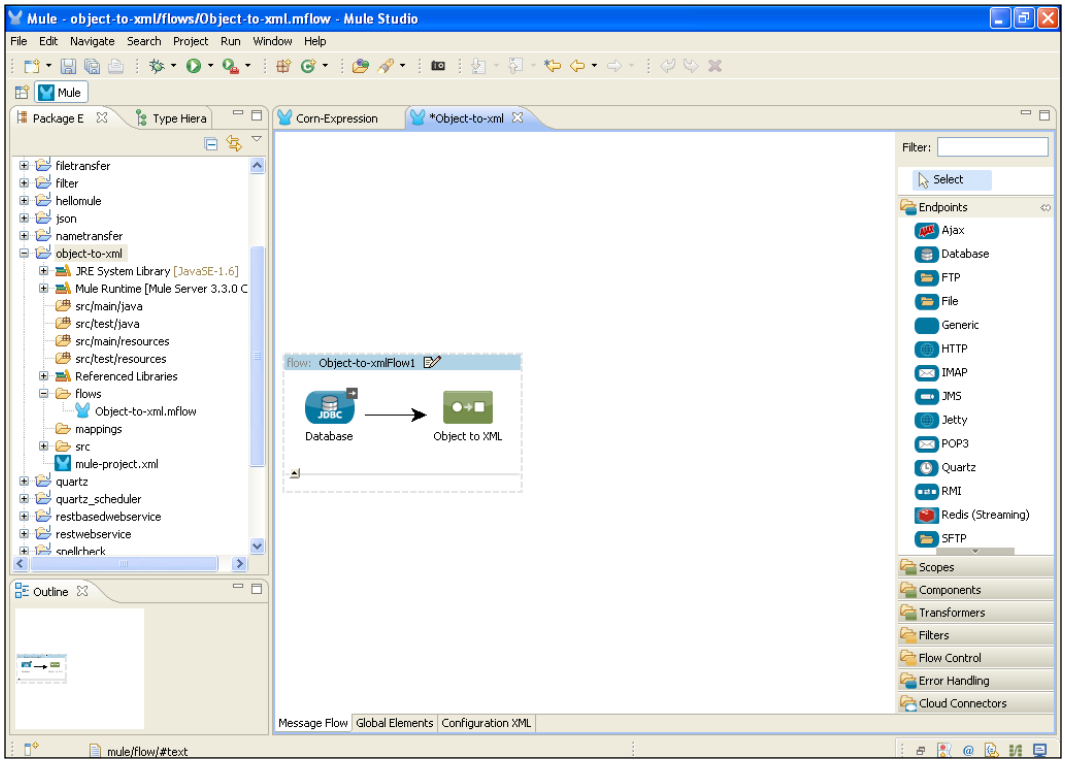
3. Go to the **Message Flow** tab, drag the **Database** Endpoint onto the canvas, and configure it. Click on the **References** tab and select the `connector-ref` name. Here, select the **JDBC** connector. Click on the **General** tab and select the query key that was created in the **Database** connector.



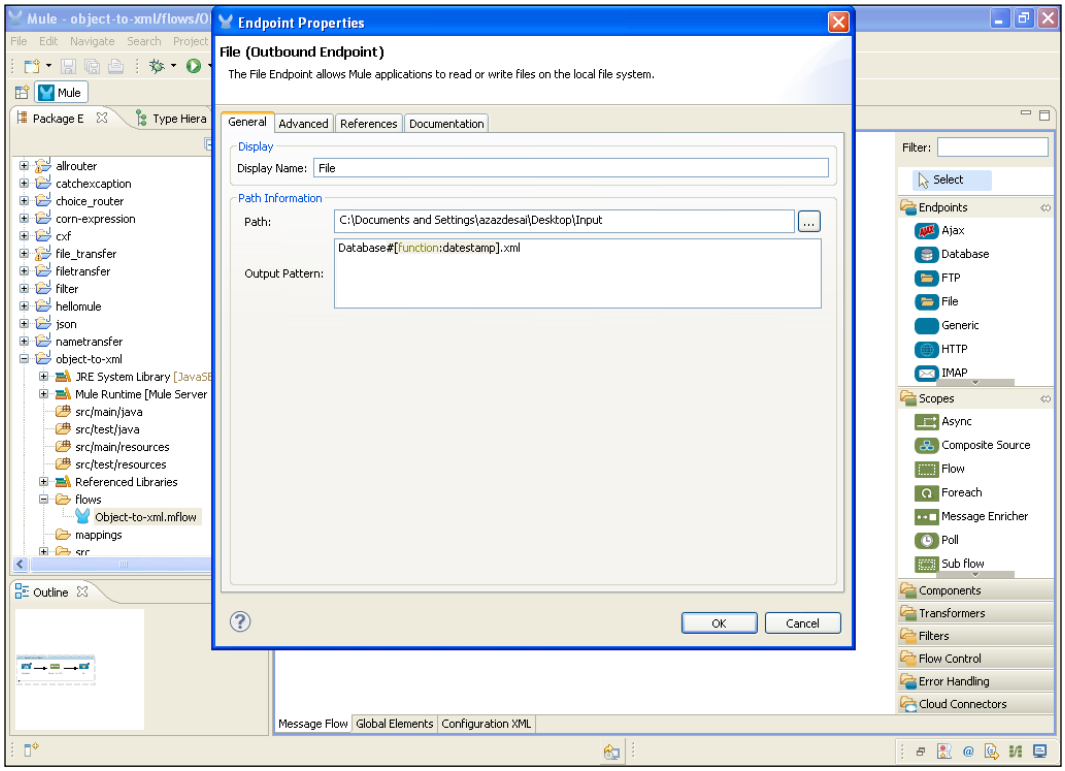
4. To import a JAR file for the PostgreSQL database, right-click on your project and go to **Properties**. Here, you have to import the PostgreSQL `jdbc` connector JAR file.



5. To convert an object to XML format, drag the **Object to XML** transformer onto the canvas. The transformer will create a map with the result of the query and send the map in a message. You can now use this map and convert it to XML.



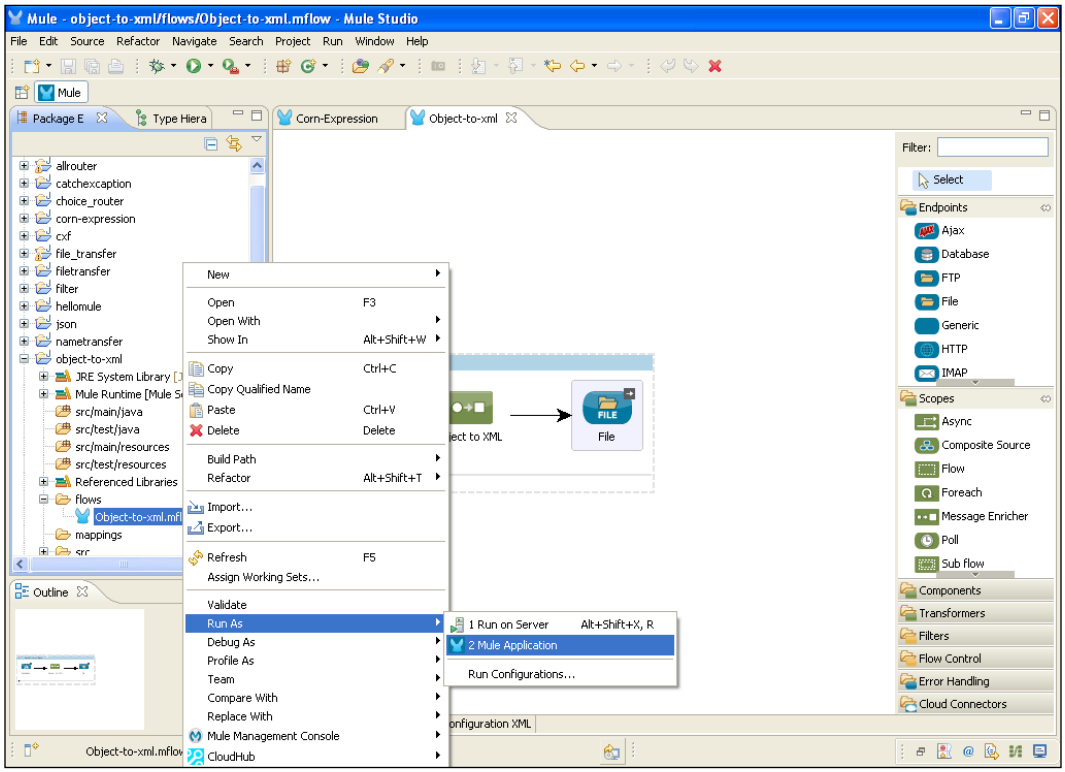
6. To store data in a file, drag the **File** Endpoint onto the canvas and configure it. Double-click on the **File** Endpoint. Firstly, select the storage file path in the following output pattern: `Database#[function:datestamp].xml` (`Database` represents a filename and `datestamp` represents the current date and time).



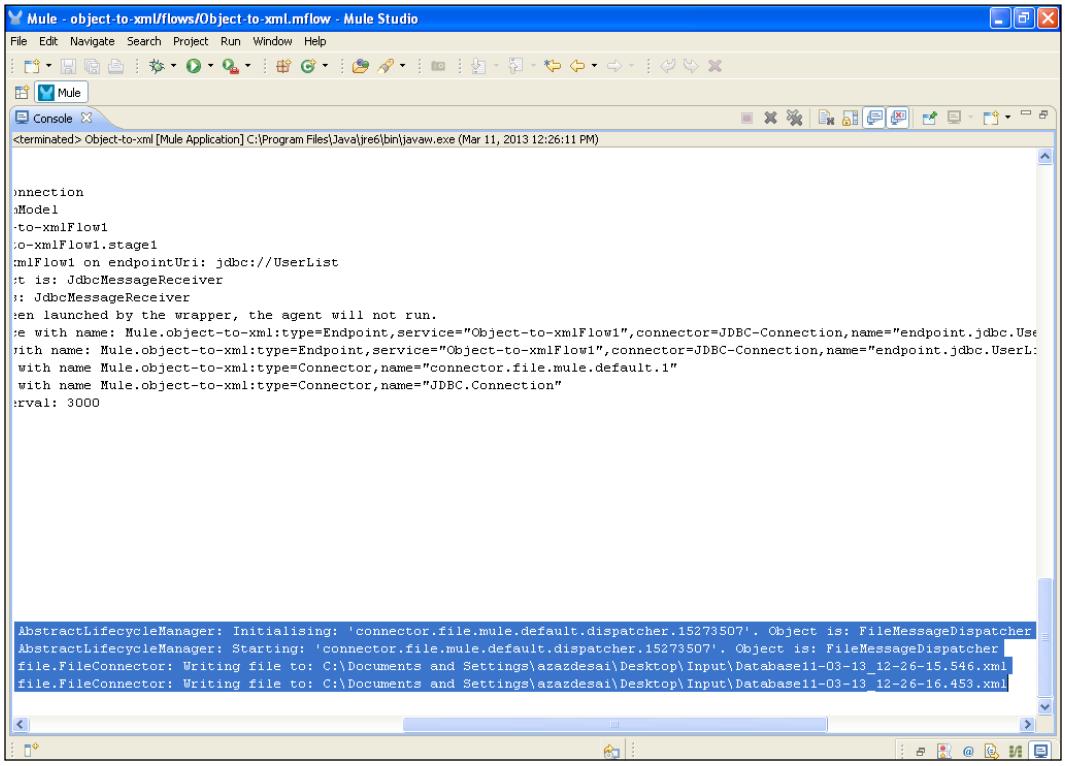
How it works...

In this section, you will learn how to deploy the application in Mule Studio and how to run the application in the browser after deploying it.

1. To deploy the application code in the Mule server, go to **Run As** | **Mule Application**; the Mule server will deploy your application.



2. You can see in the console output that the file has been created in the mentioned path with the name and current date/time, as seen in the following screenshot:

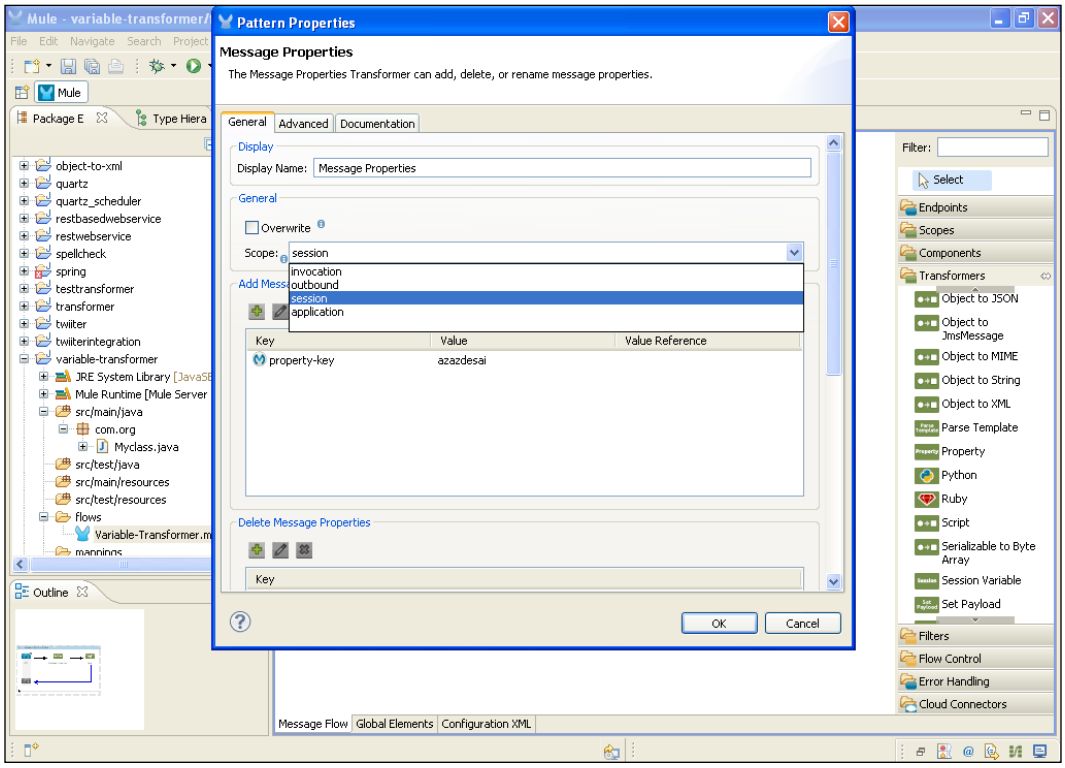


Configuring the Message and Variable transformers

Message properties and variables are the most frequently used features in Mule. Message properties have mainly four scopes: **invocation**, **outbound**, **session**, and **application**. A Mule message consists of three parts: header, payload, and attachments.

Getting ready

You can configure the Message Properties transformer as shown in the following screenshot. A Mule message supports two types of properties: **inbound properties** and **outbound properties**.



How to do it...

In this section you will learn how to add session properties into the Mule `config.xml` file and how to use the session scope in the Mule configuration file.

How to add session properties

Drag the properties of the transformer onto the canvas. To configure, double-click on the **Message Properties** transformer.

```
<message-properties-transformer scope="session">
  <add-message-property key="property-key" value="property-value"/>
</message-properties-transformer>
```

You can also trigger a Mule session using `MuleEventContext`.

```
public class MyClass implements Callable
{
  public Object onCall (MuleEventContext eventContext) throws
Exception
  {
    eventContext.getSession().setProperty("property-key","some
value");
    return eventContext.getMessage();
  }
}
```

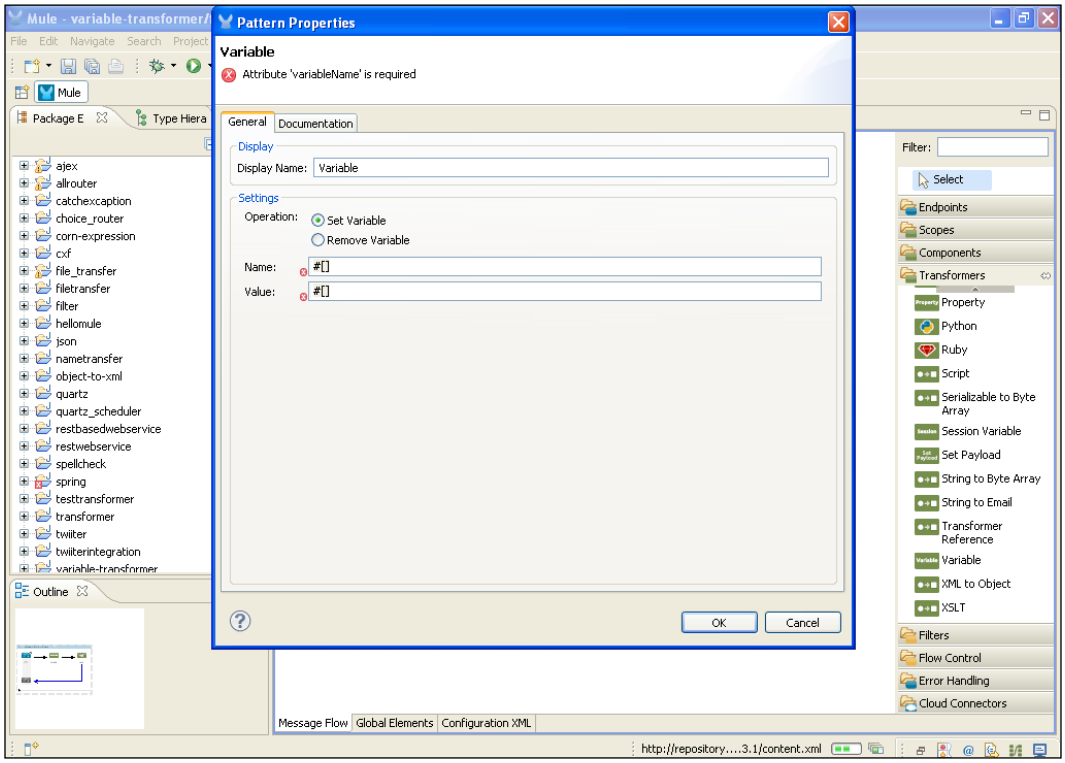
A Mule session is maintained while the message goes through Mule instances, so don't store things you don't need, to avoid wasting memory.

How it works...

In this section, you will learn what a **Variable** transformer is and how to use it in Mule Studio. You will see how to set variables in Mule Studio.

The Variable transformer

The Variable transformer allows activation of variables. To activate the variable, first drag the **Variable** transformer onto the canvas. Double-click on the **Variable** transformer. You can declare a variable as an expression or as a literal. If you declare the variable as an expression, Mule evaluates the variable against the content of the current message.



In the **Name:** textbox, specify the variable name as a string. In the **Value:** textbox, type a string that specifies the variable value, either in expression or literal, depending on your choice.

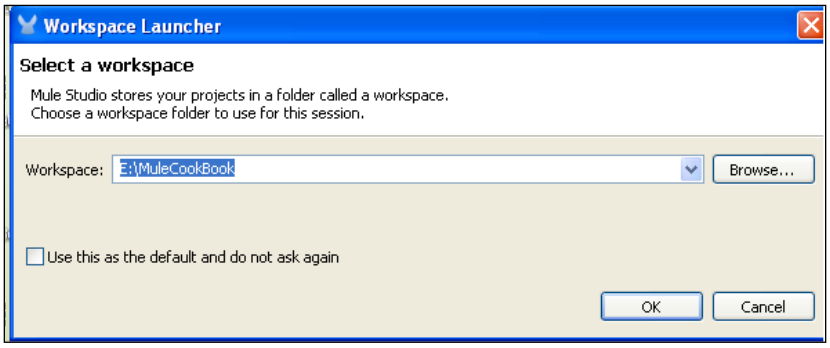
Creating the custom transformer

A **custom transformer** is a user-defined transformer class that implements the `org.mule.api.transformer` transformer. Your class can extend `AbstractTransformer` or `AbstractMessageAwareTransformer`, depending on your requirements.

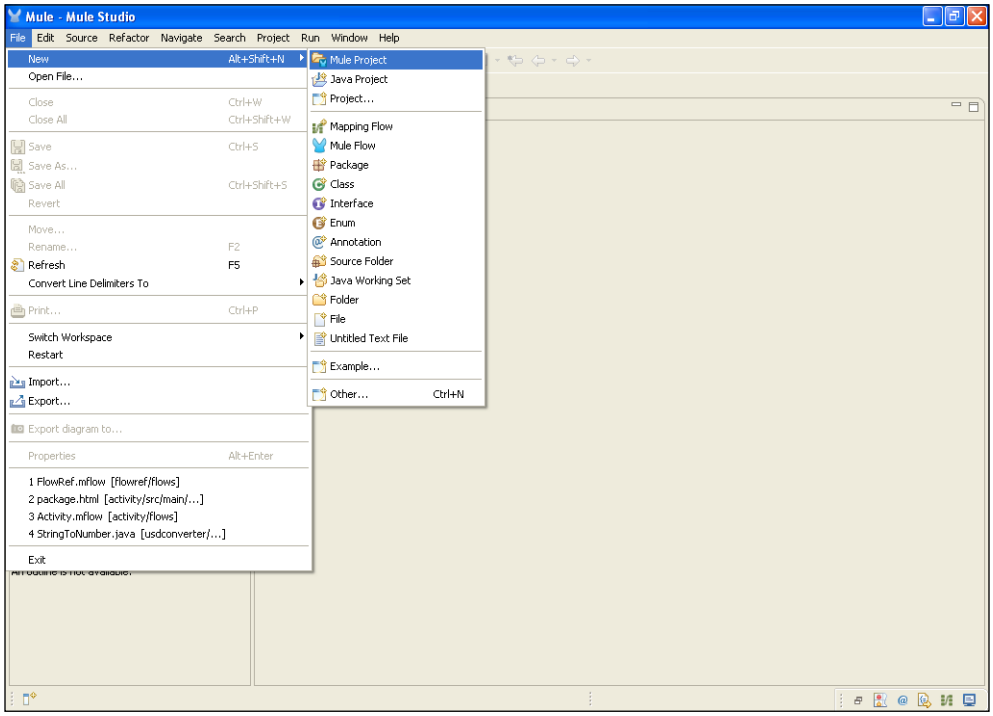
Getting ready

In this section, you will learn what a custom transformer is and how to configure it.

1. Open Mule Studio and enter a name for the workspace as shown in the following screenshot:



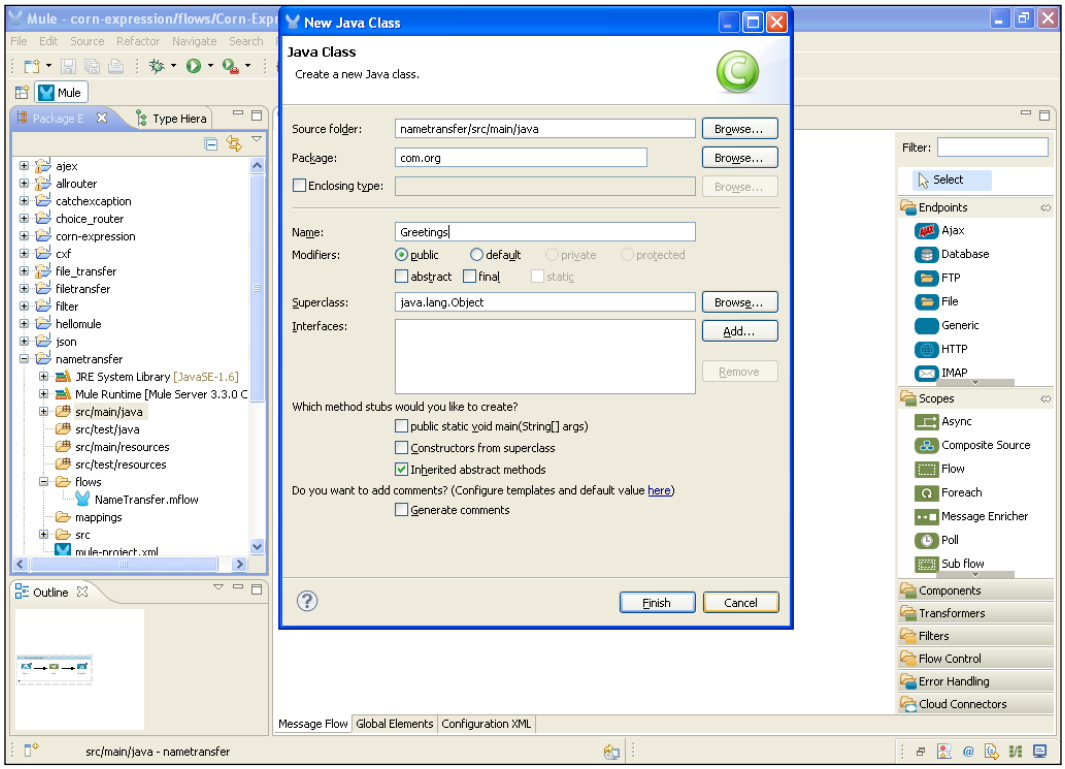
2. To create a new project, go to **File** | **New** | **Mule Project**. Enter the project name as `NameTransfer` and click on **Next** and then on **Finish**. Your new project is created. You can now start the implementation.



How to do it...

In this section, you will use four components: the HTTP Endpoint, the custom transformer, the Java component, and the Append String transformer. Also, you will learn how to create a custom transformer and a custom Java component in Mule Studio.

1. Right-click on `src/main/java` and go to **New** | **Class**. Enter the package name as `com.org` and the class name as `Greeting`; click on the **Finish** button.



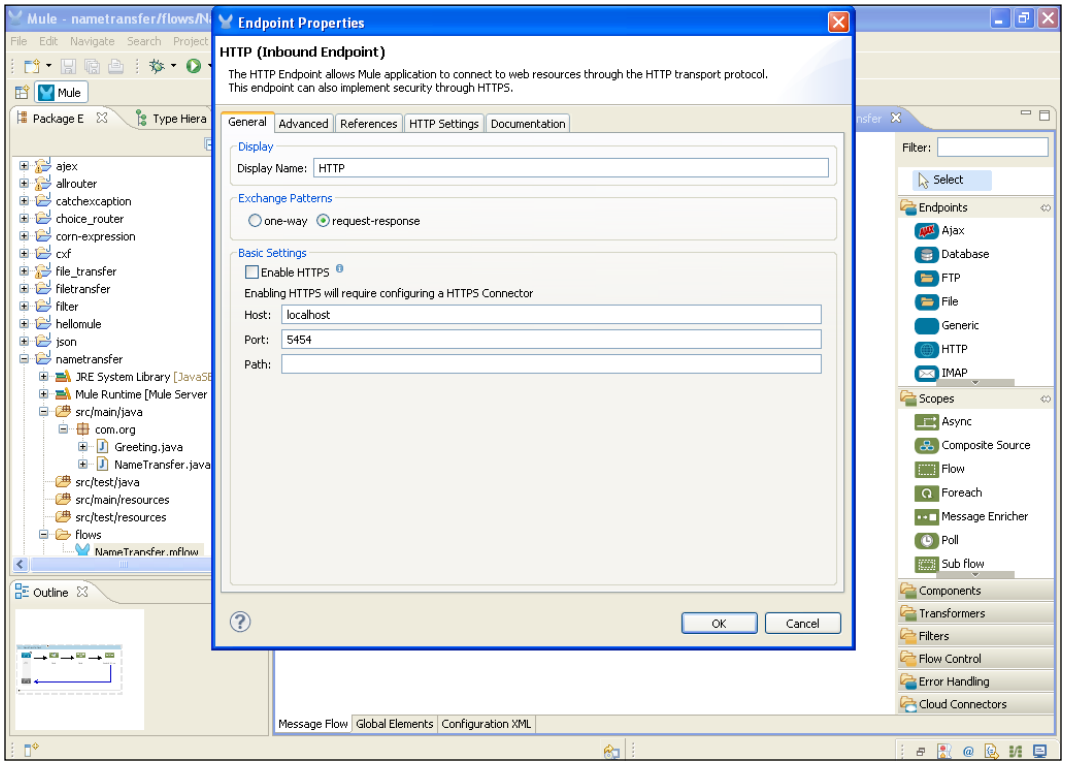
2. Here, you can create a simple `sayHi` method; the return type is set to `String`.

```
package com.org;
public class Greeting {
public String sayHi(String str)
  {
    return "Hello "+str;
  }
}
```

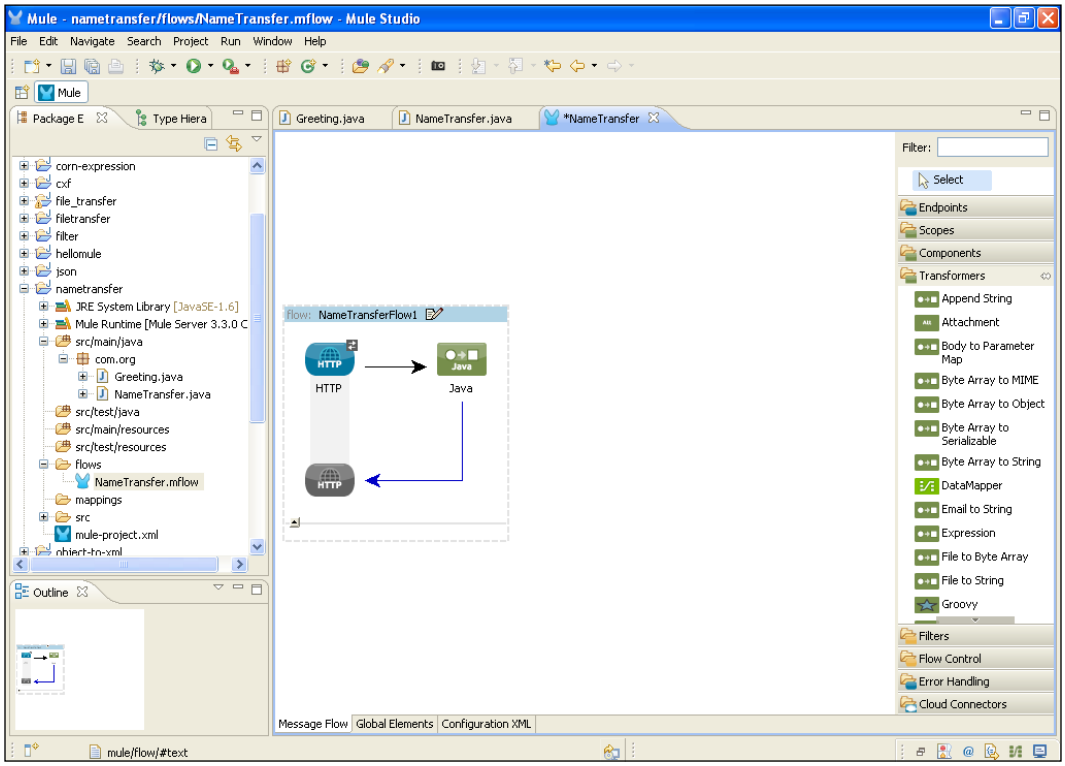
3. In the same package, you had created another class called `NameTransfer`. This class is called custom transfer, and through this class you will remove the extra forward slash. Here you can extend `AbstractTransformer`.

```
package com.org;
import org.mule.api.transformer.TransformerException;
import org.mule.transformer.AbstractTransformer;
public class NameTransfer extends AbstractTransformer{

@Override
  protected Object doTransform(Object src, String enc)
      throws TransformerException {
    if(src instanceof String)
    {
      String name=((String)src);
      {
    if(name.charAt(0)=='/')
       {
         name.substring(1);
       }
      }
    }
    return src;
  }
}
```

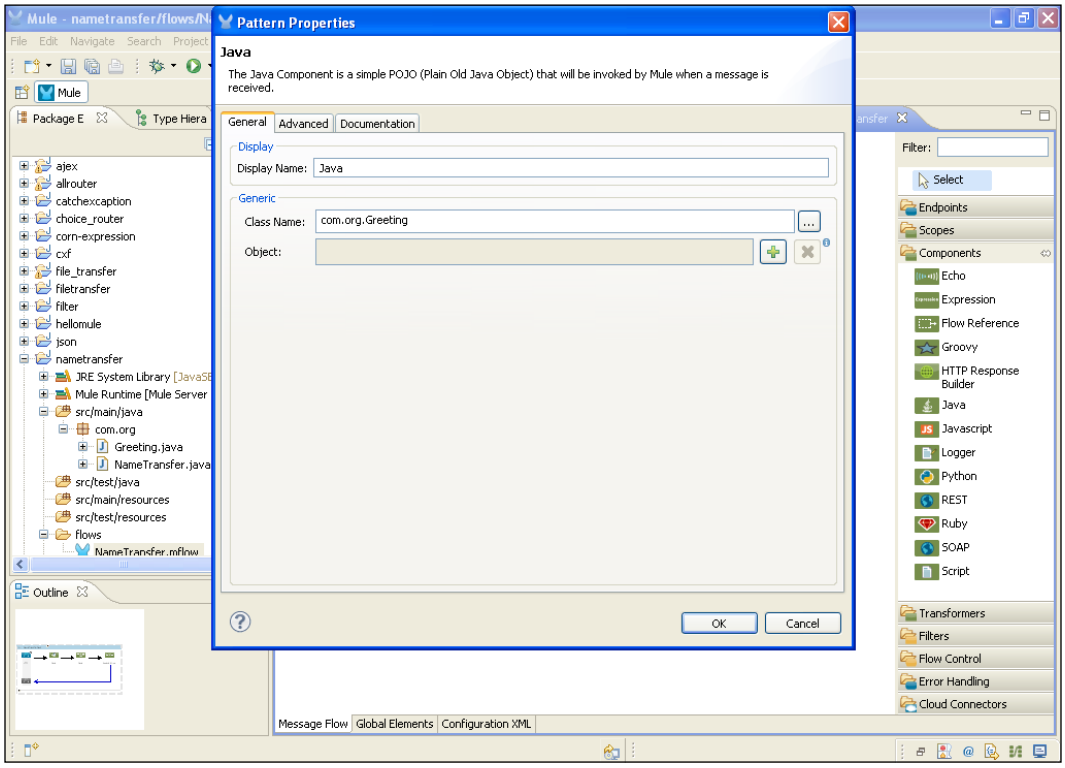
4. Go to the `NameTransfer.mflow` file. To configure the localhost URL, drag the **HTTP** Endpoint onto the canvas. Double-click on the **HTTP** Endpoint. Enter the port number and hostname and click on the **OK** button.



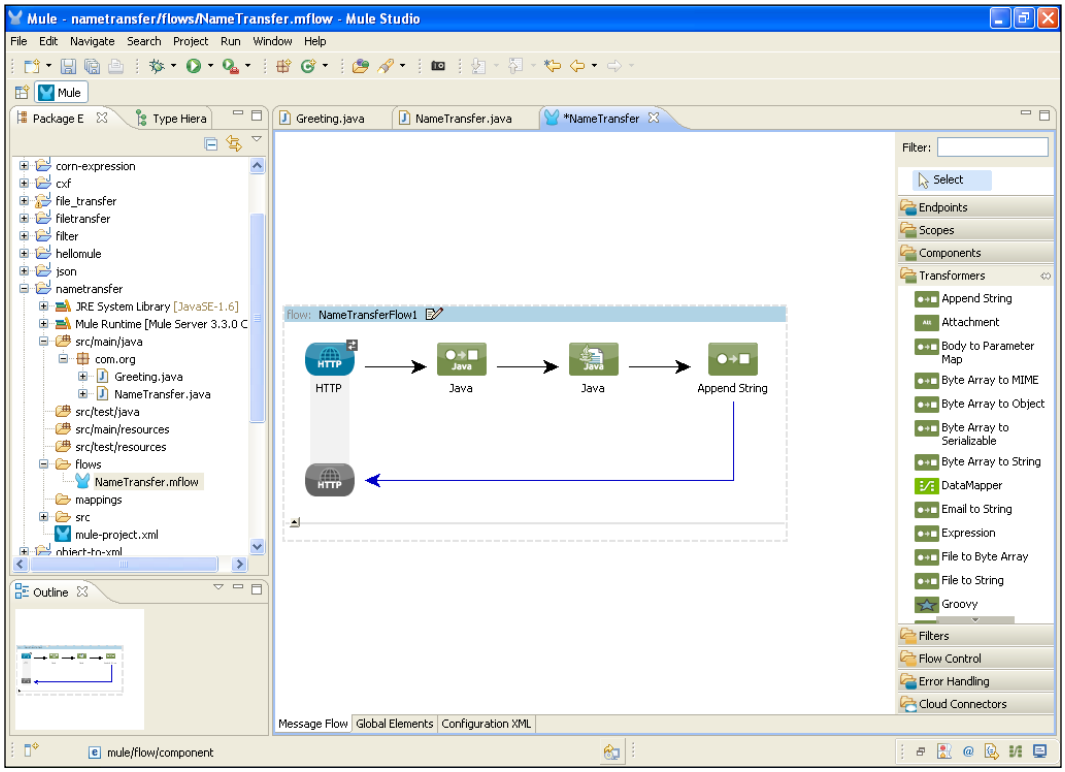
5. To configure the custom class, drag the **Java** transformer onto the canvas. Double-click on the **Java** transformer. Here, you need to import the `NameTransformer` class that was created earlier.



6. Drag the **Java** component onto the canvas. To configure it, double-click on the **Java** component. Here, you import the `Greeting` class that was created earlier.



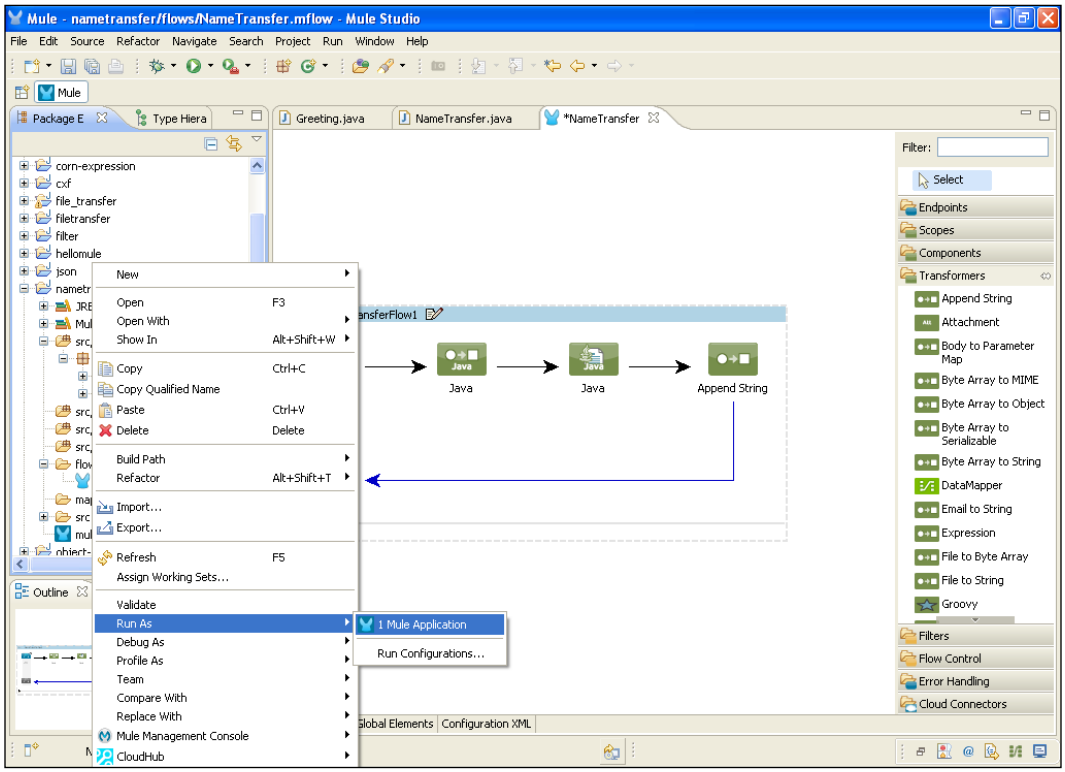
7. Drag the **Append String** transformer onto the canvas. To configure, double-click on it. In the **Message** textbox, enter the value `ESB` and click on the **OK** button.



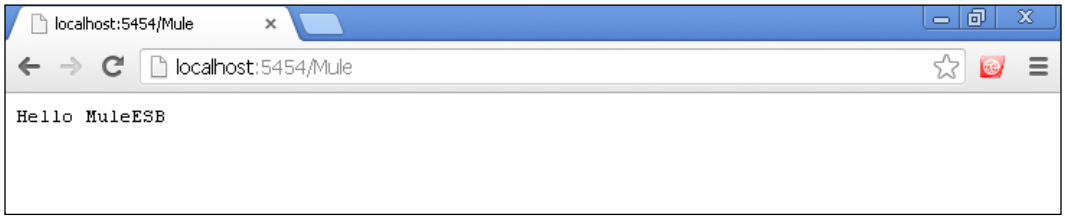
How it works...

In this section, you will learn how to deploy the application in Mule Studio and how to run this application in the browser after deployment.

1. To deploy the application code in the Mule server, go to **Run As** | **Mule Application**; the Mule server will deploy your application.



2. Copy the localhost URL `http://localhost:5454`. Open the browser, paste the URL, and type in `Mule`. You can see that the forward slash has been removed. The custom transformer removed the forward slash.



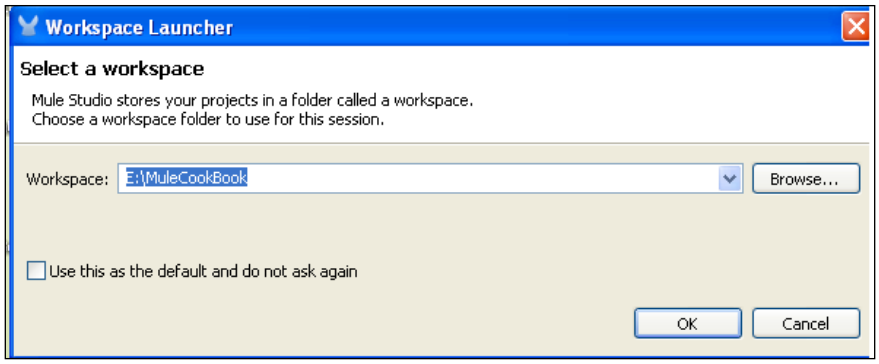
Understanding the DataMapper transformer

The **DataMapper** transformer is more powerful and flexible as compared to the rest of the transformers that are provided with Studio. Like other transformers, DataMapper can transform data across formats and manipulate the payload values as well. DataMapper can map an input field to a different output field.

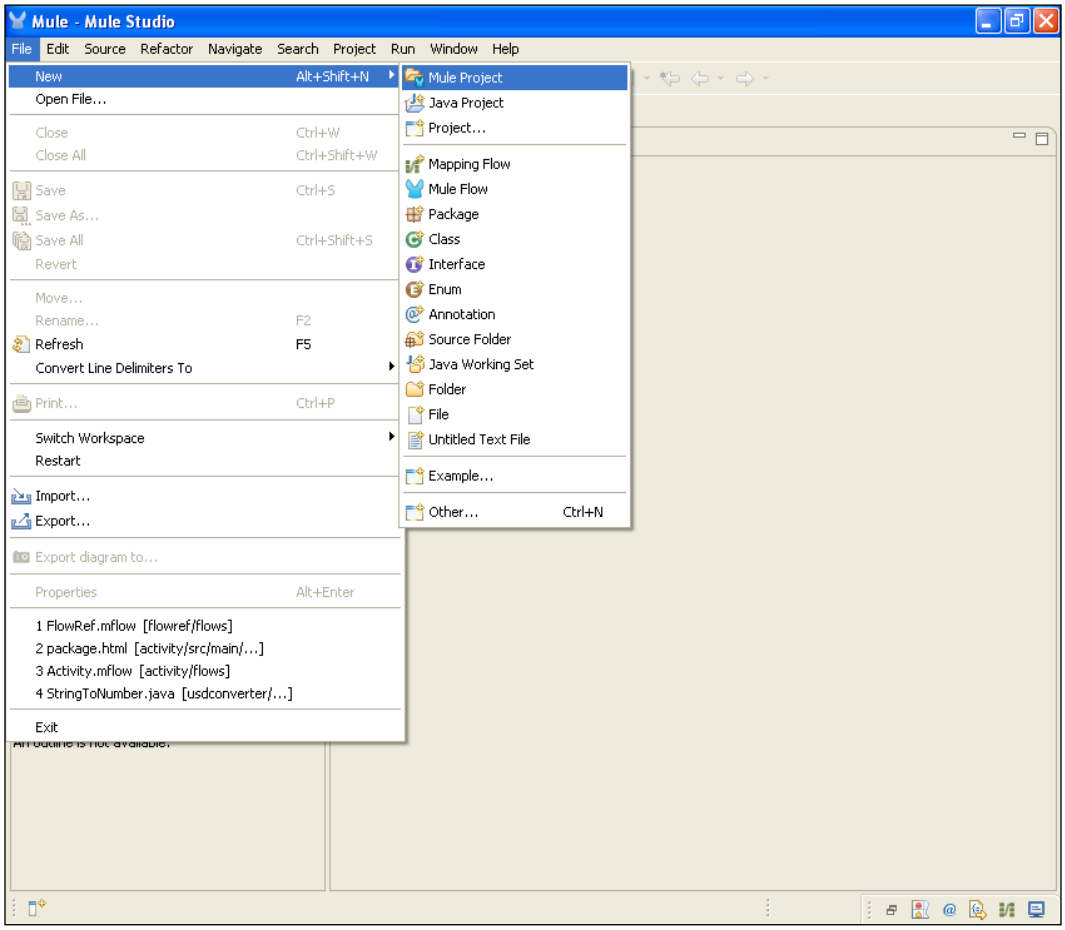
Getting ready

The DataMapper transformer works only in the Mule Studio Enterprise edition. DataMapper supports only six formats: CSV, XML, Java, Map, JSON, and Excel.

1. Open Mule Studio and enter a name for the workspace as shown in the following screenshot:



2. To create a new project, go to **File** | **New** | **Mule Project**. Enter the project name as `Mapper` and click on **Next** and then on **Finish**. Your new project is created. You will now have to start the implementation.

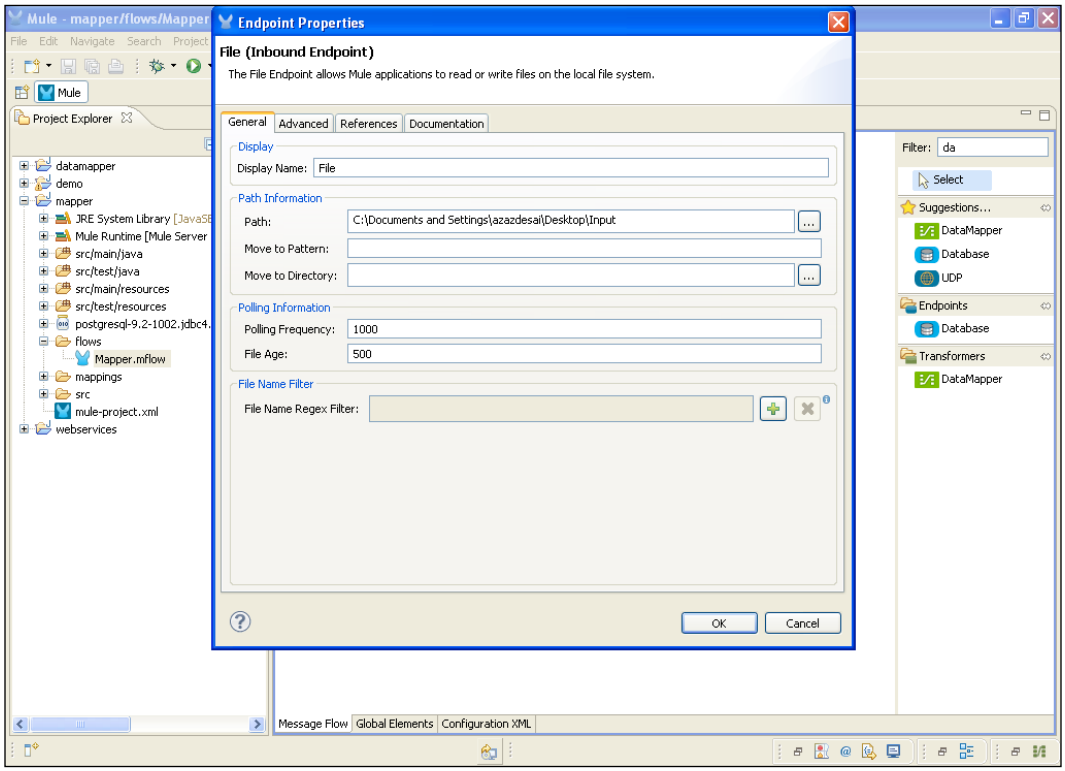


How to do it...

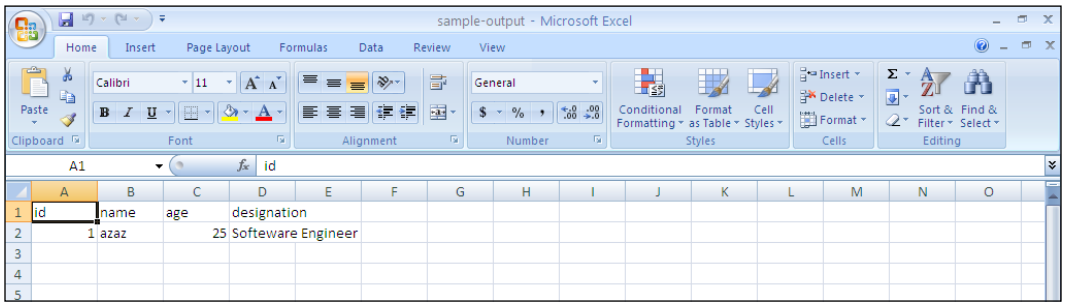
In this section, you will learn what is DataMapper, how to configure the DataMapper in Mule Studio, and when to use DataMapper in Mule Studio.

In this example, you will use three components: the File Inbound Endpoint, DataMapper, and Database.

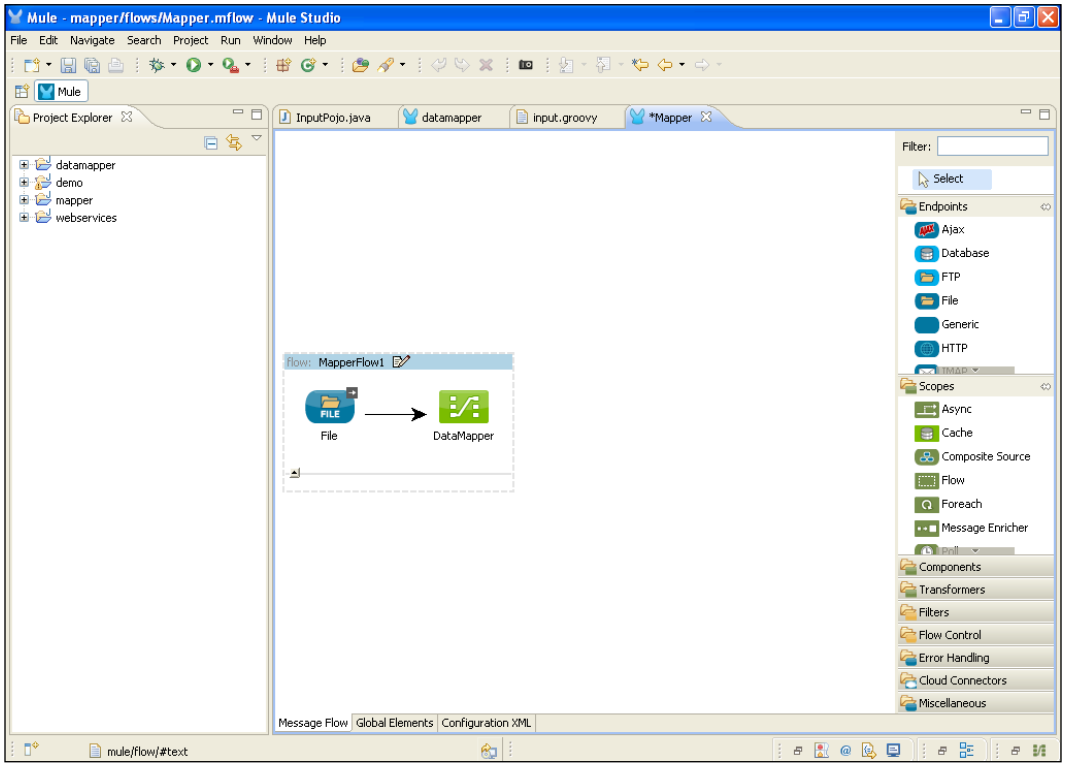
1. Go to the `Mapper.mflow` file and drag the **File** Inbound Endpoint onto the canvas. To configure, double-click on the **File** Endpoint. Here, you import the CSV file.



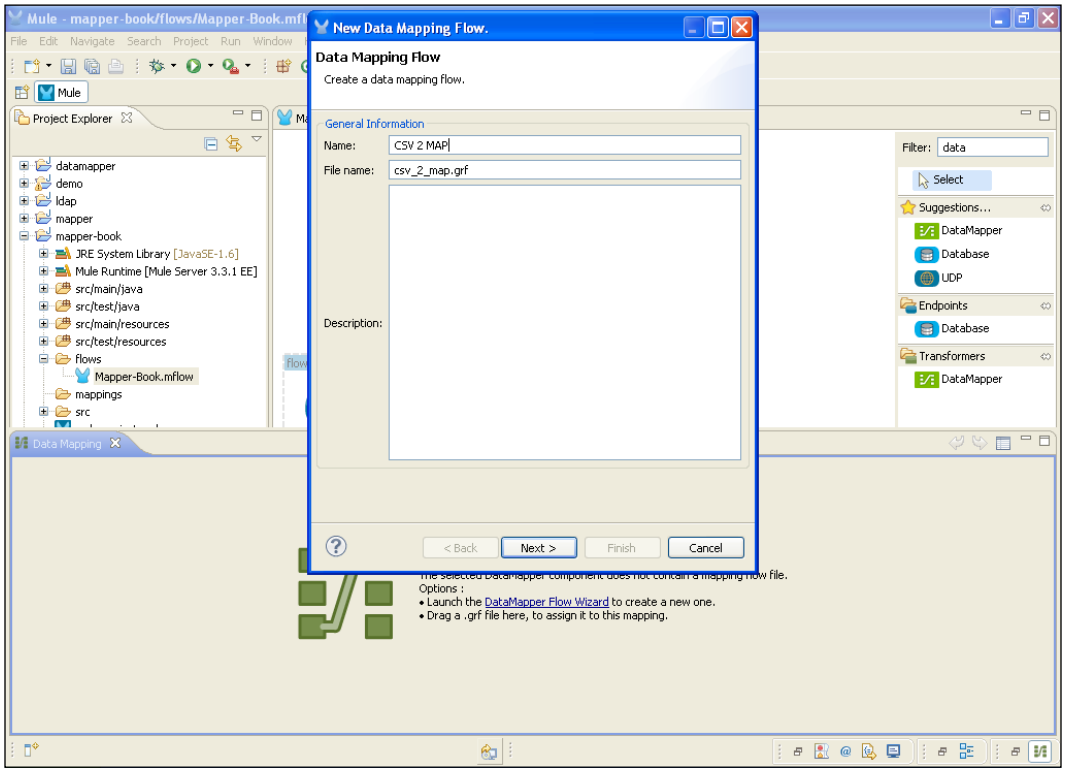
This is how the `sample-output.csv` file looks:



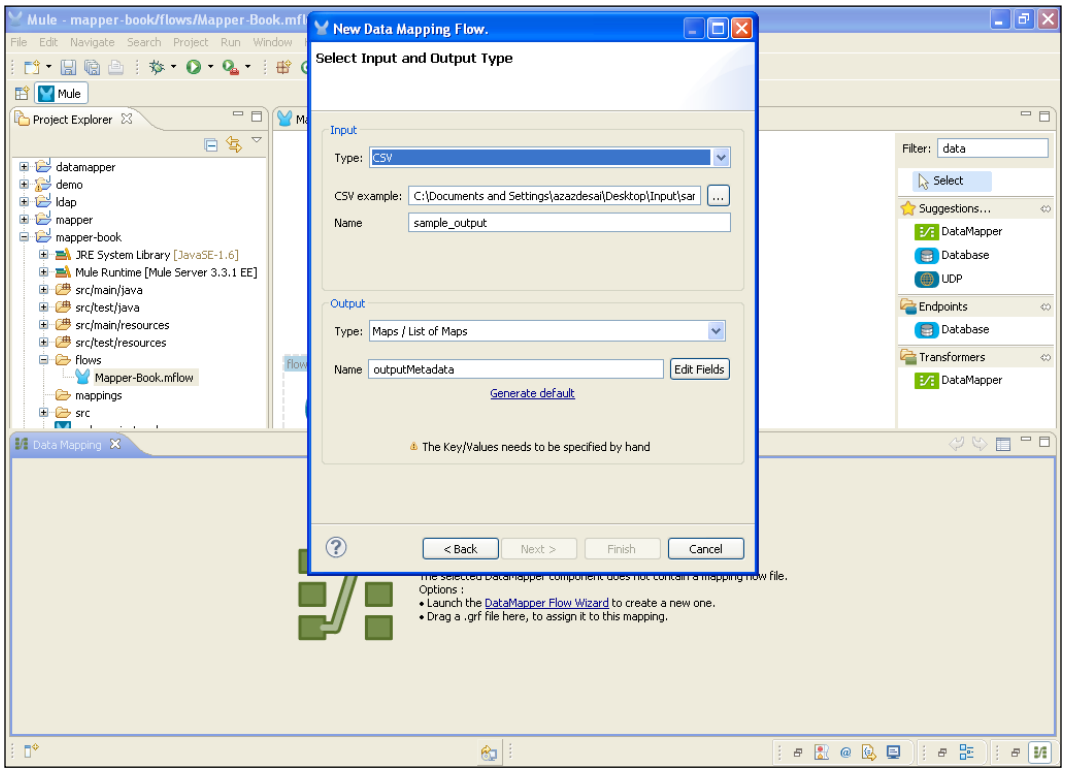
2. To configure mapping, drag the **DataMapper** transformer onto the canvas and double-click on it.



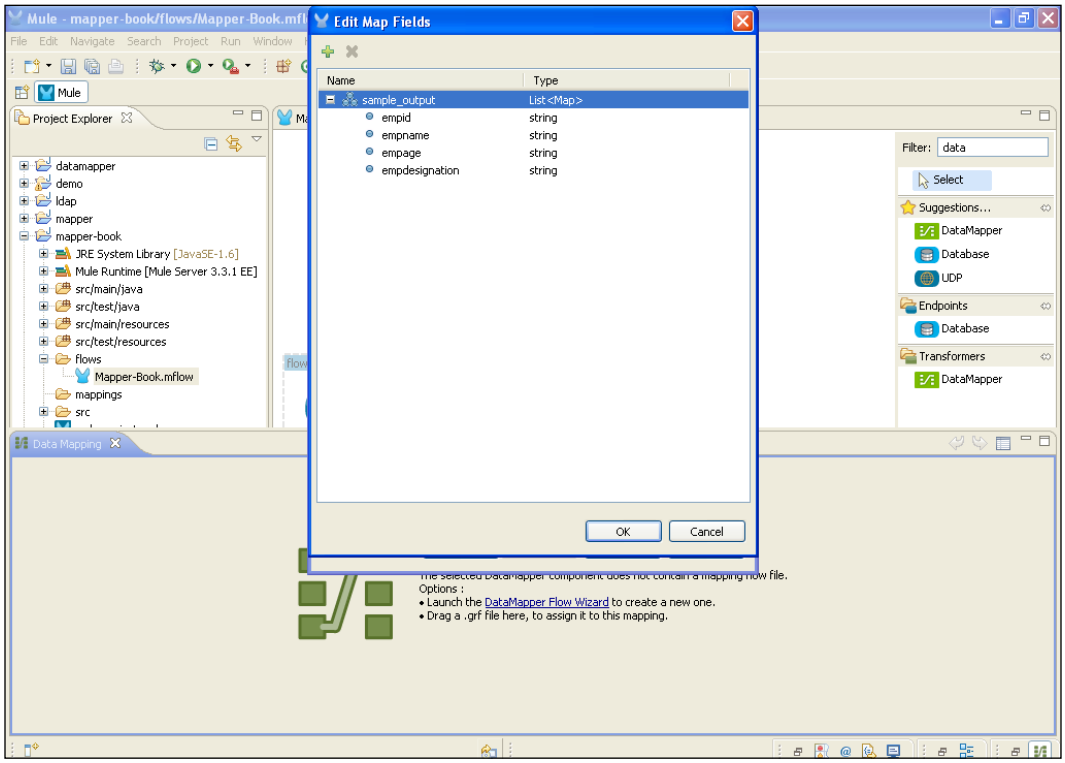
3. Here, you configure the **DataMapper** transformer and enter the name `CSV 2 Map` in the **Name** textbox. The filename `.grf` automatically appears in the **File name** textbox. The `.grf` extension file is used for the attributes that you map. Click on the **Next** button.



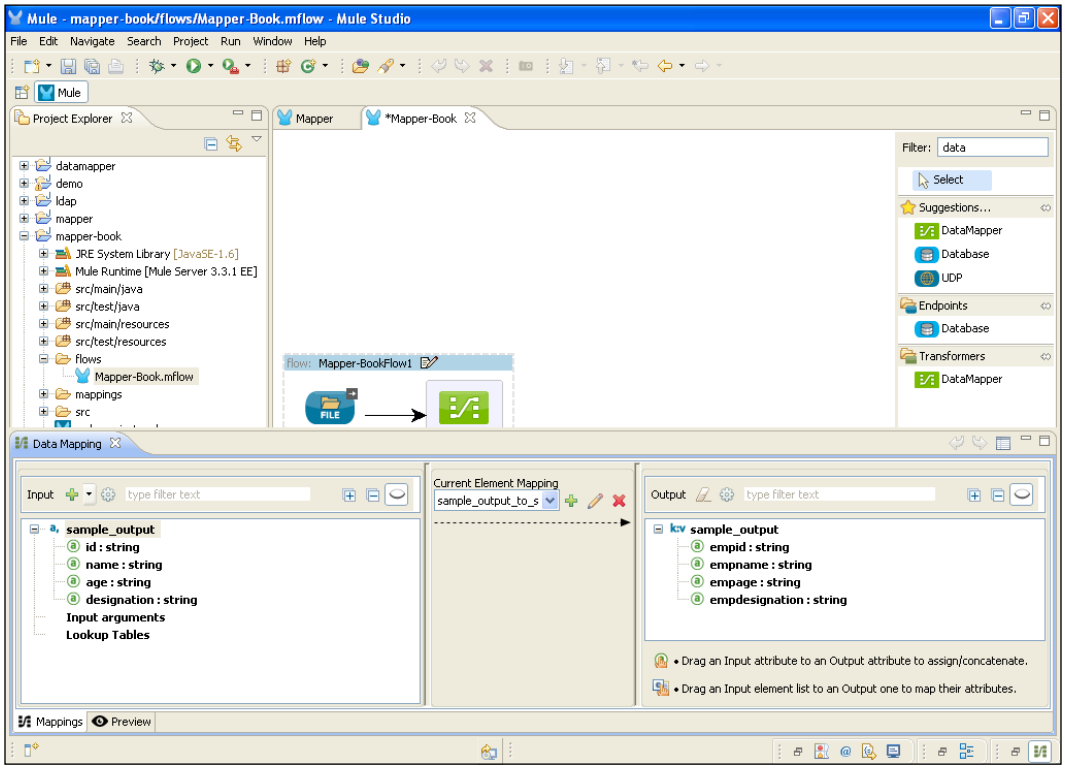
4. The DataMapper feature requires three files to hold the configuration information: the `.grf` file, the input type, and the output type. For the input type, you have to select the CSV type from the drop-down list. Click on the ellipsis (**...**) button to the right and choose the `.csv` file. In the output pane, use the drop-down list in the **Type:** field and select **Maps/List of Maps**. Note that you cannot specify a sample file for the `Maps` format. Click on the **Edit Fields** button to the right of the **Name:** field in the **Output** pane.



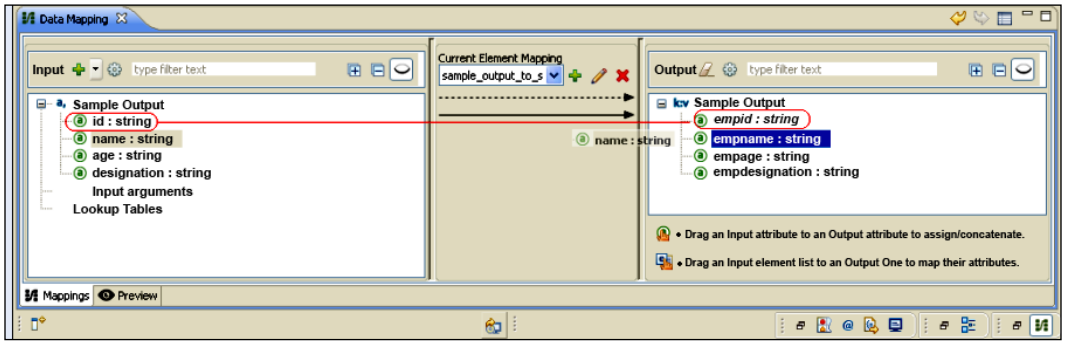
5. In the **Edit Map Fields** window, click on the plus button. Here, you enter the same column name that was entered in the database. Click on **OK** to finish the **DataMapper** wizard.



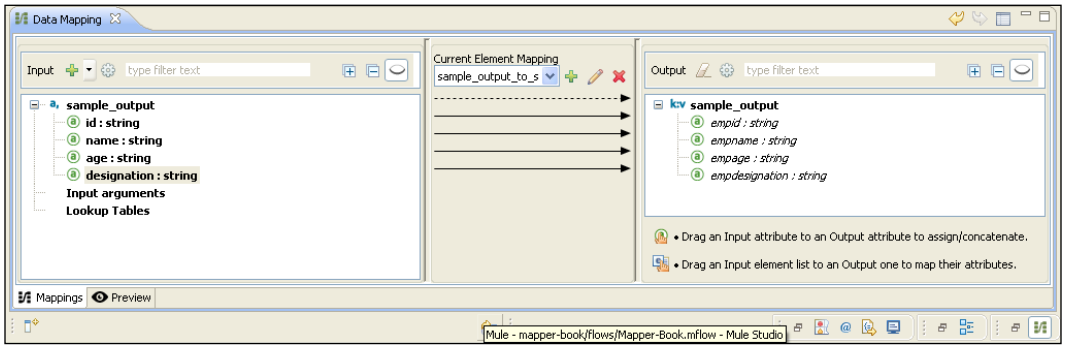
6. After you click on the **Finish** button, you will see a screen similar to the one shown in the following screenshot. The area on the left-hand side of the screen is for the `sample.csv` file attribute and the right-hand side of the screen area is for the DataMapper attribute.



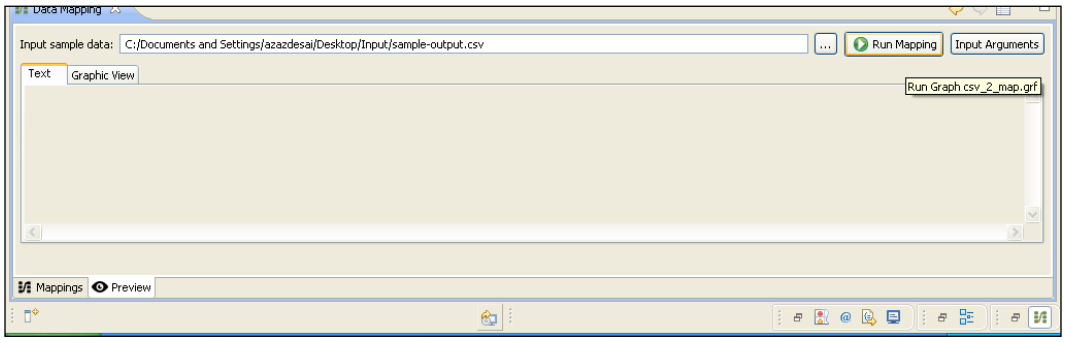
7. Click on **id: string** and drag it on top of **empid: string**. The solid black line indicates mapping.



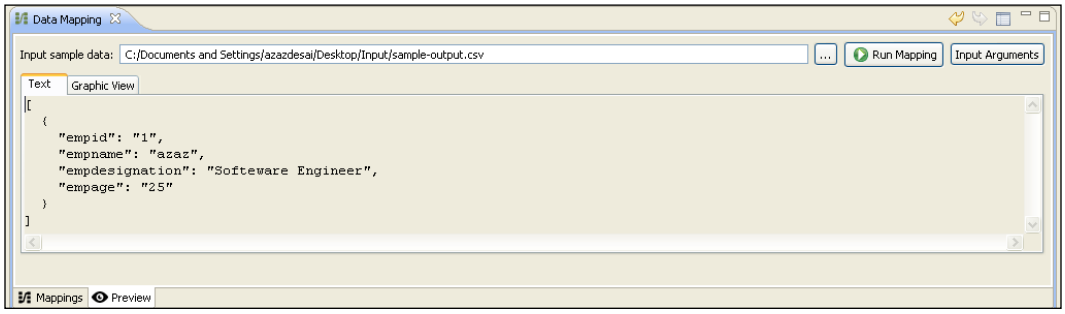
8. Drag all the attributes to be mapped onto the right-hand side. The final output looks like the following screenshot:



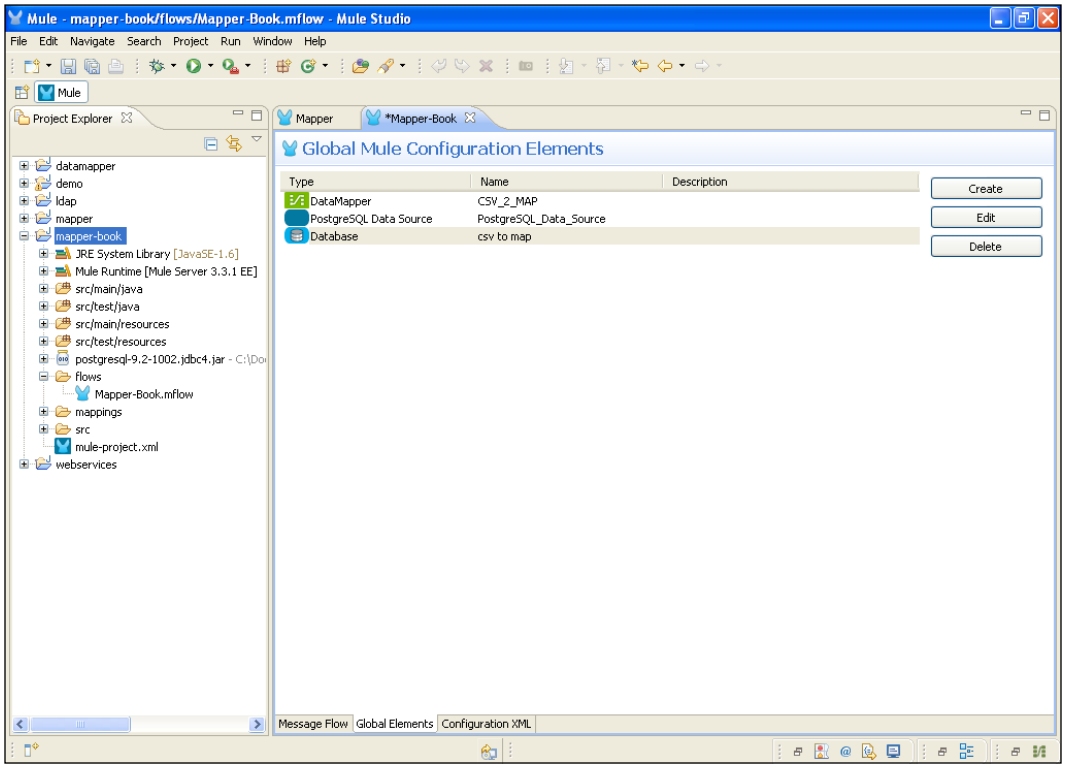
9. Click on **Preview**, and then click on **Run Mapping** on the right-hand side of your screen.



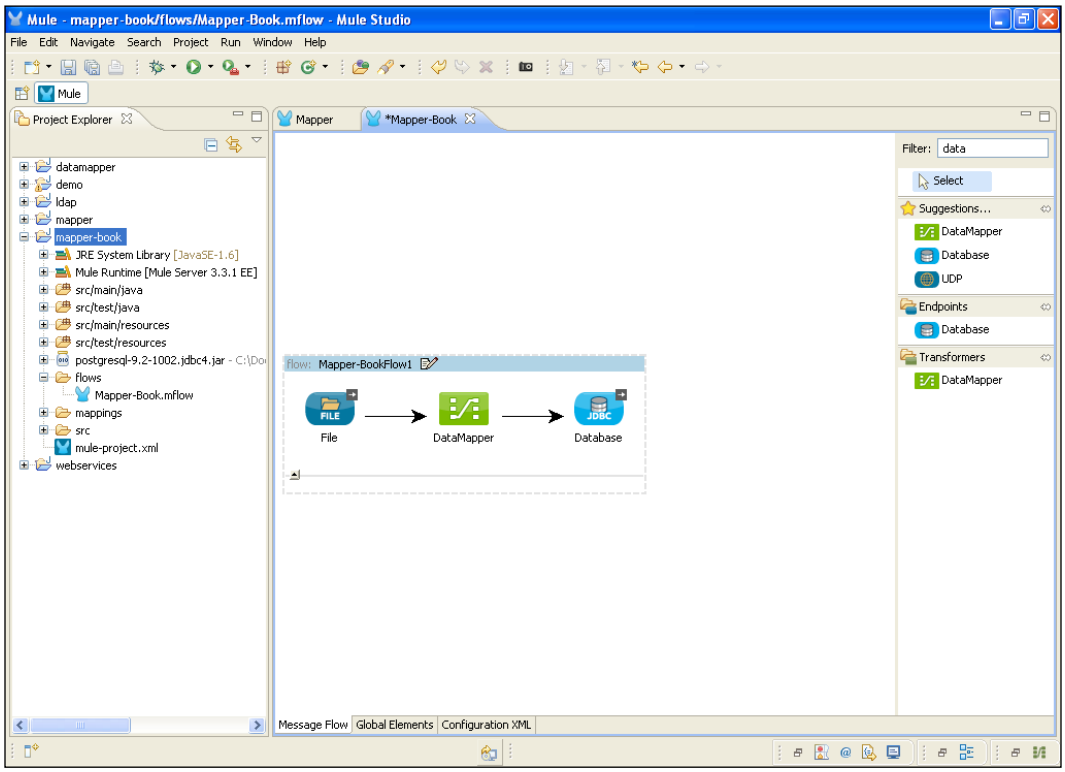
You will see the result being displayed on the screen. One really attractive feature is the possibility to test your mappings without the need to launch your Mule application.



10. Click on the **Global Elements** tab. Here, you have to configure the database that we have already seen in the previous example.



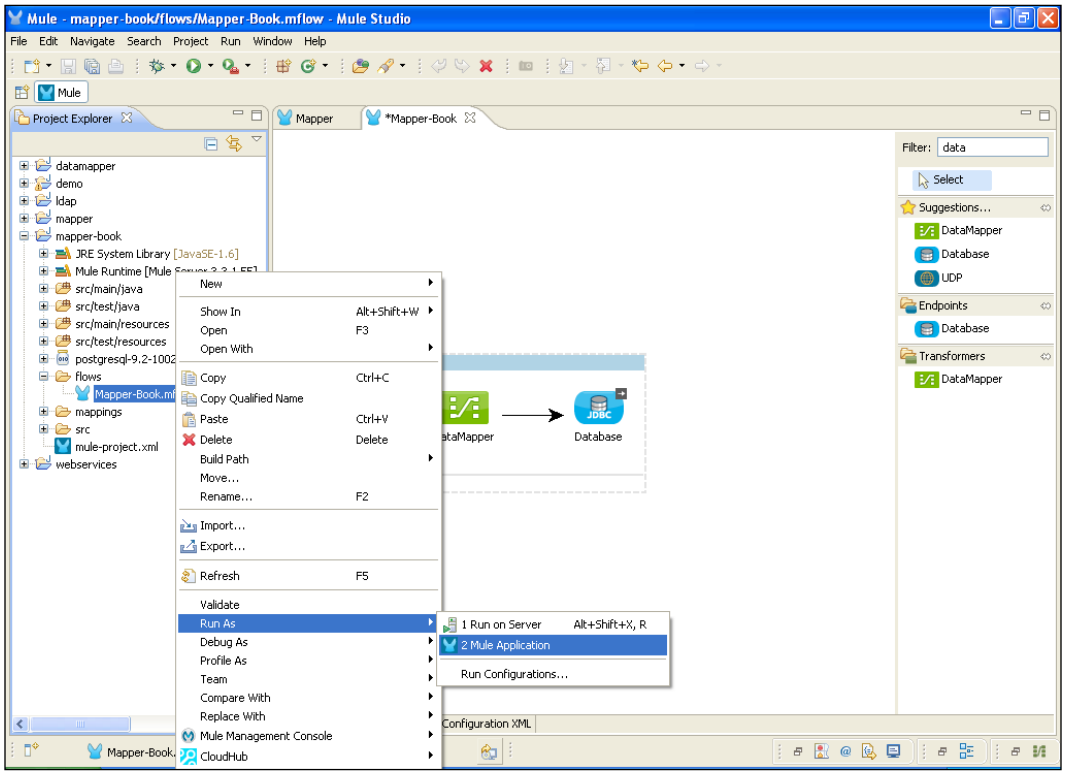
Your final flow should look like the following screenshot:



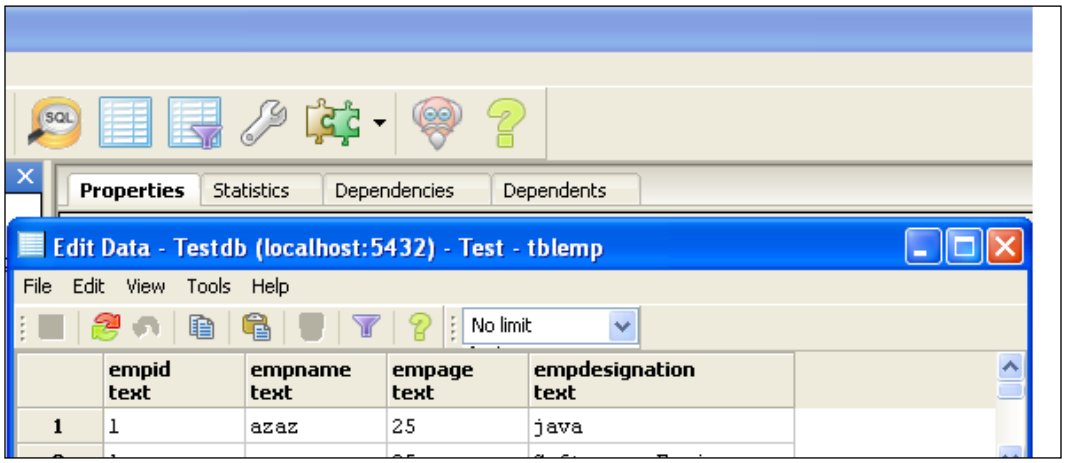
How it works...

In this section, you will learn how to deploy the application in Mule Studio and how it works.

1. To deploy the application code in the Mule server, go to **Run As** | **Mule Application**; the Mule server will deploy your application.



2. Open the PostgreSQL database and check the table entries. You will observe that the CSV data is transferred to the external database.



6
Configuring Filters

In this chapter, we will cover the different types of filters. You will also learn the following:

- Configuring the Logic filters – And/Or/Not
- Performing filtering according to the exception type
- Filtering messages by evaluating expressions
- Handling incoming events or messages using the Message filter
- Configuring the Wildcard filter
- Creating a Custom filter

Introduction

Filters specify conditions that must be met for a message to be routed to a service. Several standard filters come with Mule where you can create your own filters. You will learn about some filters in this chapter.

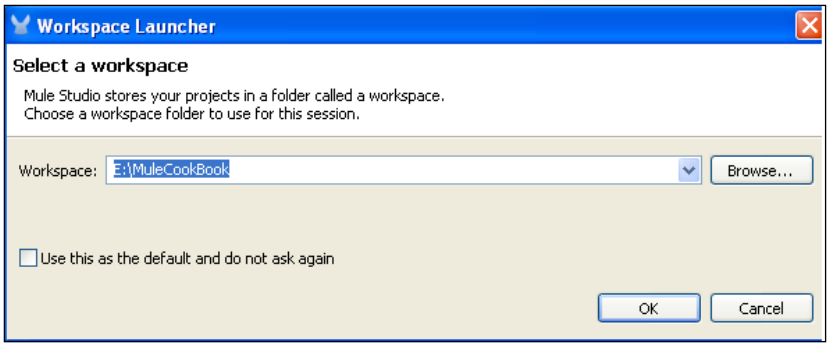
Configuring the Logic filters – And/Or/Not

Use the And filter to join two or more filters. The And filter accepts a message and returns true only if all of its enclosed filters return true. The Or filter accepts the message if the message matches the criteria of any of its filters. The Wildcard filter applies a wildcard pattern to the message payload. This filter applies a string to the payload, so you might also want to apply a Payload Type filter to the message using an And filter to make sure the payload is a string.

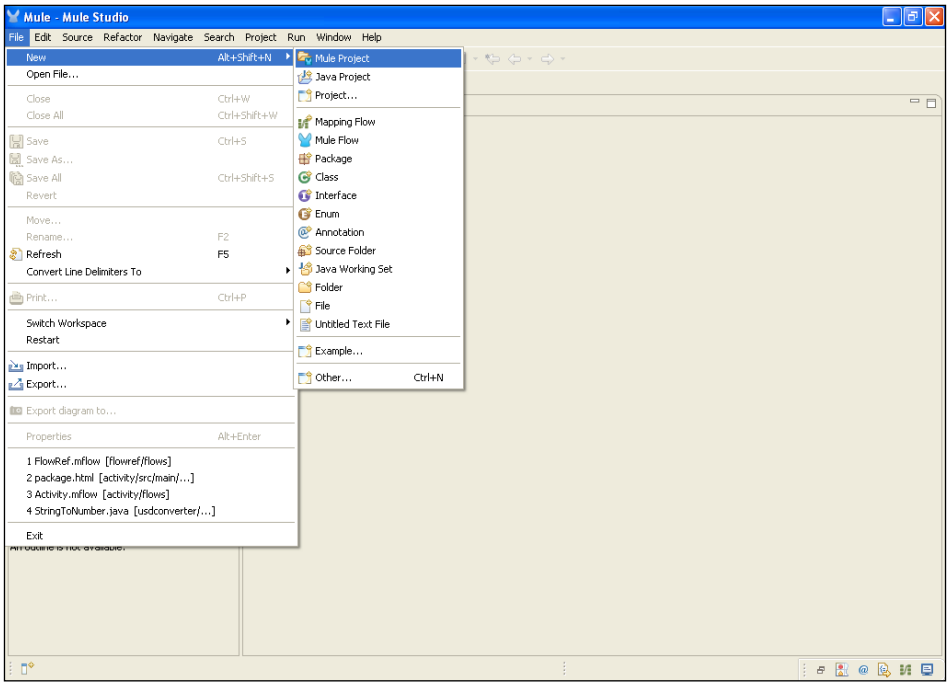
Getting ready

In this section, you will use three components: the HTTP Endpoint, the Or filter, and the Java component. You will also learn how to configure the Or filter. Perform the following steps to create a new project in Mule Studio:

1. Open Mule Studio and enter the workspace name as shown in following screenshot:



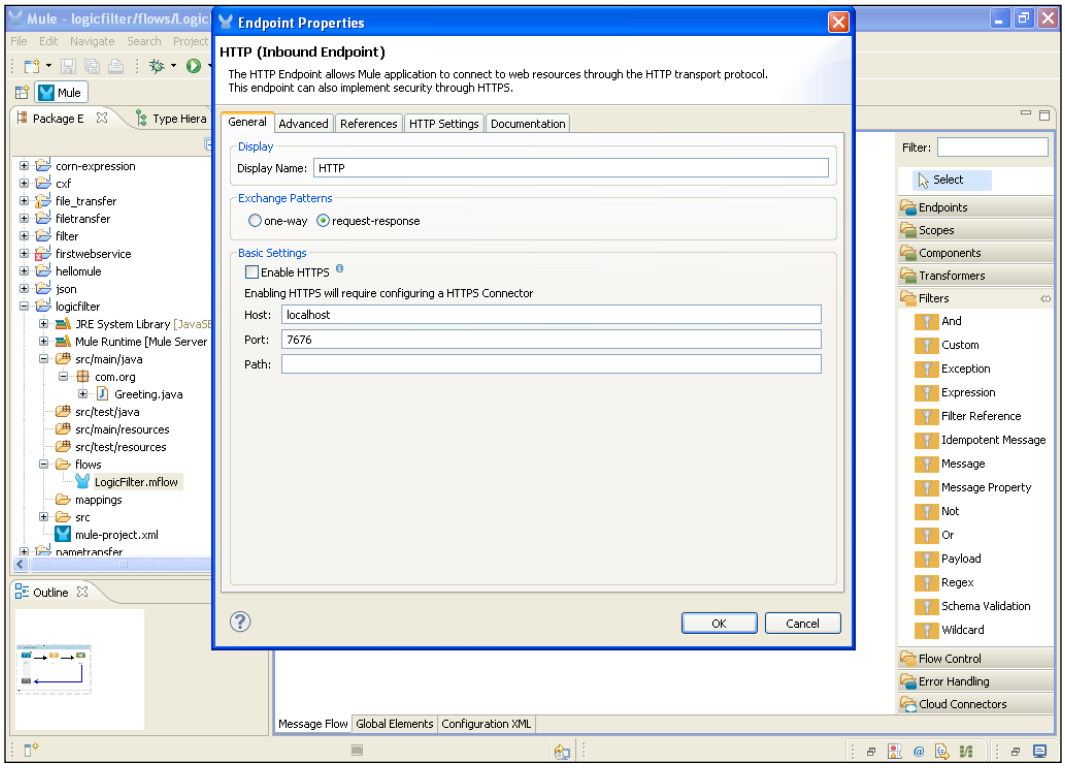
2. To create a new project, Go to **File** | **New** | **Mule Project**. Enter the project name `Logic Filter`, click on **Next** and then on **Finish**. Your new project is created now, so you can start with the implementation.



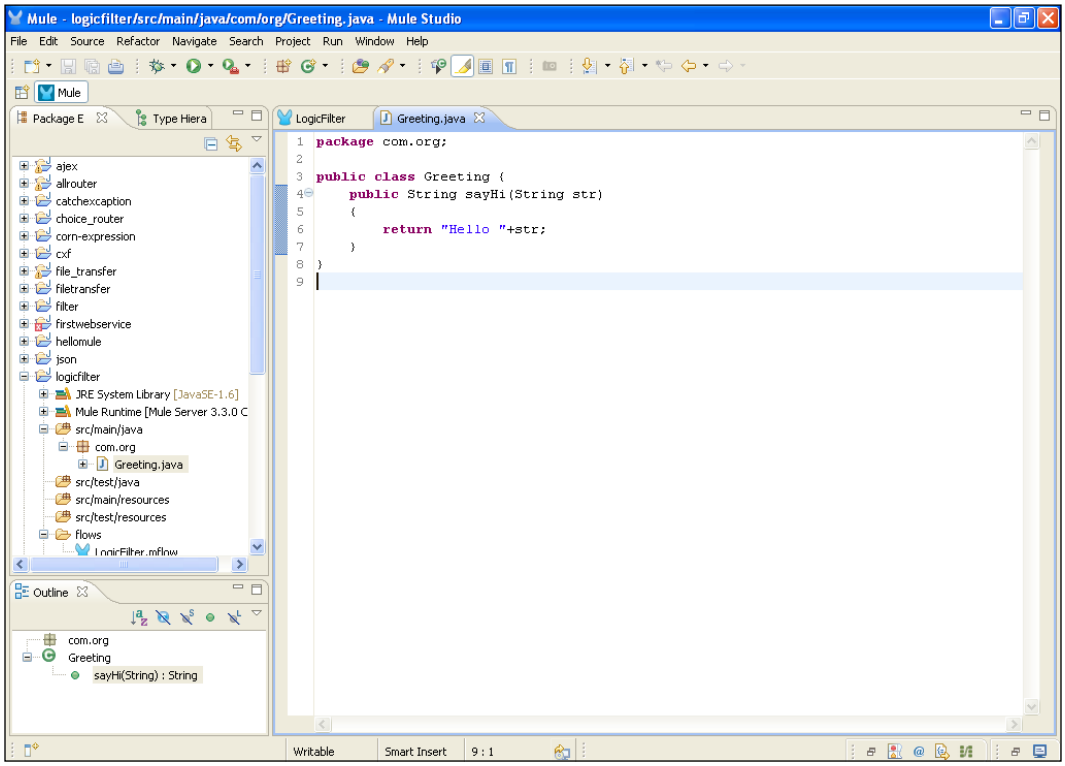
How to do it...

In this section, you will see how to use a Logic filter and how to configure the Java component.

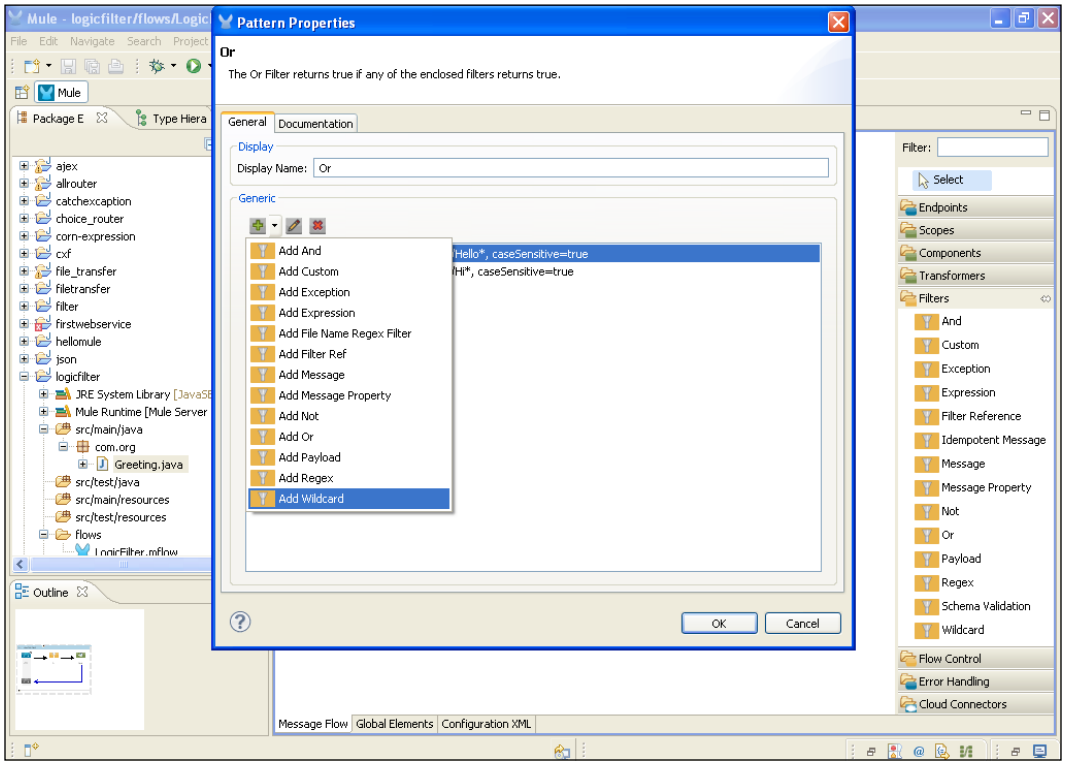
1. Go to the `LogicFilter.mflow` file and drag the **HTTP** Endpoint onto the canvas. Double-click and configure it. Change the port number, and click on the **OK** button.



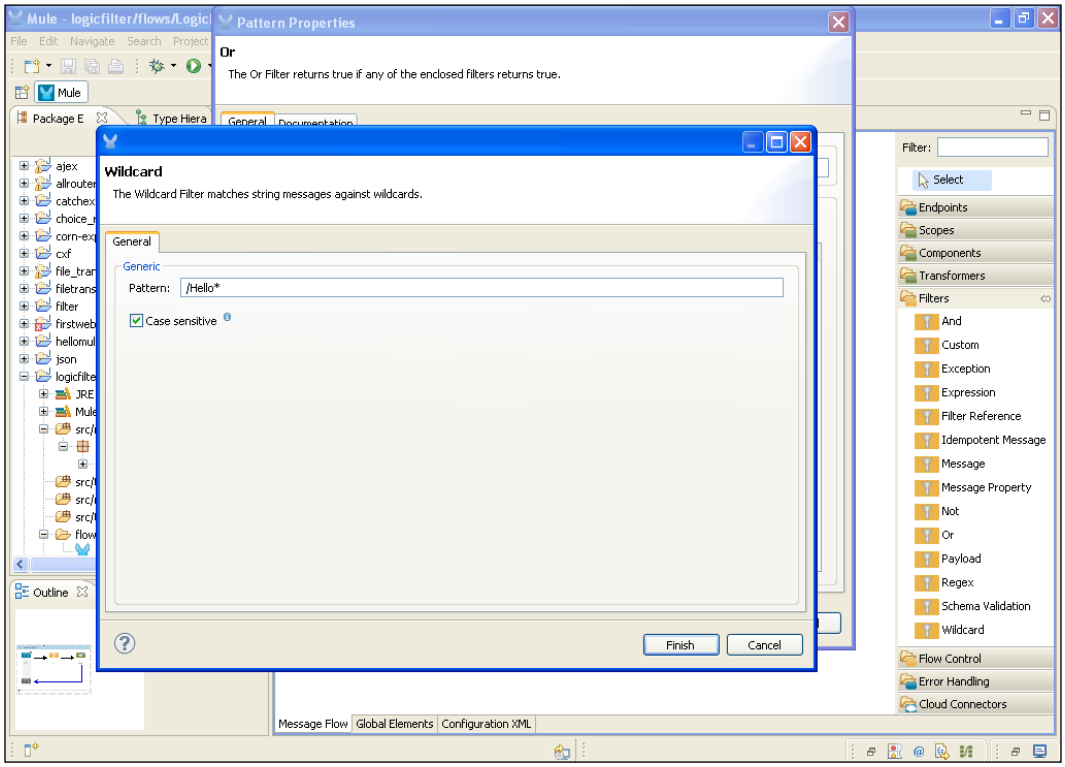
2. To create a class, go to `src/main/java`, right-click on it, and go to **New** | **Class**. Create a class called `Greeting` under the package `com.org`; here, we have created the `sayHi` method and its return type is set to `String`.



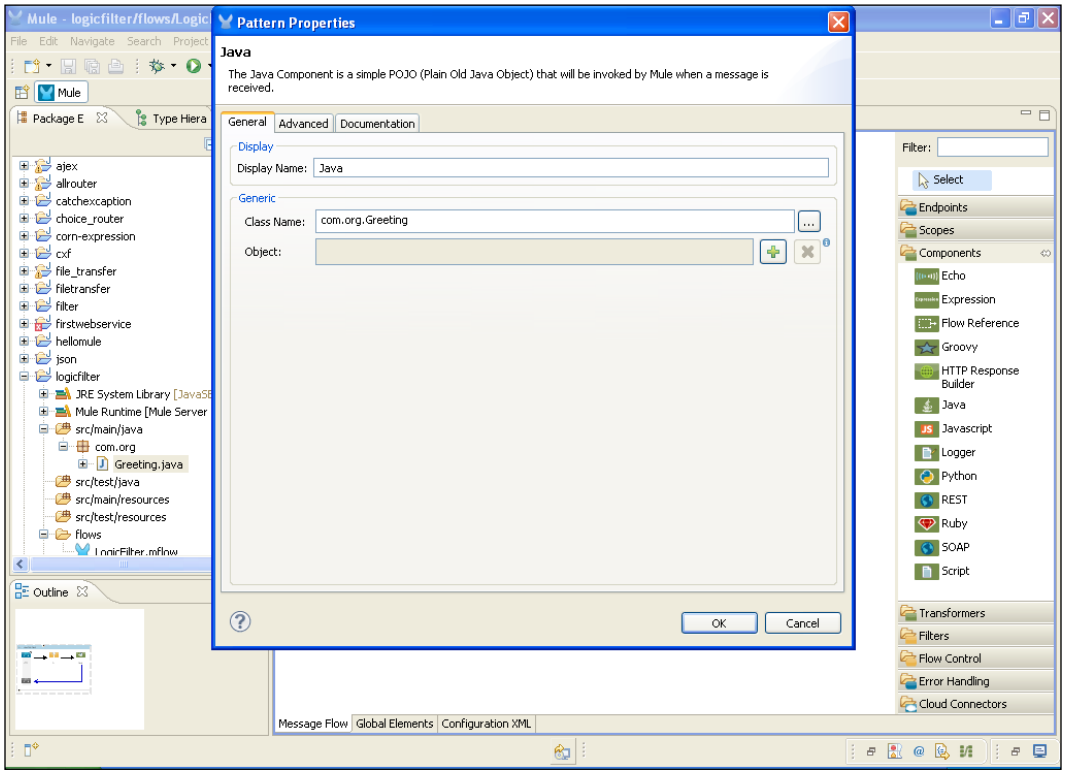
3. To configure a Logic filter, go to the `LogicFilter.mflow` file and drag the **Or** filter onto the canvas. Double-click and configure it. Here, you have to select the **Add Wildcard** filter twice.



4. In the **Add Wildcard** filter, you have added a pattern `/Hello*`. Add another pattern in the second **Add Wildcard** filter, `/Hi*`, and click on the **Finish** button. This pattern means that the string starts with `Hi` or `Hello`; if not, the output will not be displayed on the console.



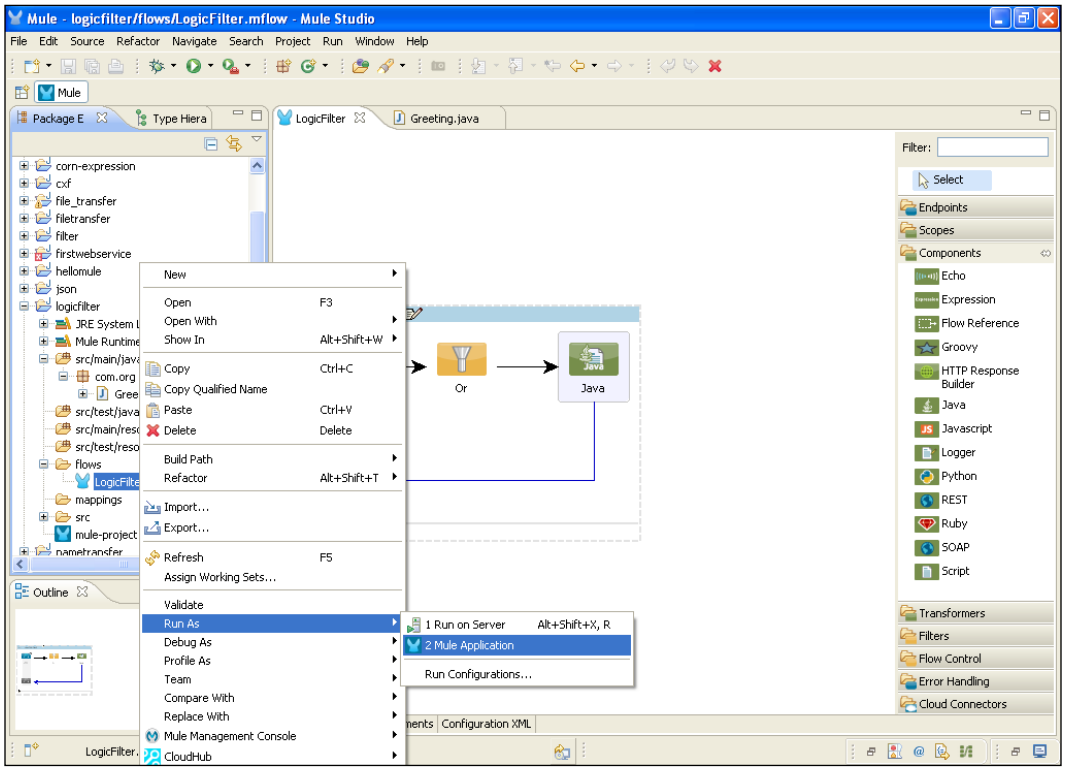
5. To configure a custom class, you have to drag the **Java** component onto the canvas and import the custom Java class that you created before.



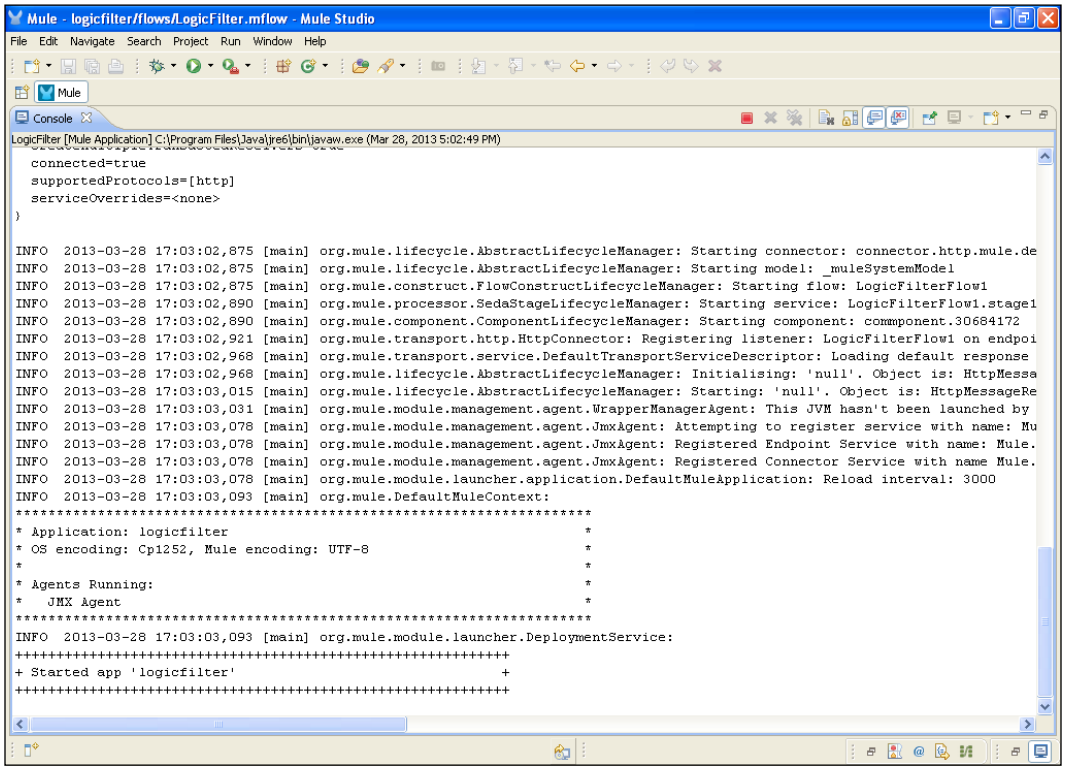
How it works...

In this section, you will see how to deploy the application using Mule Studio, and after deploying the application, you will see how it will run.

1. Now you are ready for the deployment. If you haven't saved your application code, please do it. After saving your project, right-click on the `LogicFilter.mflow` file and go to **Run As** | **Mule Application**.



2. If your application code is successfully deployed, you will see the message `Started app 'LogicFilter'` on the console.



3. To see the output in your browser, copy the URL `http://localhost:7676/HiAzaz` and paste it in your browser. If the string doesn't match the filter, the output will not be displayed on the console.

Performing filtering according to the exception type

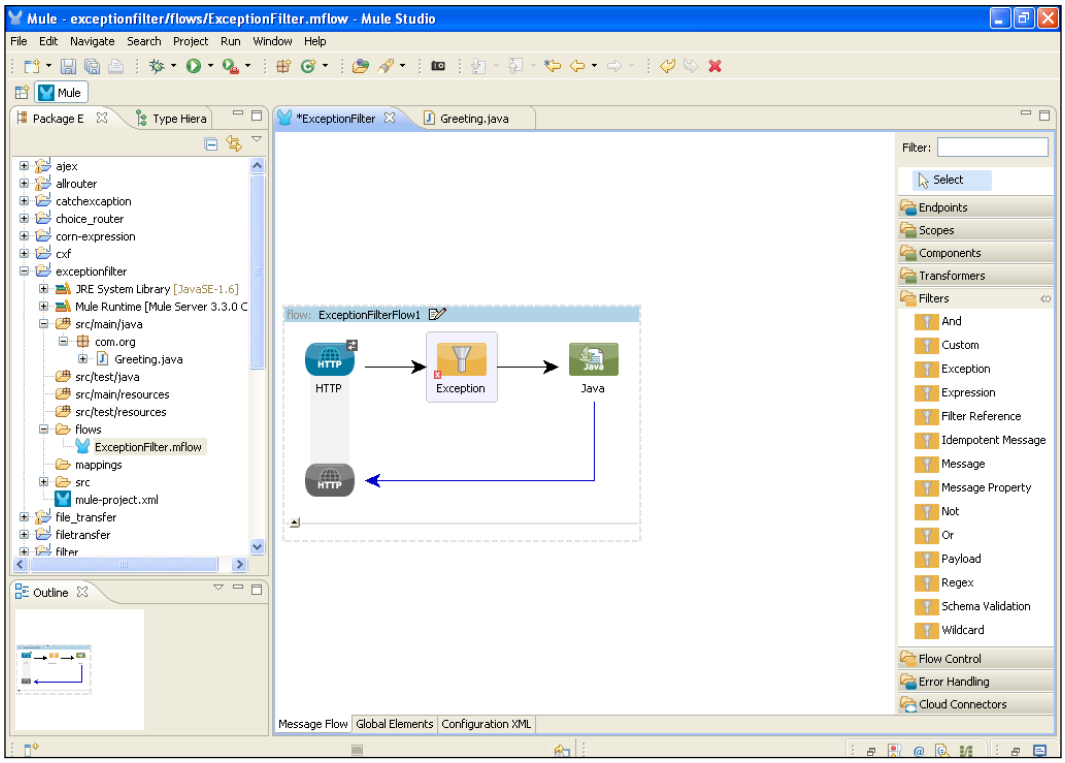
An exception is a problem that arises during the execution of a program. An exception can occur for many different reasons, including the following: user errors, programmer errors, and physical resources that have failed in some manner.

Getting ready

You can supply the class indicating the exception type to the **Expected Type:** property. For example, for a null pointer exception type, you might set **Expected Type:** to `java.lang.NullPointerException`.

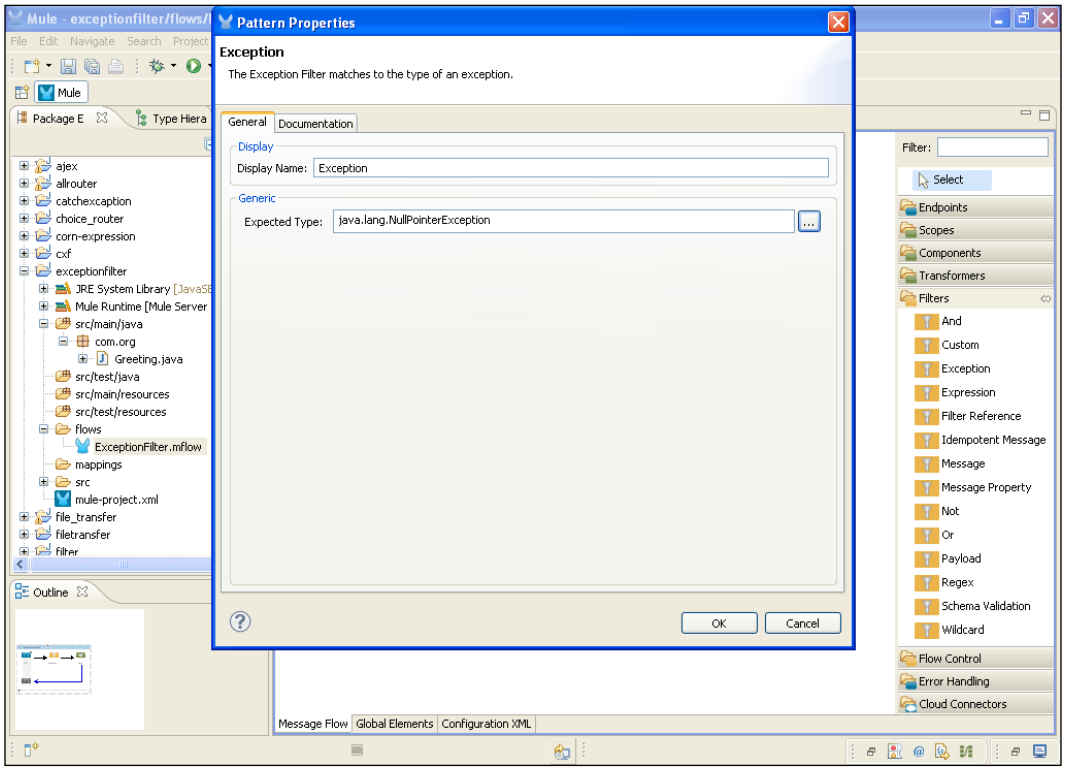
How to do it...

From the palette, drag the **Exception** filter onto the canvas. The **Exception** filter handles errors.



How it works...

Double-click on the **Exception** filter to configure it. Here, you can define the exception depending upon the requirement.



Filtering messages by evaluating expressions

Mule ESB provides several default expression evaluators allowing you to embed expression logic in a variety of expression languages; alternatively, you can create your own evaluators to support additional languages. This filter lets you evaluate a range of expressions. It supports expression types such as header, payload, regex, and wildcard. Set the evaluator to specify the expression evaluator type to be used. The **RegEx** filter applies a regular expression pattern, such as a pattern that includes wildcards or other character substitution symbols in the message payload. The filter applies the `toString()` method to the payload to convert the payload into a string.

Getting ready

In Mule, there are different types of evaluator expressions, such as the header payload type, the exception type, wildcard, regex, bean, and groovy. Each expression depends on the evaluator type. For example, if the expression type is XPath, bean, or OGNL, the expression should be a Boolean. Expressions allow you to extract information from the current message or determine how to handle the message. Expressions are very useful with routers.

How to do it...

In this section, you will see how to create a custom evaluator and how to use it in a Mule configuration file.

1. To create a custom evaluator, the first step is to implement the `ExpressionEvaluator` interface. Note that this interface implements `NamedObject`, which allows the evaluator to be named. This is the name you use for the evaluator attribute when using this evaluator in the configuration. After that, create another class named `MessageHeaderExpressionEvaluator` that implements with the interface, which we have created before.
2. After creating your custom expression evaluator, you must register it with Mule. If you are using an XML configuration, you can just configure your expression evaluator as a bean, and Mule will discover it.

How it works...

In this section, you will see different types of expressions.

XPath expressions

XPath expressions use the standard XPath query language based on JAXP—the Java API for XML processing. Refer to *Using Filters* (`http://www.mulesoft.org/documentation/display/current/Using+Filters`) for more information about XPath expressions.

```
<expression-filter evaluator="xpath" expression="(msg/header/
resultcode)='success'"/>
```

JXPath expressions

JXPath is an XPath interpreter that can apply XPath expressions to graphs of objects of all kinds, including JavaBeans, Maps, Servlet contexts, DOM objects, and mixtures of these objects. For more information about JXPath, refer to *Using Filters* (`http://www.mulesoft.org/documentation/display/current/Using+Filters`).

```
<expression-filter evaluator="jxpath" expression="(msg/header/
resultcode)='success'"/>
```

OGNL expressions

OGNL is a simple, but a very powerful, expression language for plain Java objects. Similar to JXPath, OGNL works on object graphs. Filters using OGNL expressions enable simple and efficient content routing for payloads. Refer to *Using Filters* (`http://www.mulesoft.org/documentation/display/current/Using+Filters`) for more information.

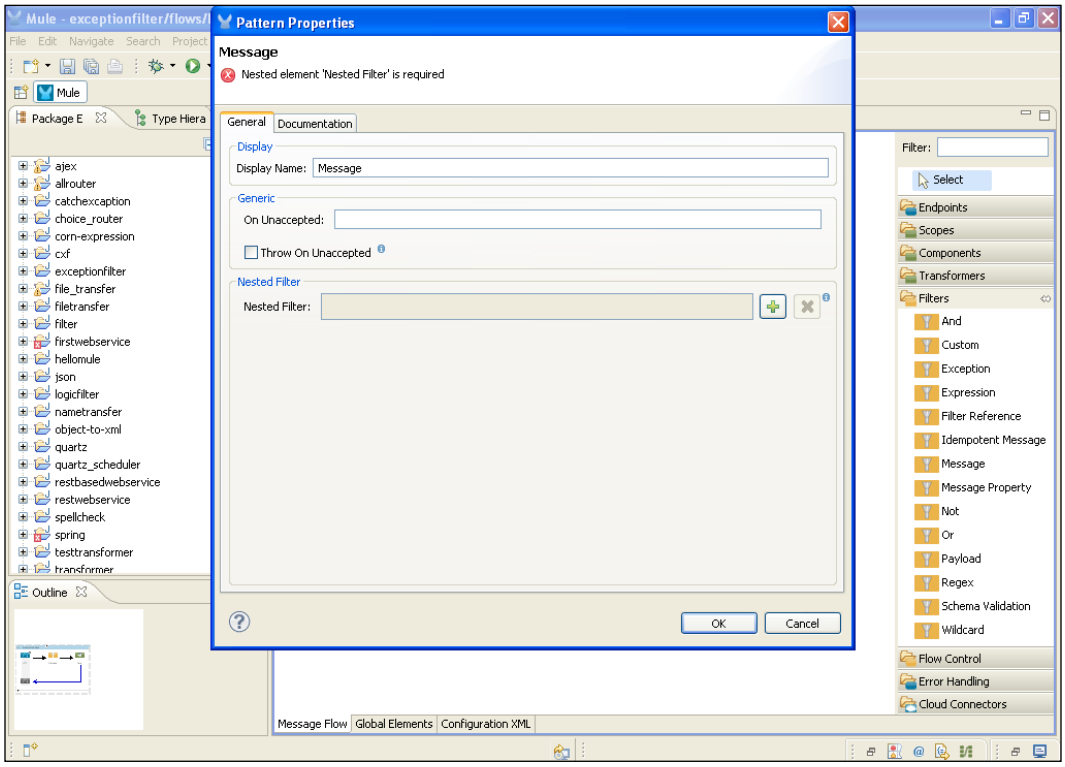
```
<expression-filter evaluator="ognl" expression="[MULE:0].equals(44)"/>
```

Handling incoming events or messages using the Message filter

The Message filter is used for deciding whether to handle incoming events or messages. You can use the **On Unaccepted** property to optionally specify the name of the message processor that should handle any unaccepted events.

Getting ready

Drag-and-drop the **Message** filter from the palette on the canvas and configure it. Double-click on the **Message** filter. Here you can see the **Throw on Unaccepted** checkbox. Select this checkbox to throw an exception if a message or event is not handled. The default—when not checked—is to not throw an exception. You can use the **On Unaccepted:** property to optionally specify the name of the message processor.

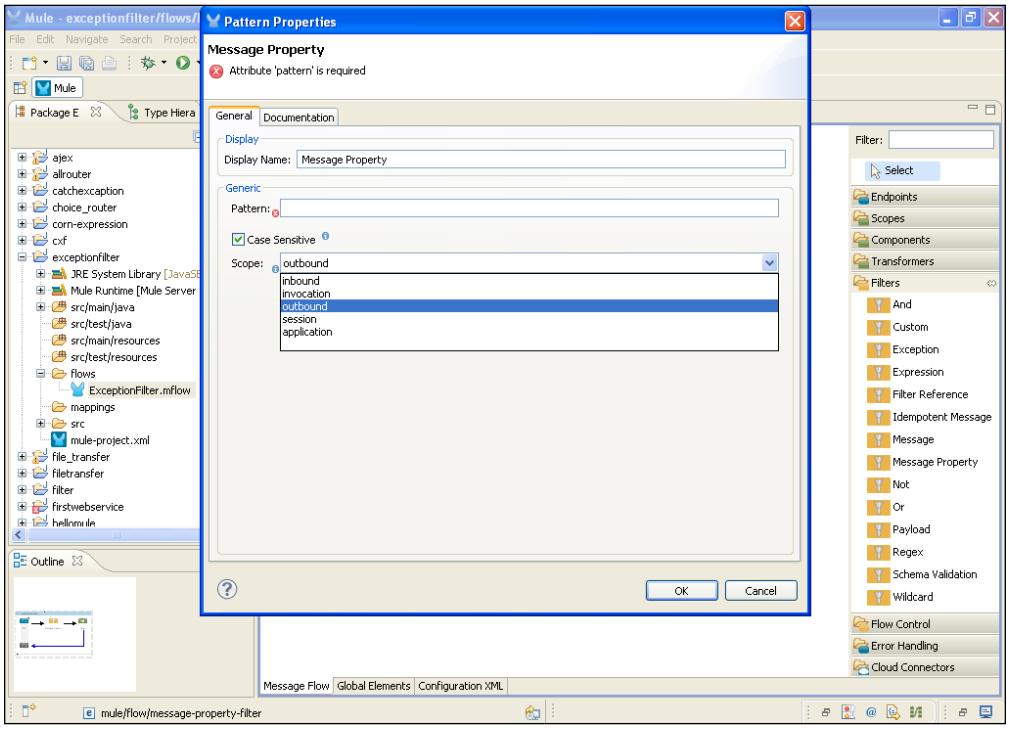


How to do it...

The Message filter is used to route messages using logic based on the values of message properties. You can define one or more message properties. Using this filter, you can access the message metadata, including transport-specific and user-defined properties. These message properties represent all the meta information about the message that is available from the underlying transport. For example, if you receive a message via an HTTP Endpoint, you can use this filter to check HTTP header values. The pattern is expressed as a key/value pair, where key is the name of the property. By default, the **Case Sensitive** checkbox is selected representing that the comparison is case sensitive. Deselect this box if you want the association to be case insensitive.

How it works...

This filter can be very powerful because the message properties are exposed allowing you to reference any transport-specific or user-defined properties. For example, you can match one or more HTTP headers for an HTTP event, match properties in JMS and e-mail messages, and much more. You can define **Scope** in the message filter properties. There can be different types of scopes available in the message properties. You will see different types of scopes in the following screenshot. The expression syntax has been improved to support scopes. The **Scope** part is optional and is case insensitive. The default scope is **outbound**. The general syntax pattern is `<evaluator>:<scope>:<expression>`; for example, `header:OUTBOUND:CookBook`.



You will see the following different types of scopes being used in the **Message Property** window:

- **inbound**: This specifies the properties/headers that come from a client's request.
- **invocation**: This is used mostly internally by Mule for the duration of this flow's call.
- **outbound**: This specifies the values deemed to be sent out from this flow. They become either request properties or response properties for the next flow in the case of a synchronous invocation.

- **session**: This specifies the values that are passed from invocation to invocation.
- **application**: This scope is used when you create two different applications.

Configuring the Wildcard filter

The Wildcard filter applies a wildcard pattern to the message payload. This filter applies `toString()` to the payload, so you might also want to apply a payload type filter to the message using an And filter to make sure the payload is a string.

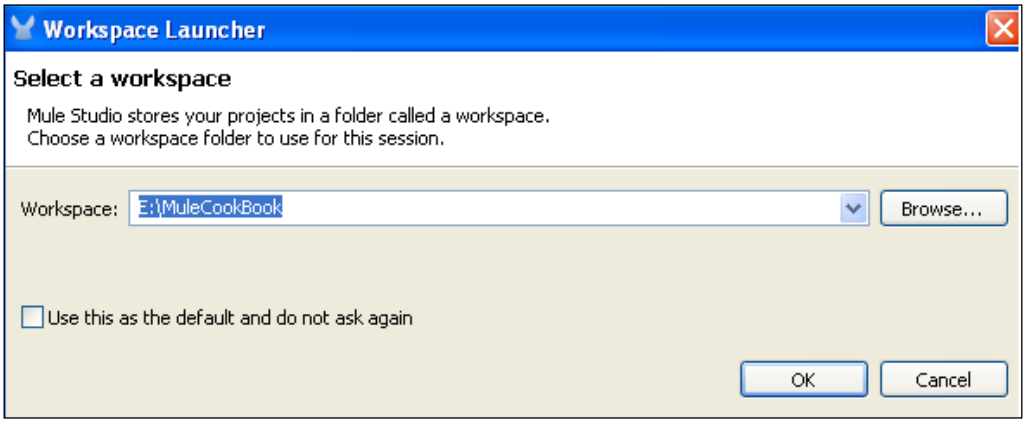
Getting ready

After dragging the Wildcard filter from the palette and dropping it on the canvas, double-click on the filter icon. A **Pattern Property** window shows up. There is only one attribute to configure for this filter and it is the pattern. You will see the example of using the Wildcard filter in this section.

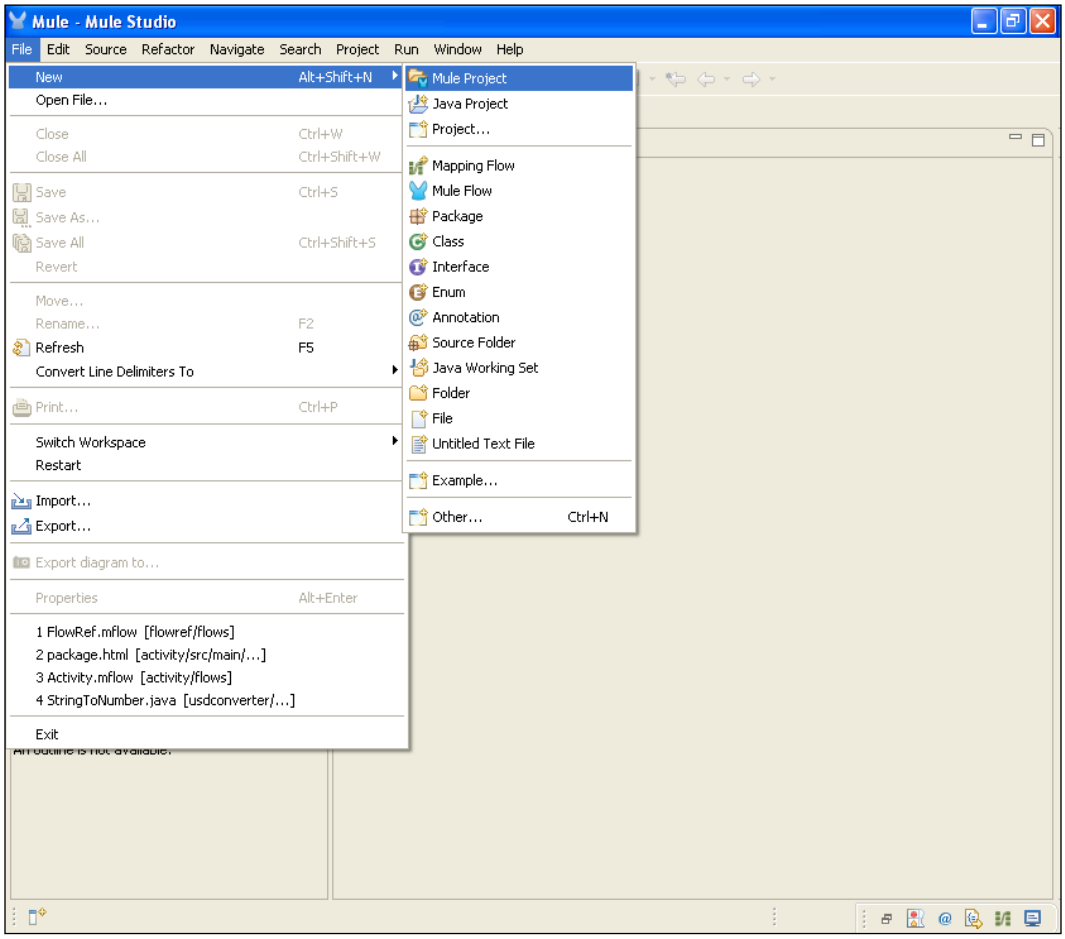
How to do it...

In this section, you will use three components: the HTTP Endpoint, the Wildcard filter, and the Custom transformer.

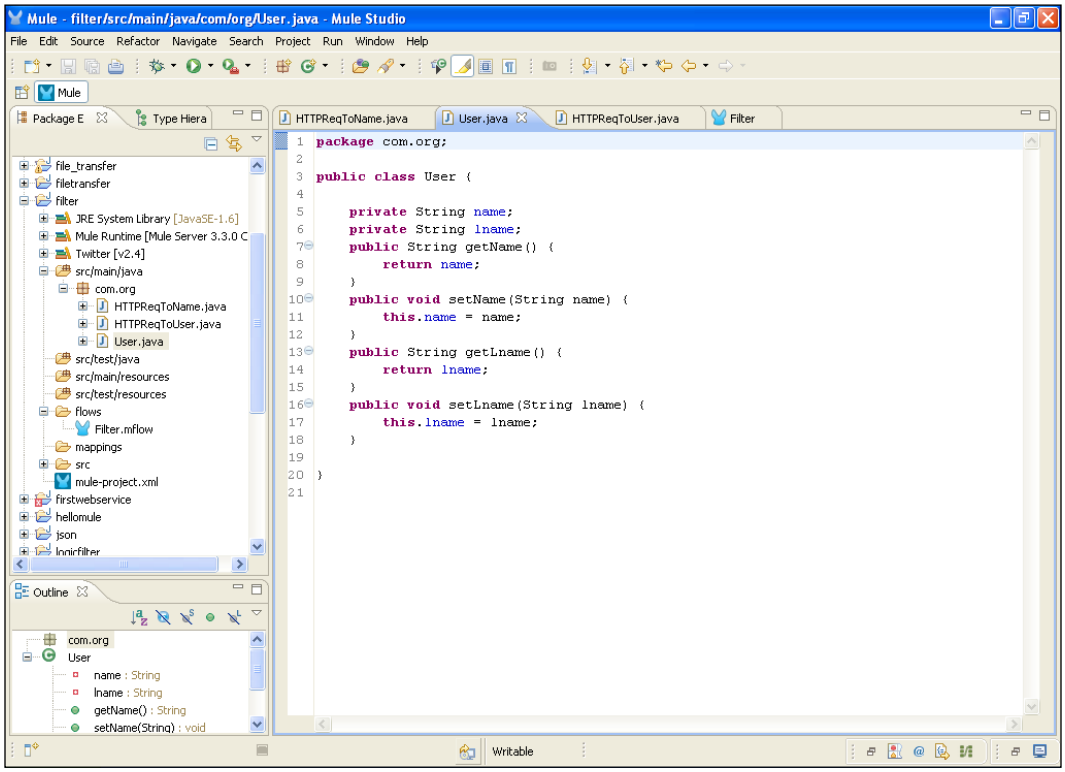
1. Open Mule Studio and enter the workspace name as shown in the following screenshot:



2. To create a new project, go to **File** | **New** | **Mule Project**. Enter the project name `Wildcard Filter`, click on **Next** and then on **Finish**. Your new project is created now, so you can start the implementation.



3. Go to `src/main/java`, right-click on it, enter the package name `com.org` and the class name `User`, and click on the **Finish** button. In this class, you have to use two private `String` variables, and then you can generate the `get` or `set` method.

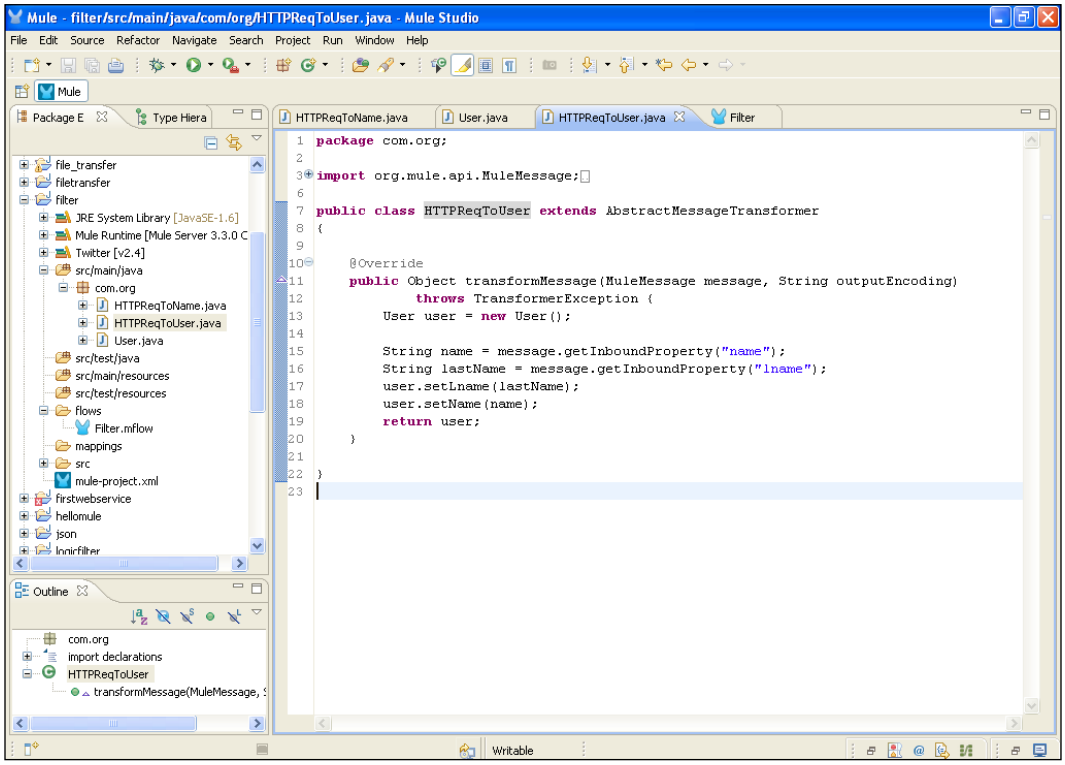


You create a class called `User` and generate the `get` and `set` methods:

```
package com.org;

public class User {

  private String name;
  private String lname;
  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
  public String getLname() {
    return lname;
```

```
    }
    public void setLname(String lname) {
      this.lname = lname;
    }

}
```

4. Go to `src/main/java`, right-click on it and go to **New** | **Class**, enter the package name `com.org` and the class name `HTTPReqToUser`, and click on the **Finish** button. Here, you create a custom transformer class and extend it with `AbstractMessageTransformer`. This way you override the `transformMessage` method.



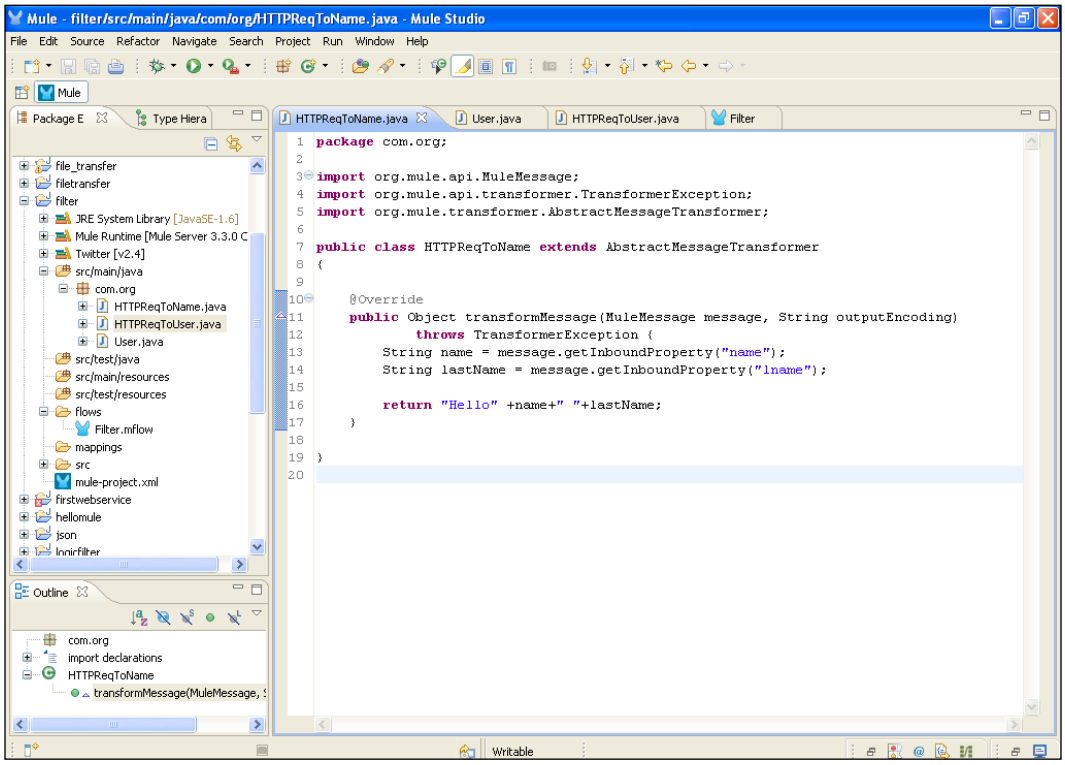

```
package com.org;

import org.mule.api.MuleMessage;

public class HTTPReqToUser extends AbstractMessageTransformer
{

    @Override
    public Object transformMessage(MuleMessage message, String outputEncoding)
            throws TransformerException {
        User user = new User();

        String name = message.getInboundProperty("name");
        String lastName = message.getInboundProperty("lname");
        user.setLname(lastName);
        user.setName(name);
        return user;
    }

}
```

You create a class and extend it with the `AbstractMessageTransformer` interface:

```
   import org.mule.api.MuleMessage;
import org.mule.api.transformer.TransformerException;
import org.mule.transformer.AbstractMessageTransformer;

public class HTTPReqToUser extends AbstractMessageTransformer
```

```
{

  @Override
  public Object transformMessage(MuleMessage message, String
outputEncoding)
      throws TransformerException {
    User user = new User();

    String name = message.getInboundProperty("name");
    String lastName = message.getInboundProperty("lname");
    user.setLname(lastName);
    user.setName(name);
    return user;
  }

}
```

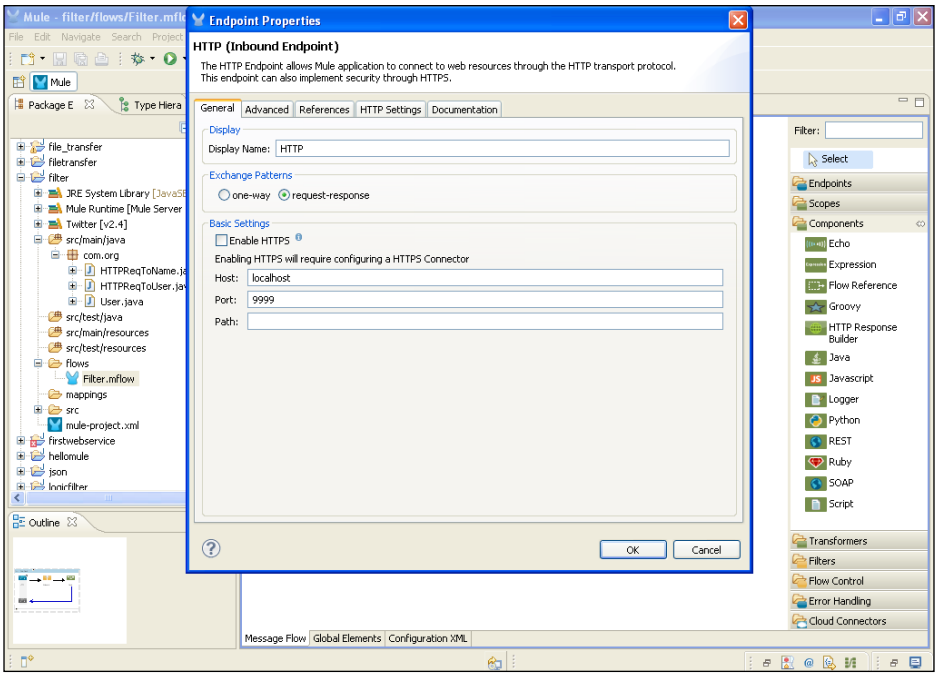
5. Go to `src/main/java`, right-click on it and go to **New** | **Class**, enter the package name `com.org` and the class name `HTTPReqToName`, and click on the **Finish** button.



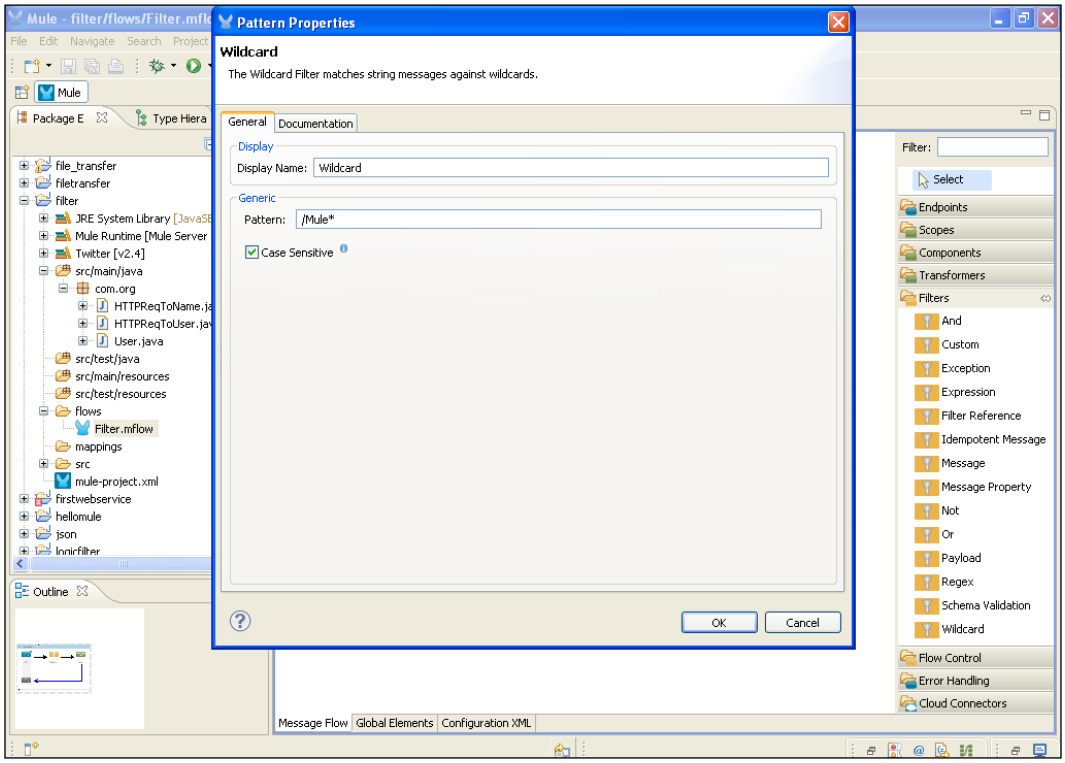
Here, you will create a class called `HTTPReqToName` and extend it with the `AbstractMessageTransformer` interface.

```
package com.org;

import org.mule.api.MuleMessage;
import org.mule.api.transformer.TransformerException;
import org.mule.transformer.AbstractMessageTransformer;

public class HTTPReqToName extends AbstractMessageTransformer
{

  @Override
  public Object transformMessage(MuleMessage message, String
outputEncoding)
      throws TransformerException {
    String name = message.getInboundProperty("name");
        String lastName = message.getInboundProperty("lname");

        return name+" "+lastName;
  }

}
```

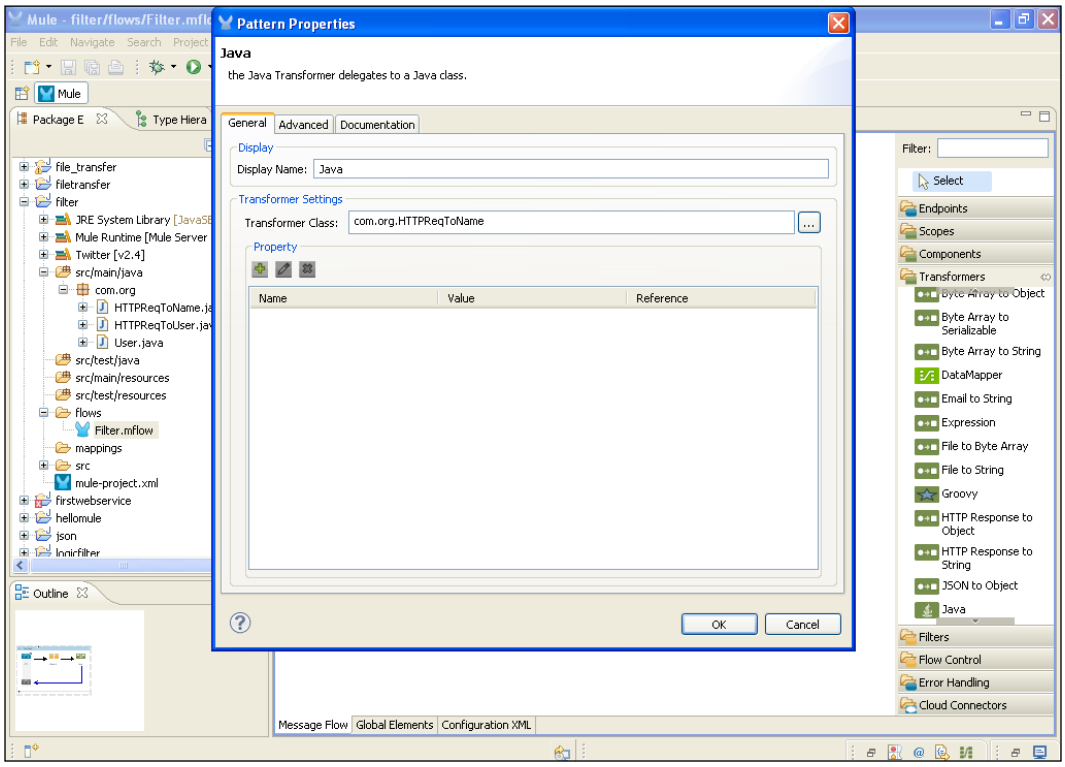
6. Go to the `Filter.mflow` file and drag the **HTTP** Endpoint onto the canvas. Double-click and configure it. Enter the port number and the hostname, and click on the **OK** button.



7. Drag the **Wildcard** filter onto the canvas. Double-click and configure it. In the **Wildcard** filter, you have to add a pattern, `/Mule*`; this means that a pattern should start with the text `Mule`.



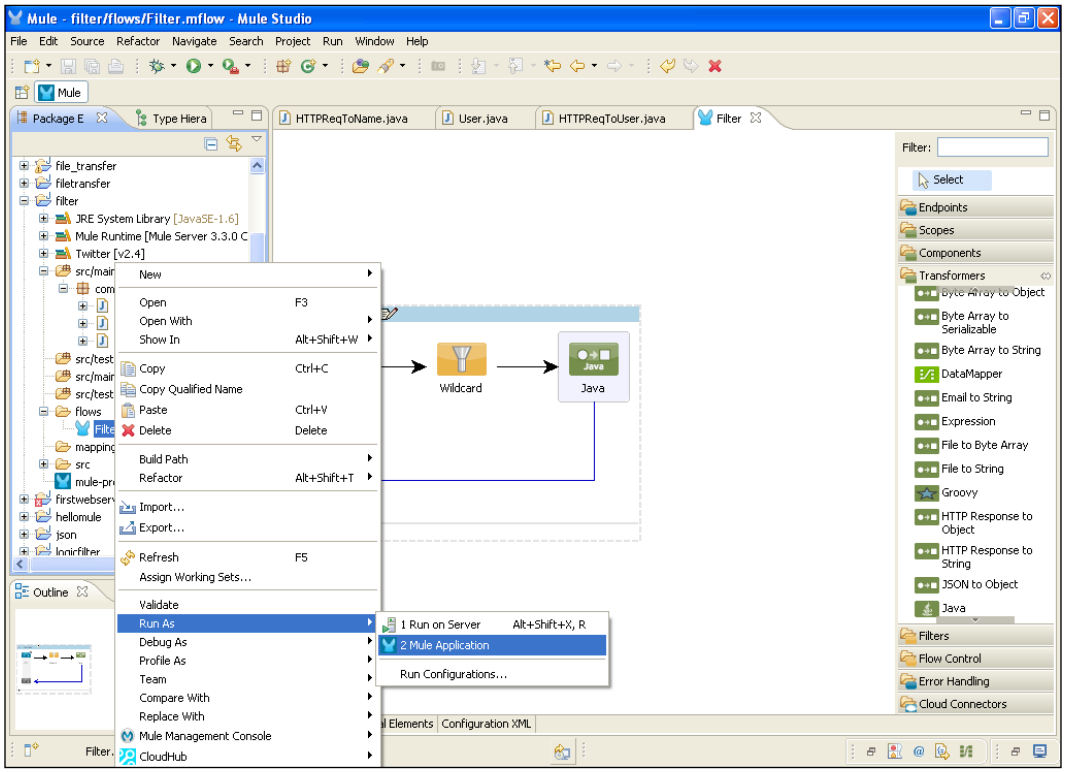
8. Drag the **Java** transformer onto the canvas, and double-click on it to configure it.



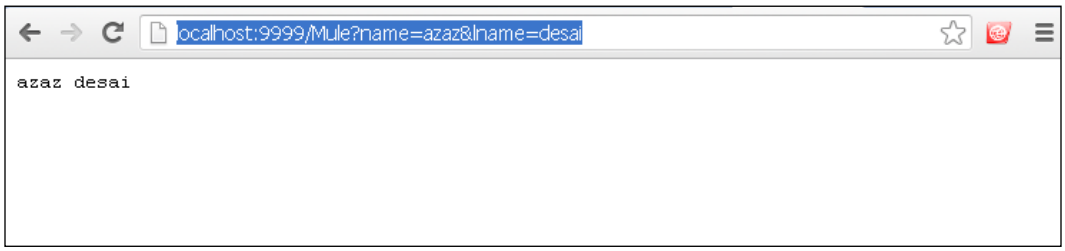
How it works...

In this section, you will see how to deploy the application on Mule Studio, and also how it works after deployment.

1. To deploy the application code in the Mule server, go to **Run As** | **Mule Application**, and the Mule server will deploy your application.



2. Then, open the browser and paste the URL `http://localhost:9999/Mule?name=azaz&lname=desai` to see the output.



Creating a Custom filter

Use a Custom filter to reference a user-implemented filter. Note that the reference is to a class implementing the `Filter` interface.

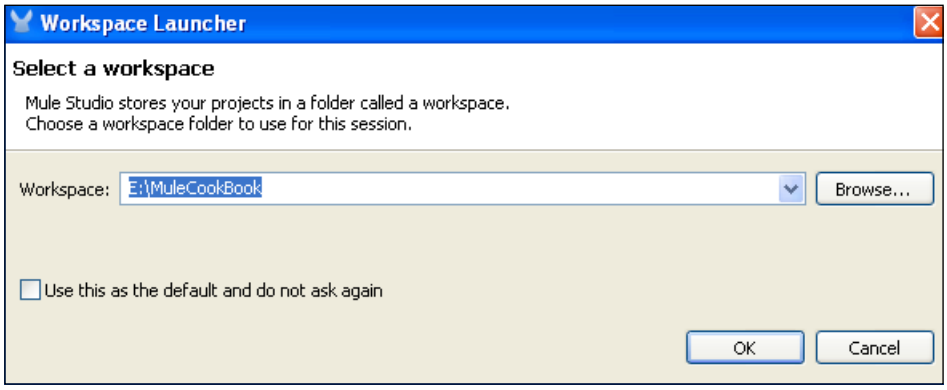
Getting ready

The `Filter` class is required if this is a global filter. After dragging the **Custom** filter from the palette and dropping it onto the canvas, double-click on the filter icon. A **Pattern Property** window shows up. There is only one attribute to configure for this filter and it is the `Filter` class.

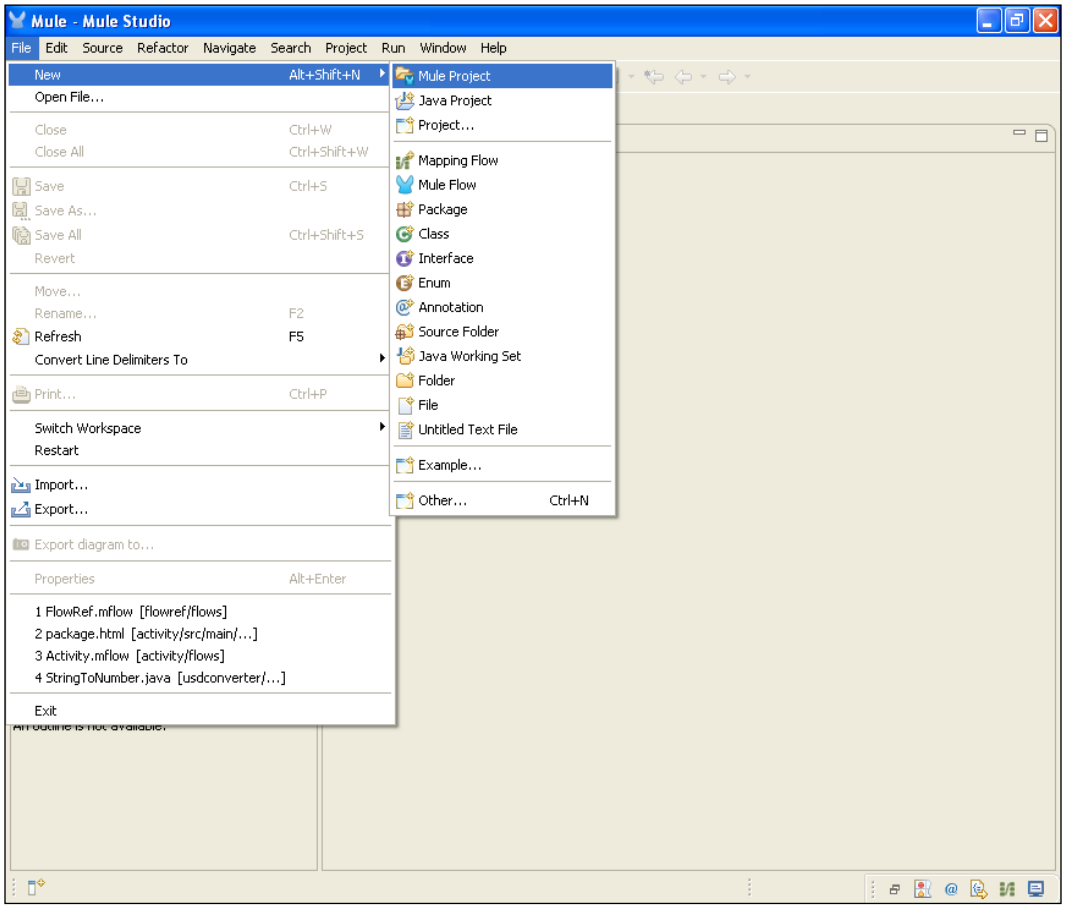
How to do it...

In this section, you will use three components: the HTTP Endpoint, the Echo component, and the Custom filter.

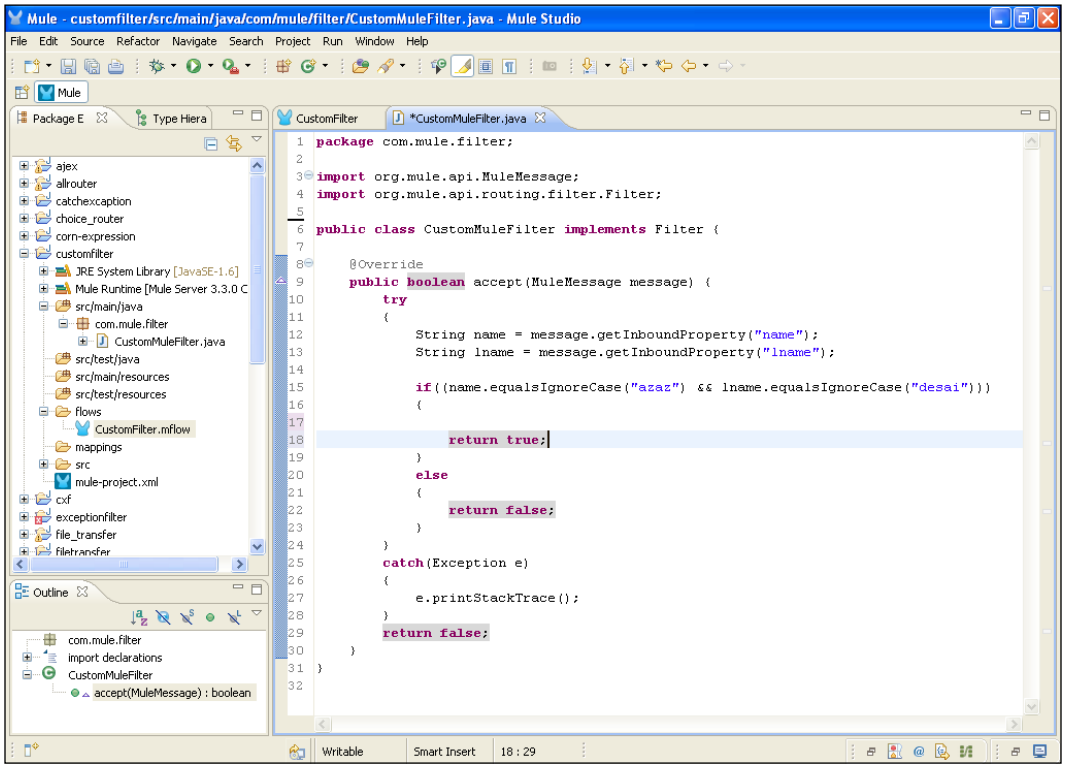
1. Open Mule Studio and enter the workspace name as shown in the following screenshot:



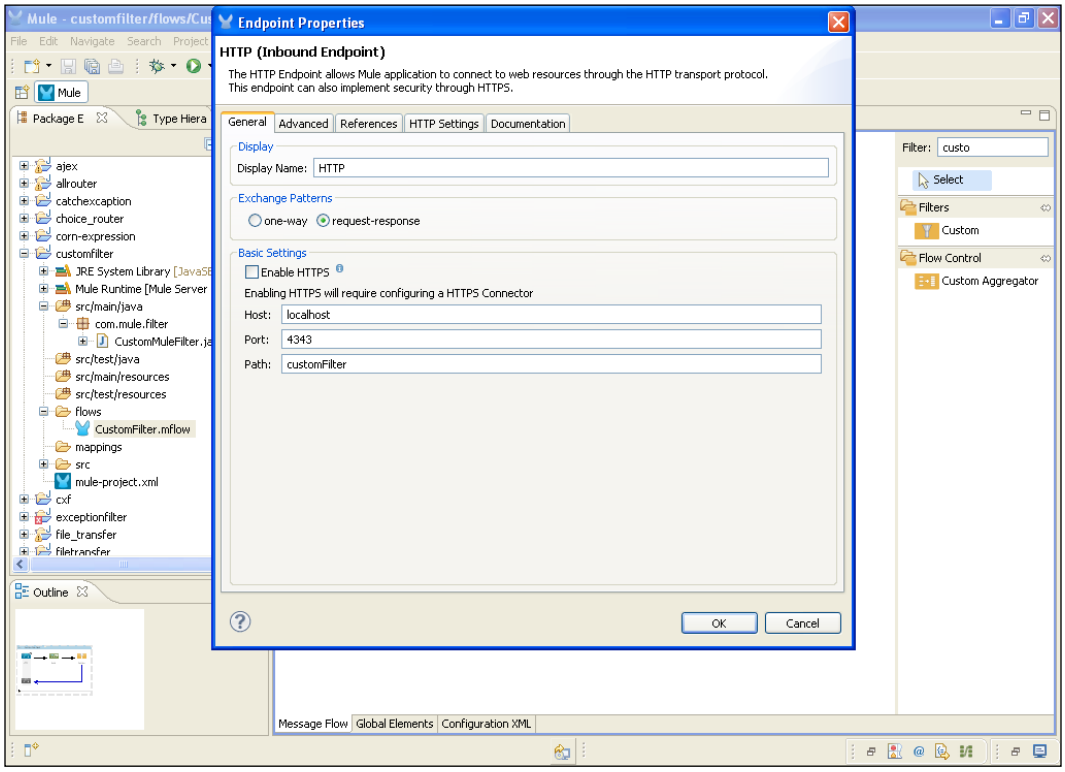
2. To create a new project, go to **File** | **New** | **Mule Project**. Enter the project name `Custom Filter`, and click on **Next** and then on **Finish**. Your new project is created now, so you have to start the implementation.



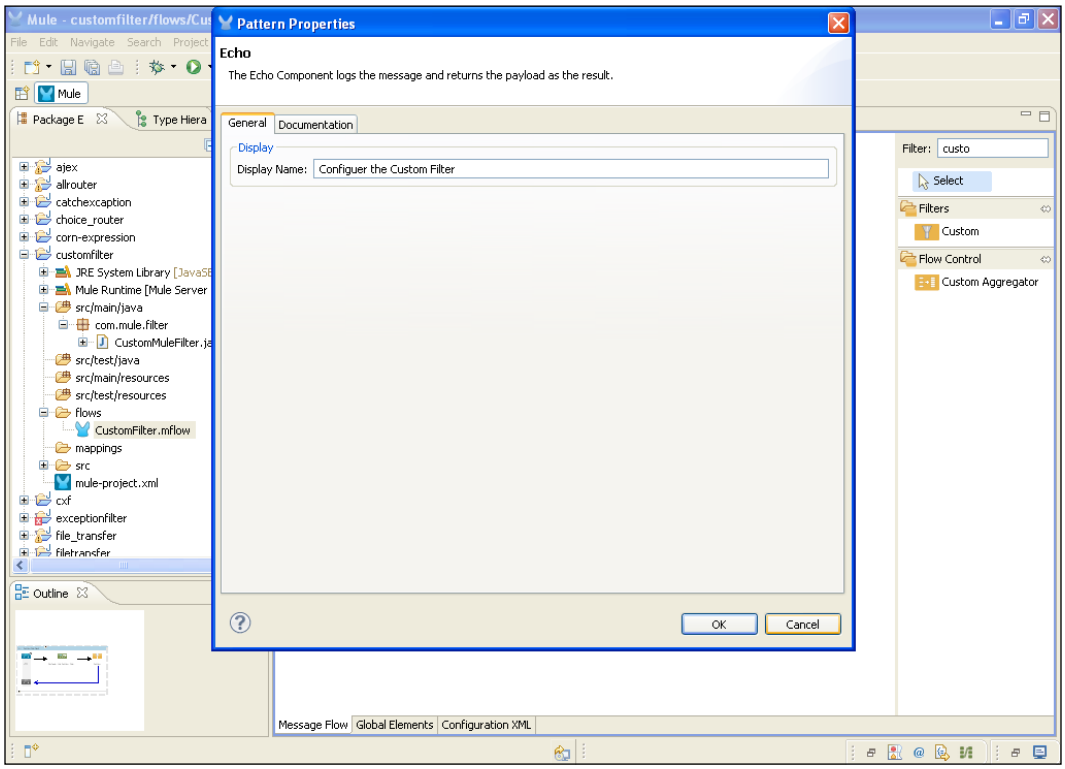
3. Go to `src/main/java`, right-click on it and go to **New** | **Class**, enter the package name `com.mule.filter` and the class name `CustomMuleFilter`, and click on the **Finish** button. In this class, we implement the `Filter` interface, and in this filter interface class, we override the `accept` method.



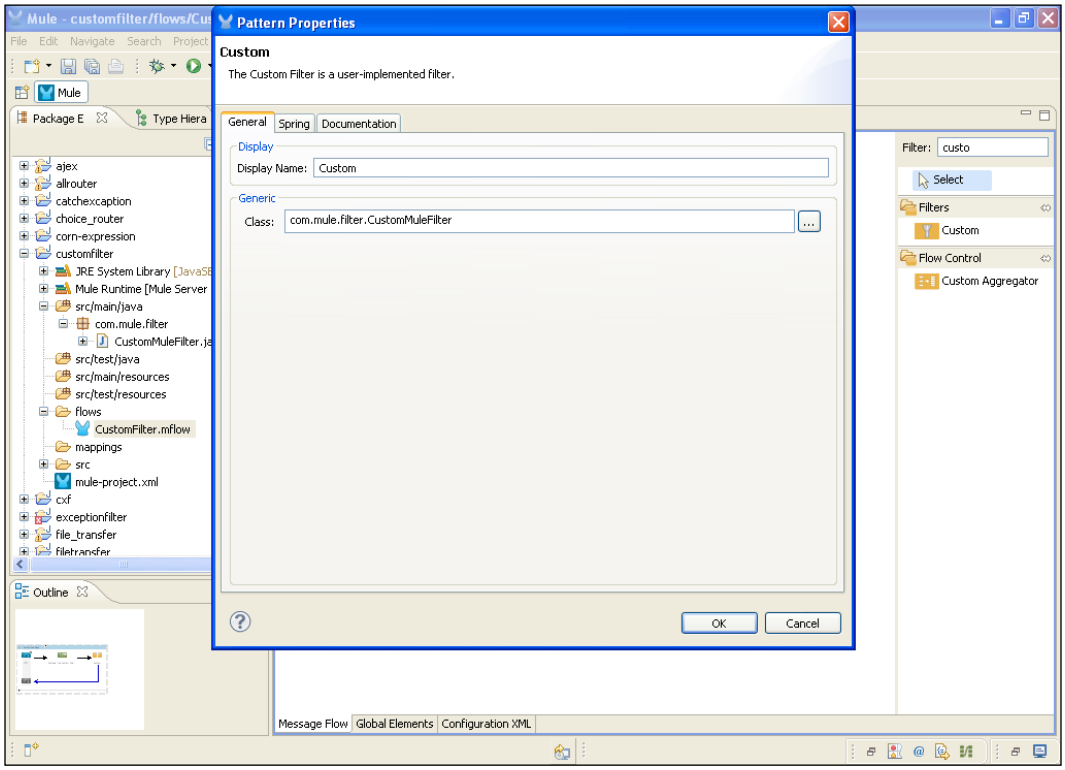
4. Go to the `CustomFilter.mflow` file and drag the **HTTP** Endpoint onto the canvas. Double-click on it to configure it. Enter the port number and the hostname, and click on the **OK** button. By doing this, you enter the pathname called `customFilter`.



5. To display the log on the console, drag the **Echo** component, drop it on the canvas, and configure it.



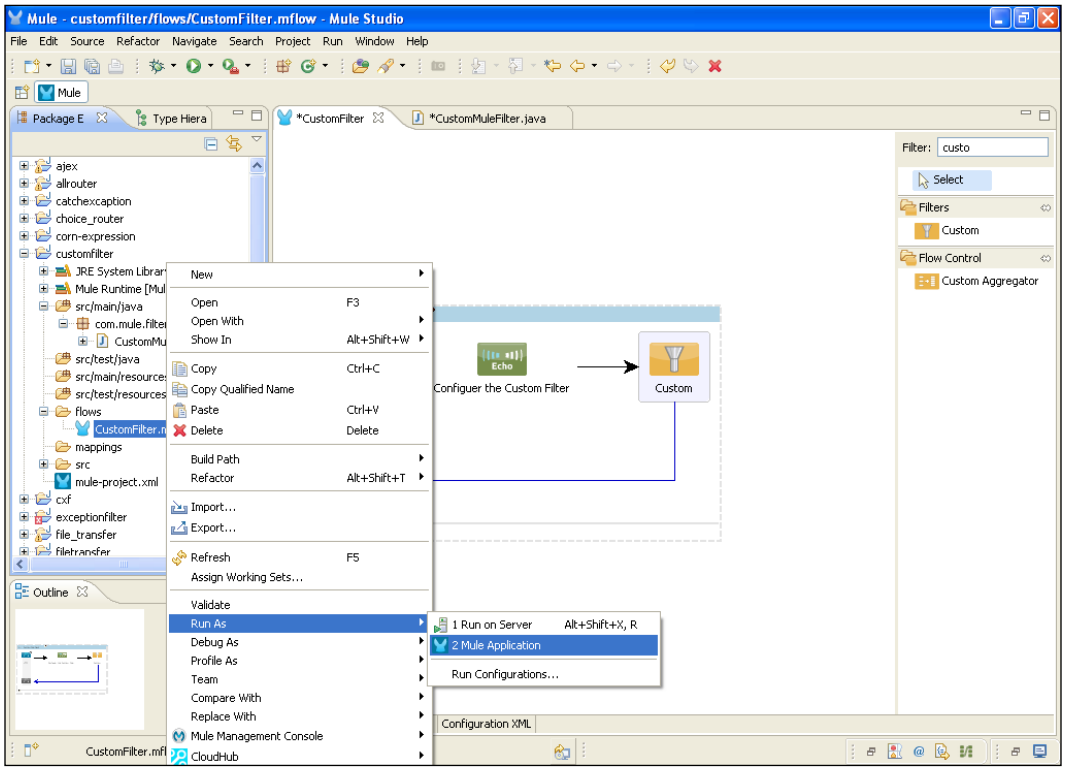
6. Drag the **Custom** filter from the palette and drop it on the canvas. Double-click and configure it. Import the class that was created before, as shown in the following screenshot:



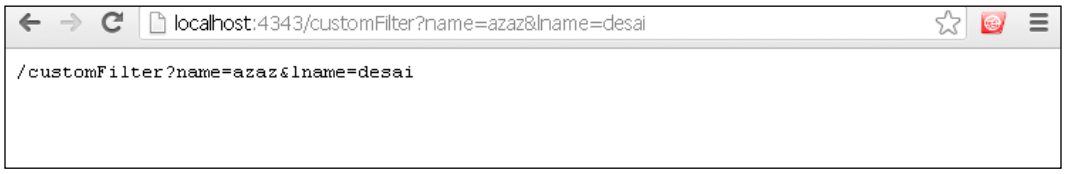
How it works...

In this section, you will see how to deploy the application.

1. To deploy the application code in the Mule server, go to **Run As** | **Mule Application,** and the Mule server will deploy your application.



2. Open the browser and paste the URL `http://localhost:4343/customFilter`—the URL that starts with `http://localhost:4343/customFilter?name=azaz&lname=desai`.



7
Handling Exceptions and Testing

In this chapter, we will cover different ways of handling exceptions and testing. You will learn how to do the following:

- Understanding Messaging Exception strategies
- Configuring the Choice Exception Strategy
- Configuring the Reference Exception Strategy
- Configuring the Rollback Exception Strategy
- Testing with JUnit in Mule ESB

Introduction

Mule errors are represented by **exceptions**; so when your Endpoint, Router, component, or any other processor fails, it throws an exception. When an exception is thrown, you need a way to handle it. In Mule there are different types of exception strategies. You will have a look at them in this chapter. There are two places where you could configure exception handling strategies: on the service and on the connector. The `src/test/` directory in every Mule ESB Maven project incorporates both unit and functional tests.

Understanding Messaging Exception strategies

Mule calls for the Messaging Exception Strategy whenever an exception is thrown in a flow. All exceptions are handled through the Messaging Exception Strategy. There are five types of Messaging Exception Strategies:

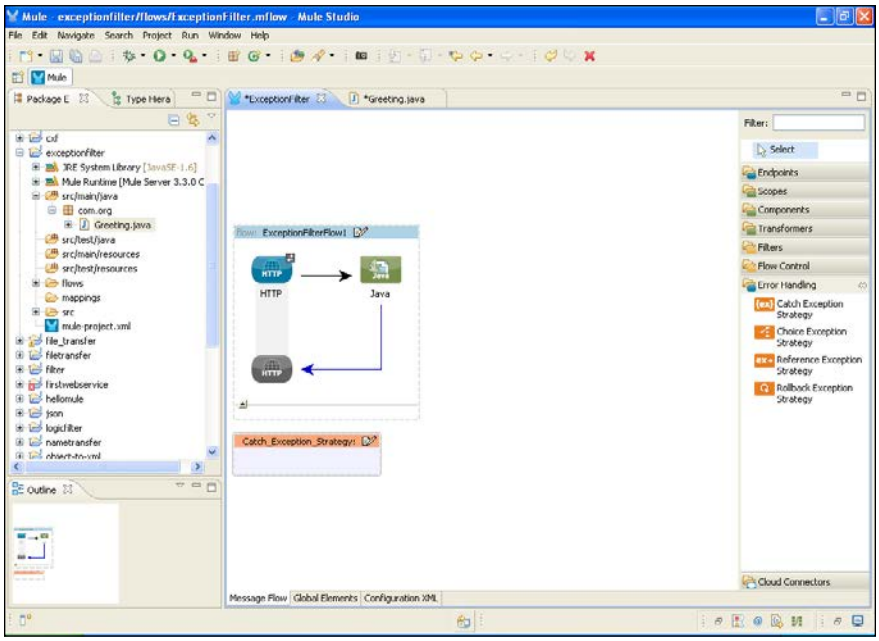
- Default Exception Strategy
- Catch Exception Strategy
- Rollback Exception Strategy
- Reference Exception Strategy
- Choice Exception Strategy

The Catch Exception Strategy

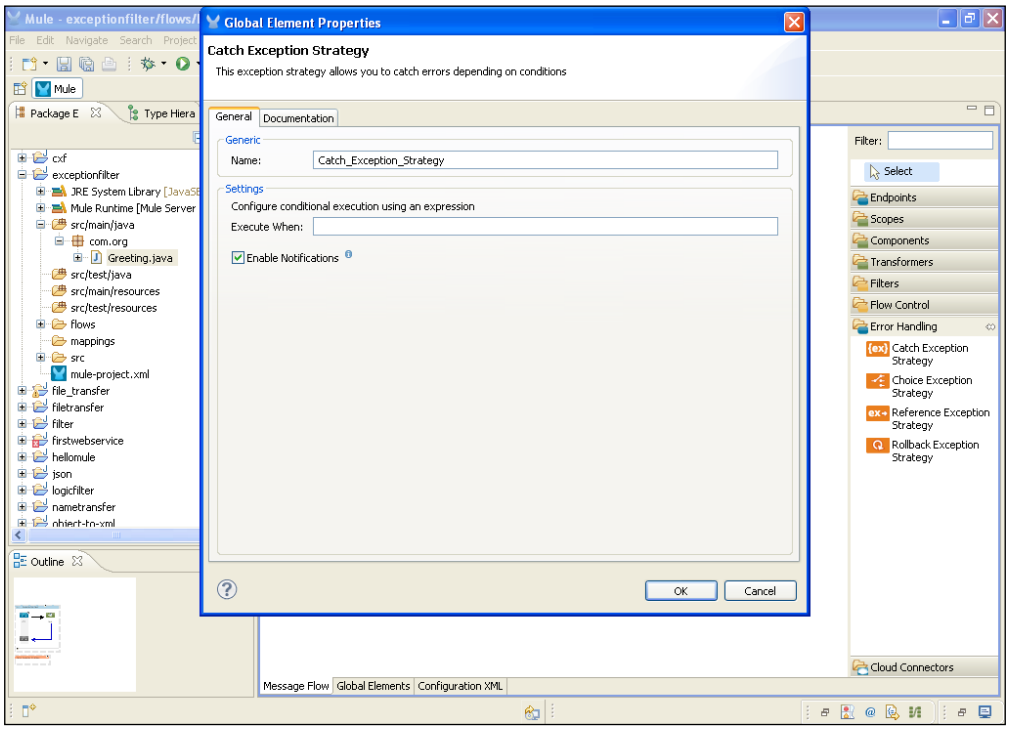
You can define a Catch Exception Strategy to customize the way Mule handles messages with errors. A Catch Exception Strategy catches all the exceptions thrown within flow and processes them. From the **Error Handling** palette group, drag **Catch Exception Strategy** and drop it onto the canvas.

Getting ready

In this section, you will see how to use the Catch Exception Strategy in Mule Studio.



Double-click on **Catch Exception Strategy** and configure it.



How to do it...

In this section, you will see how to configure the catch exception properties in Mule:

1. In the Catch Exception Strategy properties panel that appears on the screen, enter a name for your Catch Exception Strategy in the **Name:** field under the **General** tab.
2. Check the **Enable Notifications** checkbox to instruct Mule to send an exception notification to a registered listener, say, the Mule management console, whenever the Catch Exception Strategy accepts and handles an exception.
3. To enter an exception, enter information in the **Execute When:** field to indicate the kind of exception the catch exception handles when it resides within a Choice Exception Strategy.

Use case

HTTP Inbound Endpoint receives a message through the custom Java component and processes it through the flow. If the execution is successful, return a Java component message. If the execution fails then log the current message and return a Java component error response. You can define only one exception at a time.

Configuring the Choice Exception Strategy

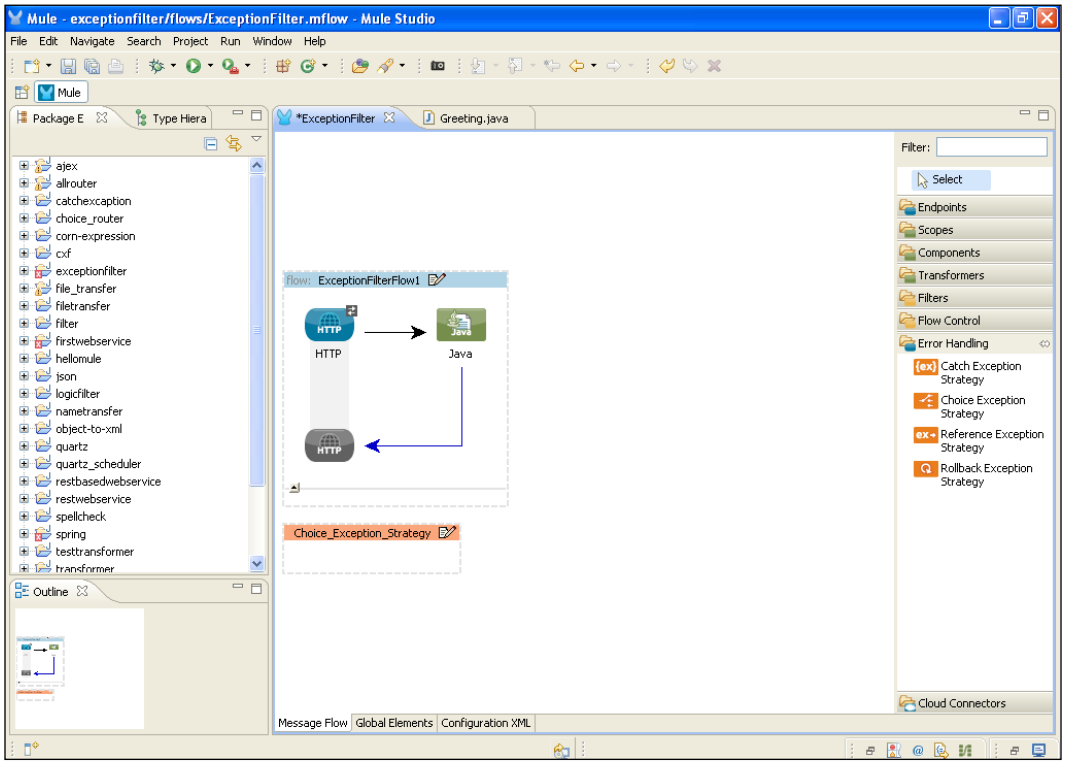
A Choice Exception Strategy catches all the exceptions thrown within its parent flow, examines the message content and exception type, and routes messages to the appropriate exception strategy for processing.

Getting ready

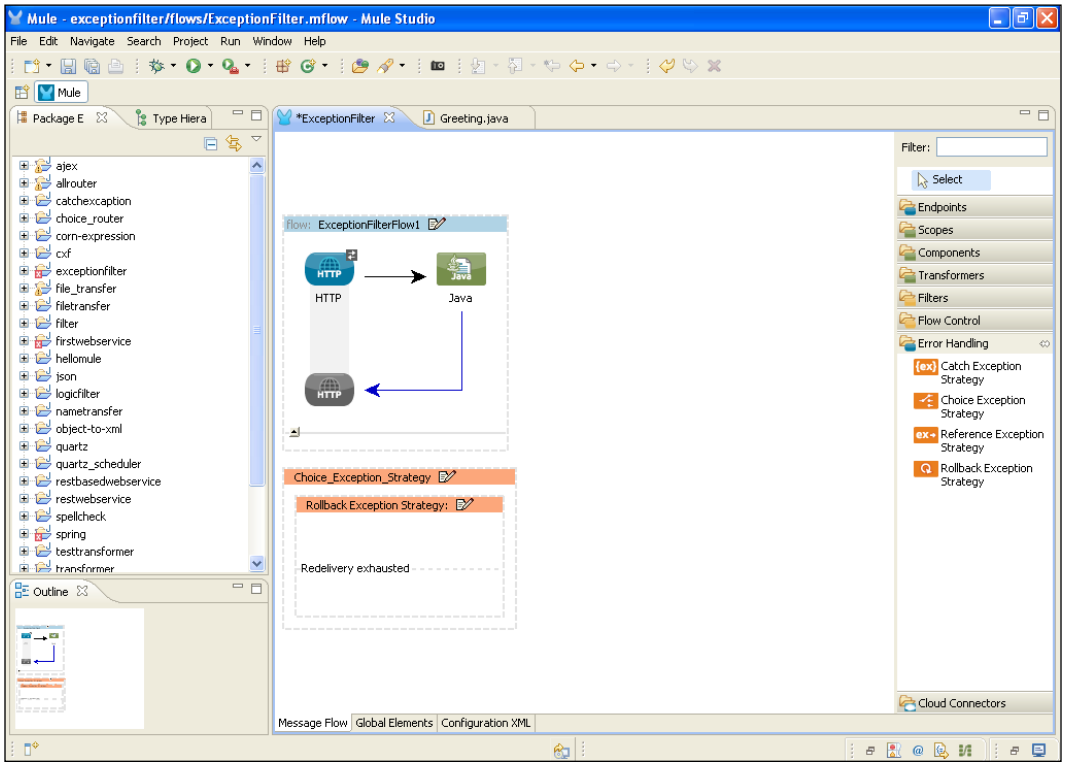
You can define a Choice Exception Strategy to customize the way Mule handles a message with an error based on the message's content the moment it throws an exception. Usually, you can define more than one exception strategy within a Choice Exception Strategy.

How to do it...

1. From the **Error Handling** palette group, drag **Choice Exception Strategy** and drop it onto the canvas. In the **Choice_Exception_Strategy** box that appears, double-click on the configuration icon in the header.



2. From the **Error Handling** palette group, drop another **Catch Exception Strategy** or **Rollback Exception Strategy** icon on the **Choice_Exception_Strategy** box. In Choice Exception Strategy you can define one or more exception strategies, but in Catch Exception Strategy you can define only one.



Use case

Use a Choice Exception Strategy to enable Mule to make decisions about how to handle each error that occurs in a flow. A Choice Exception Strategy can evaluate the exception type of each message that throws an exception in this flow and route them to one of three exception strategies:

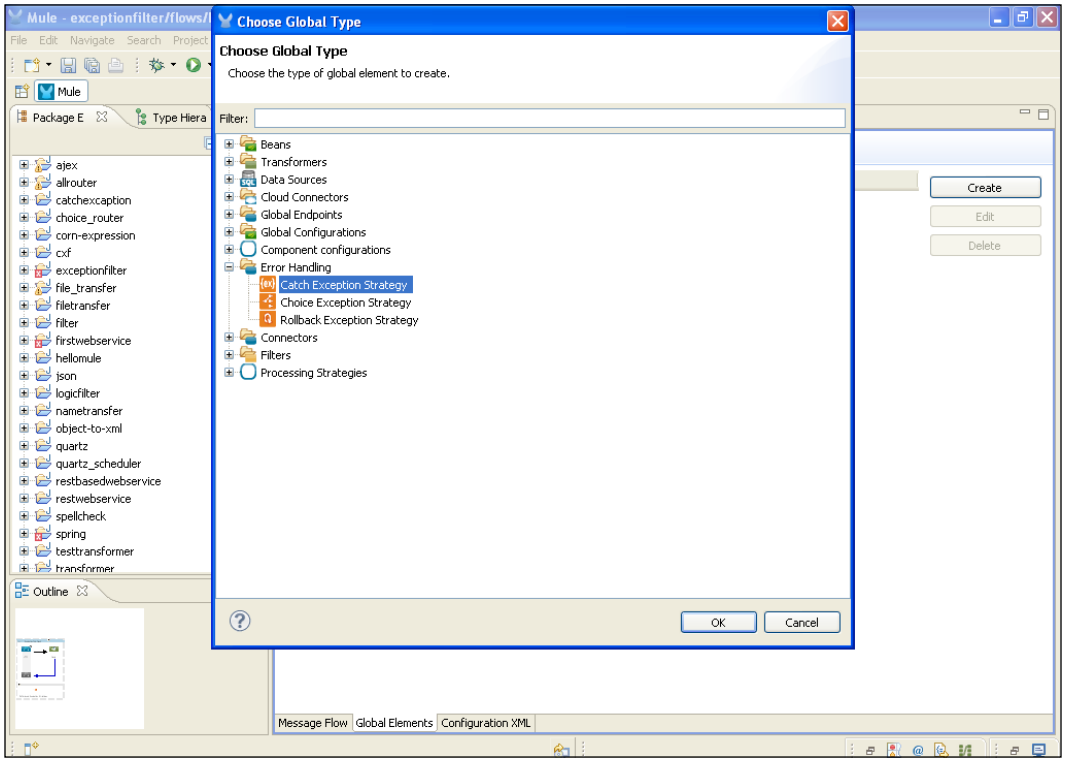
- A Catch Exception Strategy to process and discard all the "already processed" exceptions
- A second Catch Exception Strategy to process all the "validation exceptions" and send them to an invalid orders queue
- A Rollback Exception Strategy to roll back the order transaction in order to retry processing in the parent flow

Configuring the Reference Exception Strategy

Use a Reference Exception Strategy to teach a flow to employ the error handling performance defined by a global Rollback Exception Strategy. In other words, you must ask your flow to refer to a global exception strategy for instructions on how to handle errors.

Getting ready

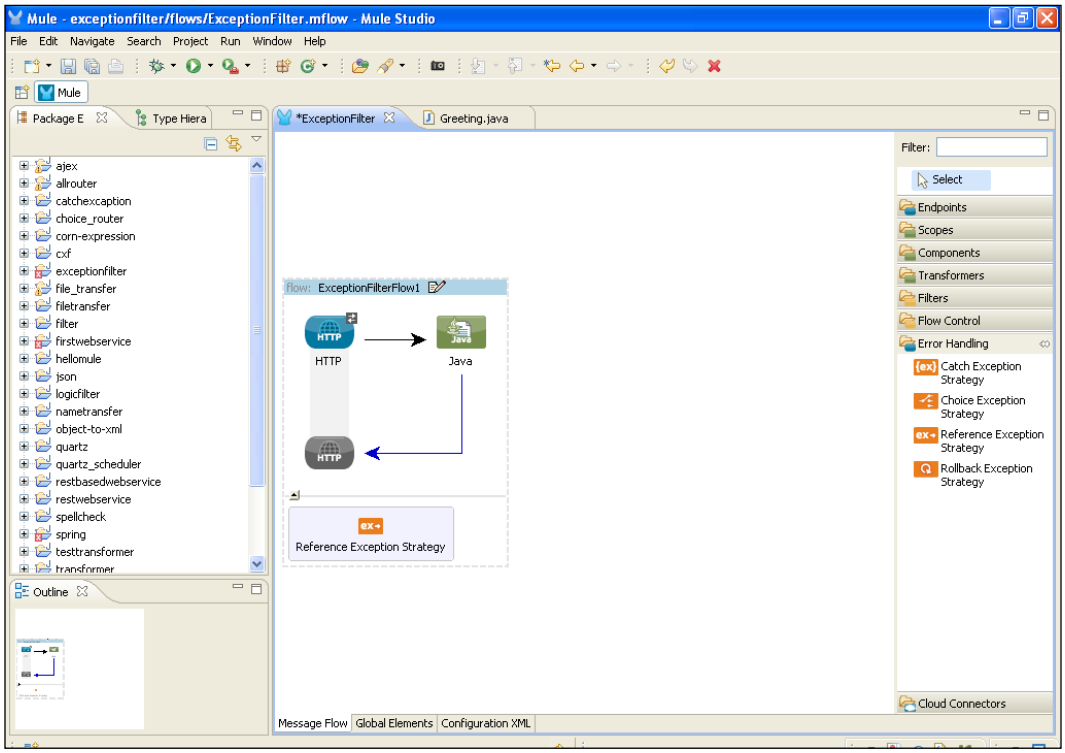
Reference Exception Strategies are created as global elements. You can create one or more strategies or re-use the one(s) in the flow throughout your Mule application. Create global exception strategies such as Catch and Rollback; choose the one which your exception strategies refers to. In the following screenshot you will see how to configure the global element:



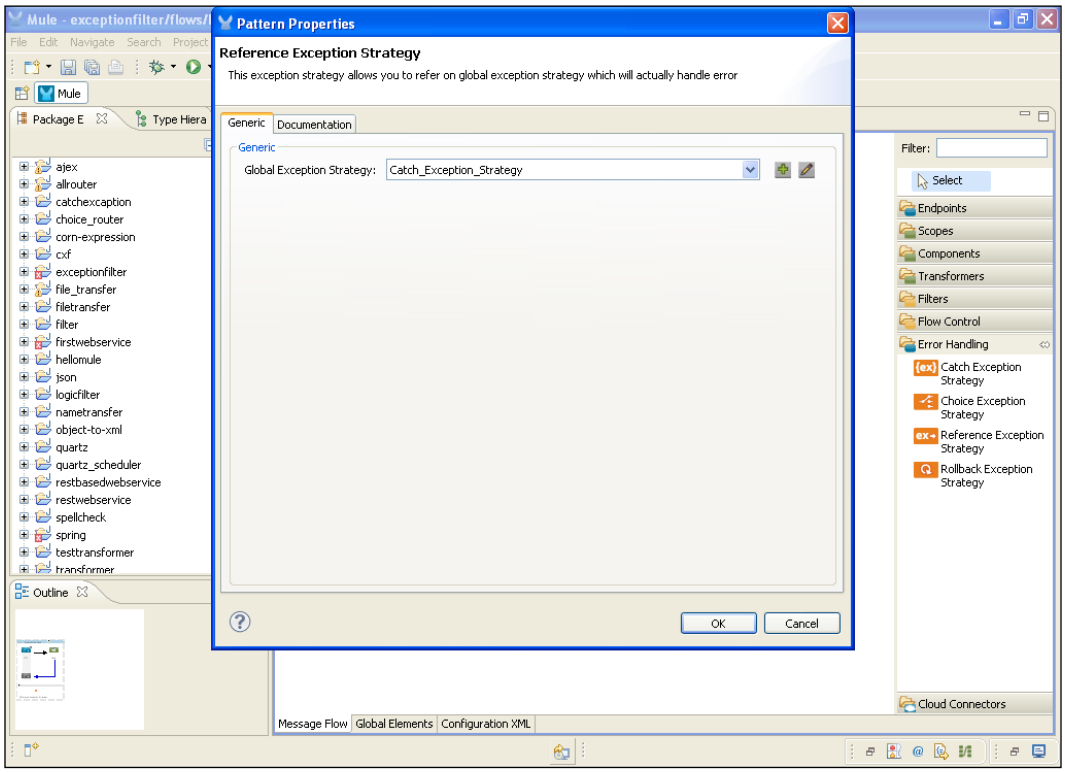
How to do it...

Perform the following steps:

1. From the **Error Handling** palette group, drag-and-drop the **Reference Exception Strategy** icon onto the canvas.



2. To configure the **Reference Exception Strategy**, double-click on **Components** and select a reference name that was created as a global element.



Configuring the Rollback Exception Strategy

You can define a Rollback Exception Strategy. This makes sure that a message that throws an exception in a flow is rolled back for reprocessing. Use a Rollback Exception Strategy when you cannot correct an error when it occurs in a flow.

Getting ready

The Rollback Exception Strategy is used to loop the executions infinitely until the conditional exceptional strategy becomes true. Let's say for example, we have a server with a count value = 0. Now if 50 requests come simultaneously and you want the server to handle not more than 50 instances, you will set a rollback exception that sets the count variable to 0 each time it reaches 50. This process is looped infinitely.

How to do it...

Usually, you can use this rollback exception in a transaction. If the transaction fails or if a message throws an exception while being processed, the Rollback Exception Strategy rolls the transactions back to where they exist in the flow. If the Inbound Endpoint is transactional, Mule delivers the message to the Inbound Endpoint of the parent flow again to reattempt processing.

How it works...

A Rollback Exception Strategy gives a message a few attempts to move through the flow successfully before the transaction is declared as "failed" and the message is consumed.

For example, suppose you have a flow that involves a bank transaction to deposit funds into an account. You configure a Rollback Exception Strategy to handle the errors that occur in this flow; when an error occurs during processing, a flow external bank account database is temporarily unavailable and the message throws an exception. The Rollback Exception Strategy catches the exception, and rolls the message back to the beginning of the flow to reattempt processing. During the second attempt at processing, the database is online again and the message successfully reaches the end of the flow.

Mule attempts to deliver the message again when your flow uses one of the following two types of transports: transactional or reliable.

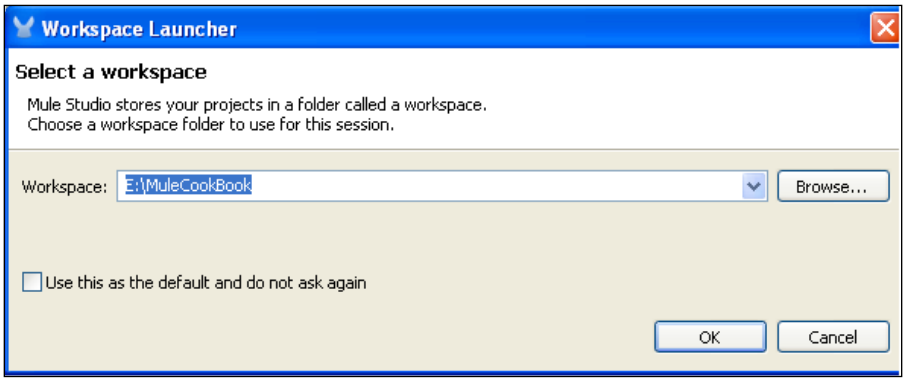
Testing with JUnit in Mule ESB

JUnit is a framework for implementing unit testing in Java. An open source Java testing framework is used to write and run repeatable automated tests. In JUnit 4.0, we do not need to extend from the JUnit framework to `TestCase`. Also, there is no need to use the prefix `Test` with the test method. You can run the test using `JUnit4TestAdapter` and the `@NAME` syntax, which already has been introduced. In the Dropbox integration example, you will learn about JUnit 4 and Selenium testing.

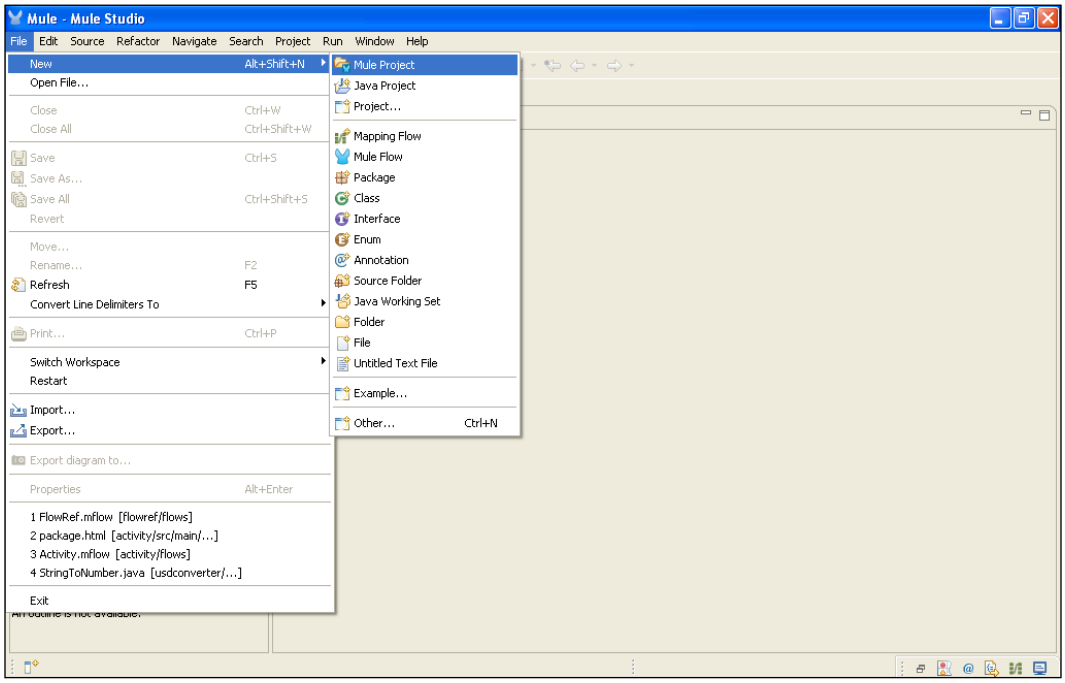
Getting ready

Create a new project on Mule to test with JUnit using the following steps:

1. Open Mule Studio and enter the workspace name as shown in the following screenshot:



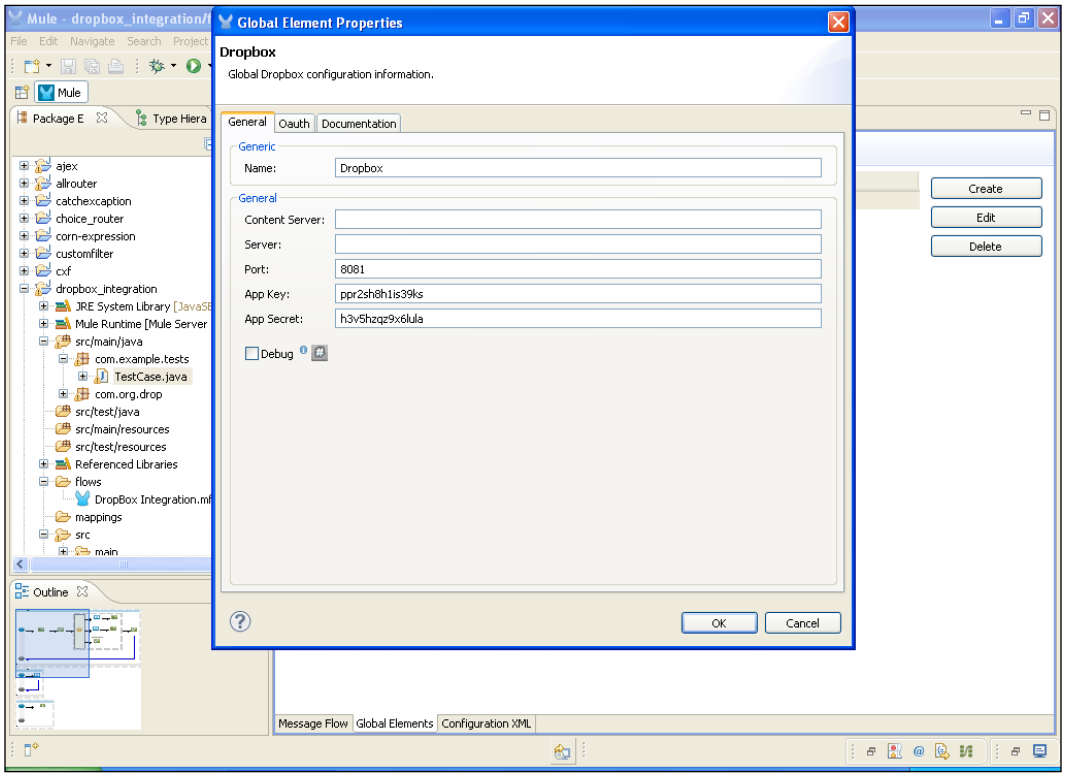
2. To create a new project, go to **File** | **New** | **Mule Project**. Enter the name of the project, `DropBox_Integration`, and click on **Next** and then on **Finish**. Your new project is created and you can now start the implementation.



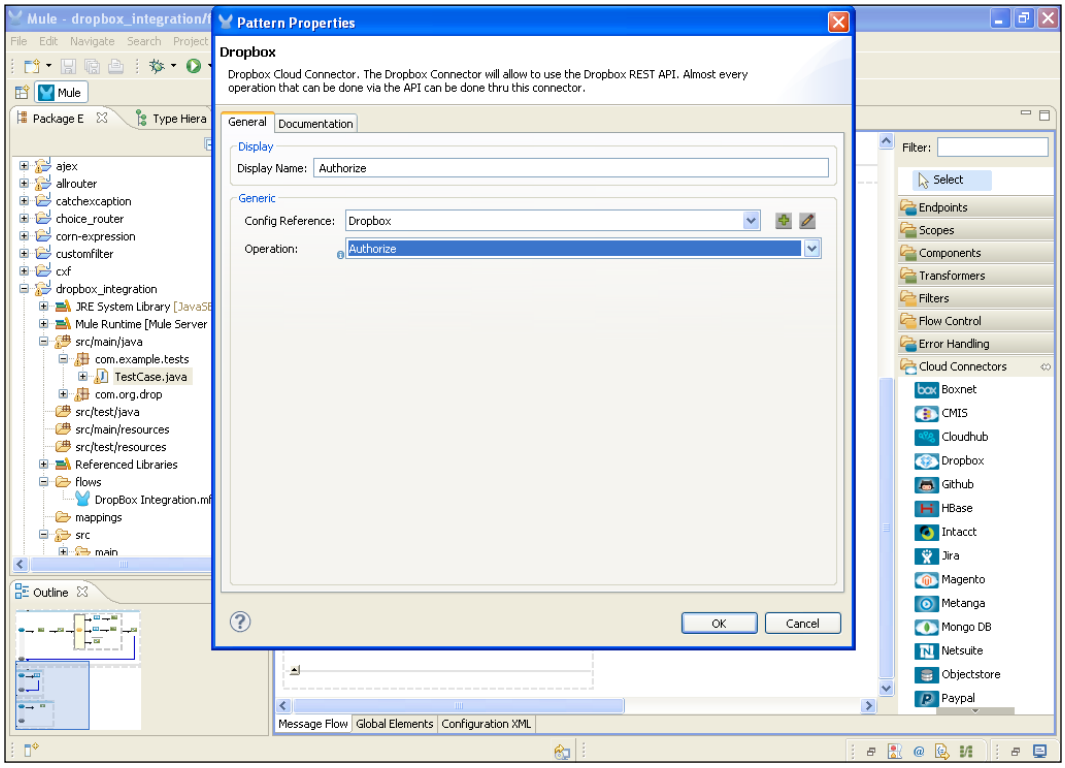
How to do it...

In this section, you will see how to configure the Dropbox connector in Mule Studio.

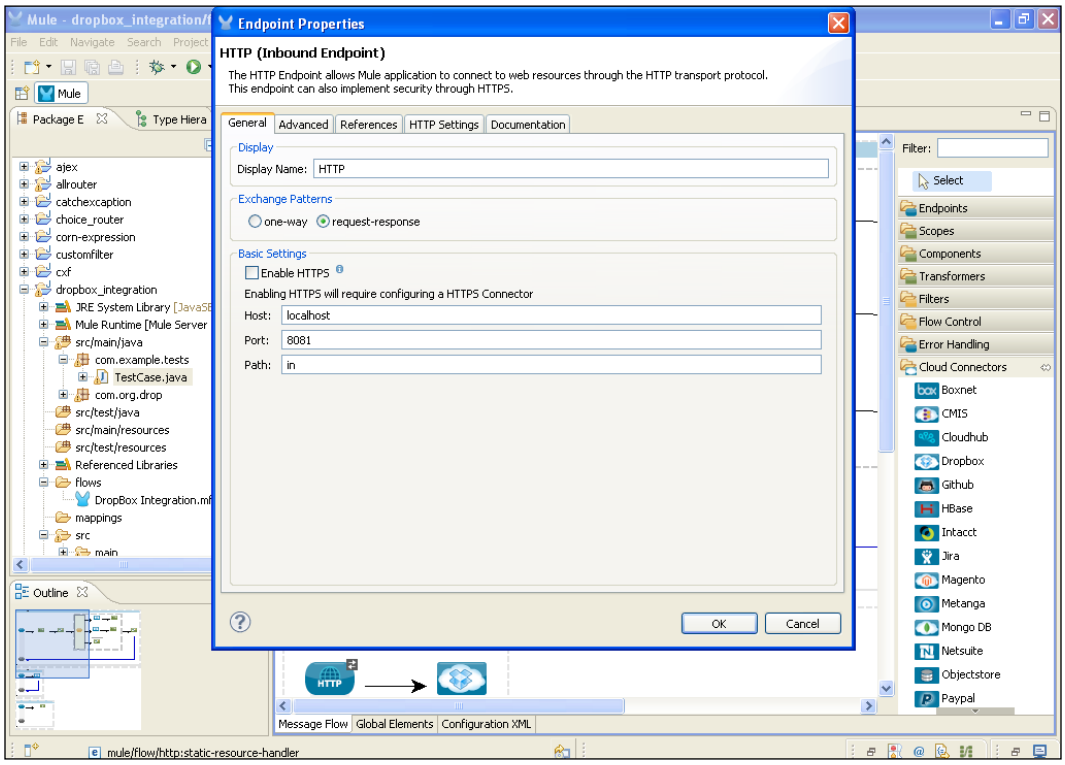
1. Navigate to the `dropbox_Integration.mflow` file in the **Global Elements** tab and click on the **Create** button. Go to **Cloud Connector** | **Dropbox**. Here you need to enter the API key or the secret key. You also have to generate this key on the Dropbox site.



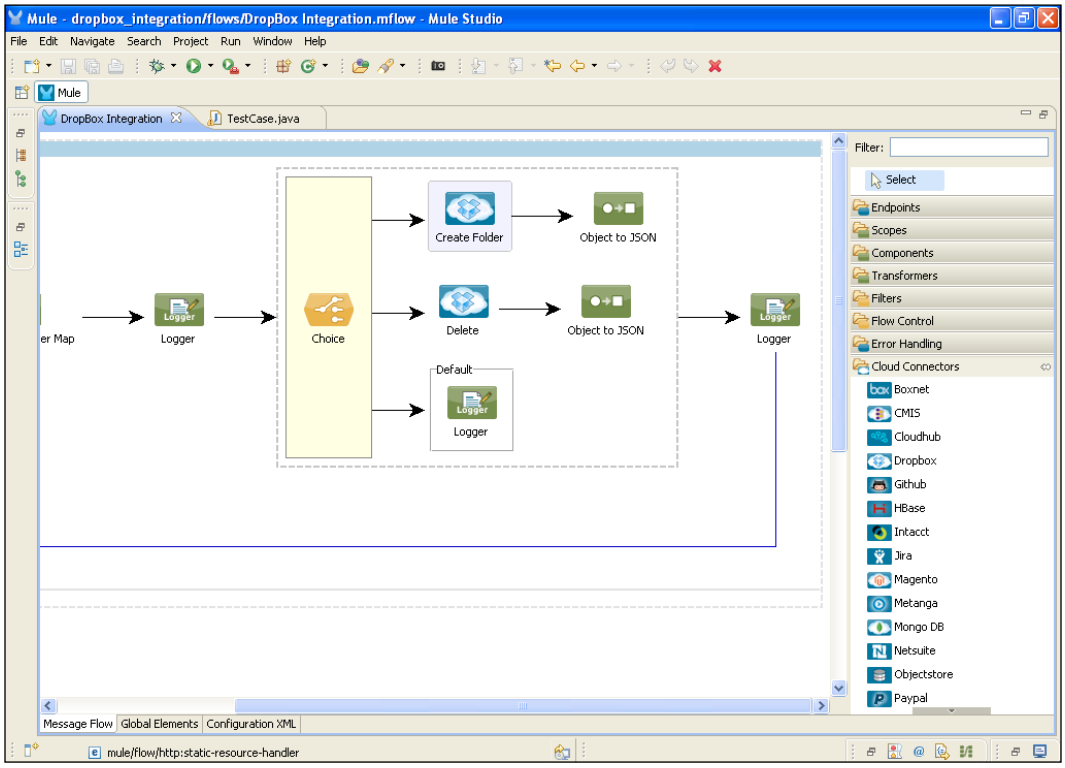
2. Click on the **Message Flow** tab and drag the **HTTP** Endpoint and the **Dropbox** Cloud Connector onto the canvas. Here you need to select the reference name, which was created earlier, and select the authorized operation.



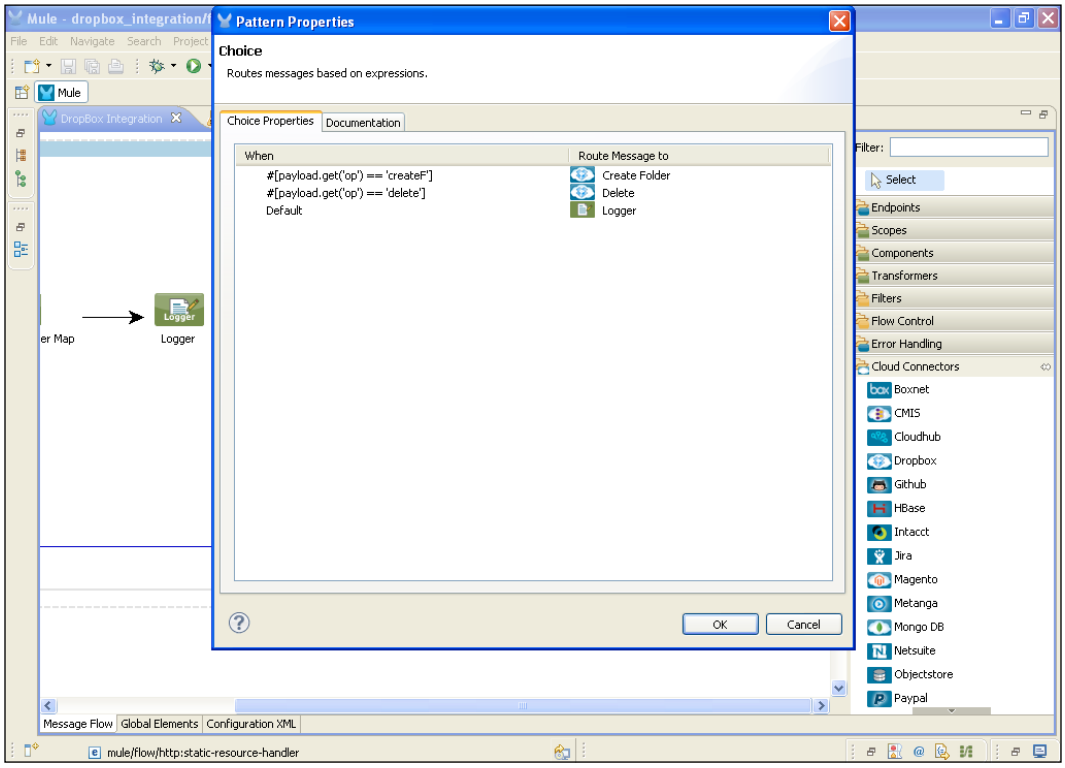
3. Now drag the **HTTP** Endpoint on the canvas and configure it.



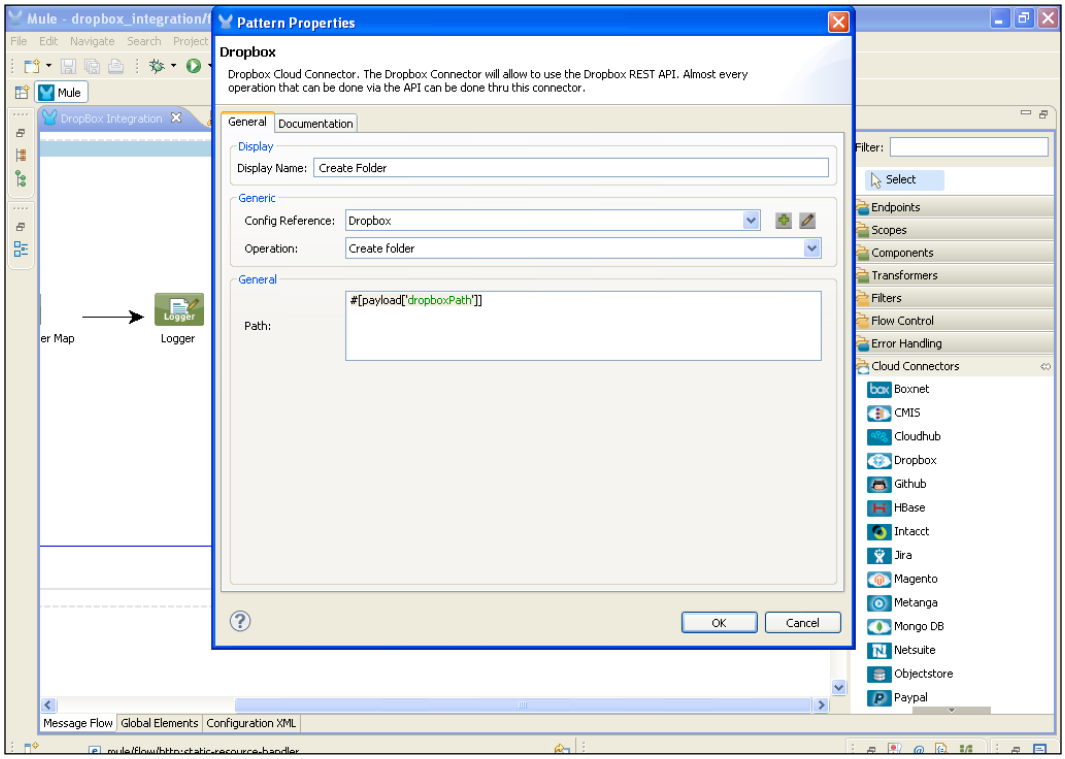
4. Drag the **Choice** Router and two **Dropbox** connectors onto the canvas. You are creating two operations now: one is for creating a folder and another is for deleting it.



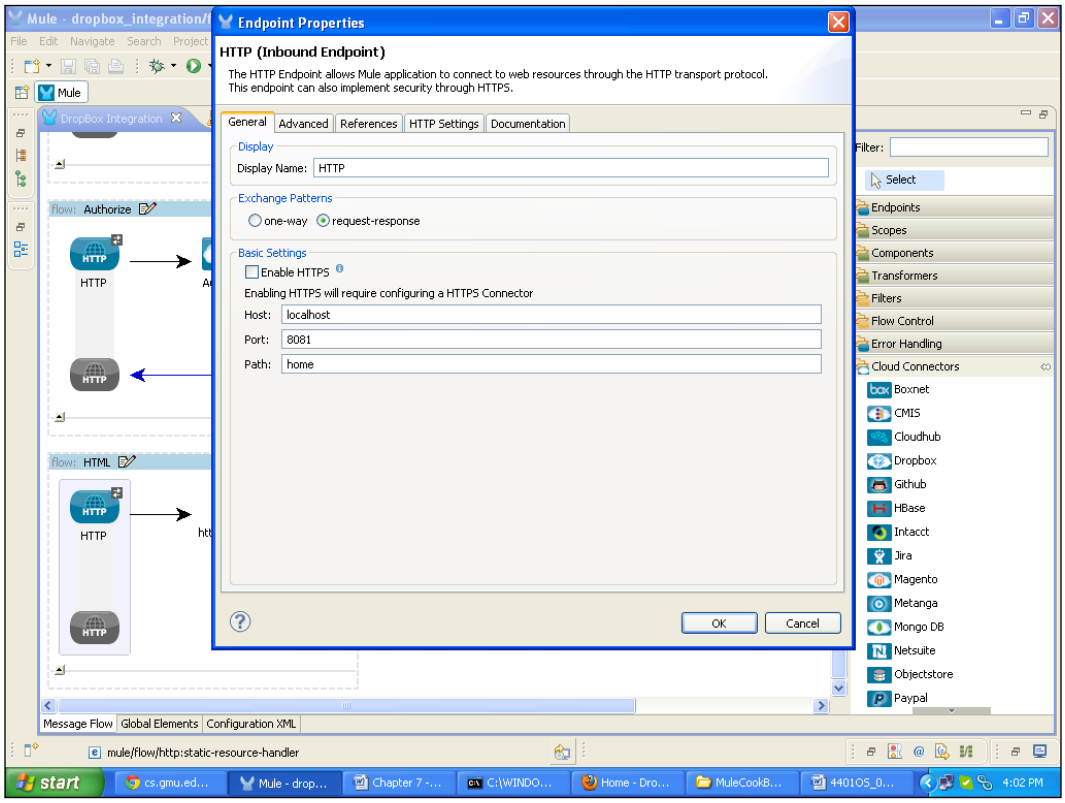
5. Double-click on the **Choice** Router to configure it. Here you can assign the following condition: if operation equals to `"createF"`, it will create a folder otherwise it will be deleted.



6. To configure the Cloud Connector, double-click on the **Create Folder** connector. Here you can select operation create folder and write an expression `#[payload['dropboxPath']]`. In the same way, you can create or delete operations on your own.



7. To create a third flow, drag the **HTTP** Endpoint onto the canvas and configure it. After this, you have to add static resources handler manually as follows: `<http:static-resource-handler resourceBase="${app.home}/docroot" defaultFile="home.html"></http:static-resource-handler>`.



8. Create a `docroot` folder inside `src/main/app`. Create a `Home.html` file inside the `docroot` folder, where `index.css` is stored.

   The `home.html` file looks like the following:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<link rel="stylesheet" type="text/css" href="index.css" />
<meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
```

```
<title>Insert title here</title>
</head>
<body>

  <script type="text/javascript">
  function processElements ( elements, style){

    for ( var i = 0; i < elements.length; i++){
      elements[i].style.display=style;
    }
  }

  function updateOptions(value){

  var hideElements = document.getElementsByClassName('hidden');
  var showElements = document.getElementsByClassName(value);

  processElements(hideElements, 'none');
  processElements(showElements, 'block')

  }
  </script>
  <!--onsubmit="this.action=document.getElementById('op').
options[document.getElementById('op').selectedIndex].value;"-->
  <form action=/in method="post">
    Welcome to AttuneInfocom!!! <br /> <br />
    Operation: <select id="op" name="op"
onchange="updateOptions(this.options[this.selectedIndex].value);">
          <option value="selectoption">--Select Option--</option>
          <option value="createF">Create Folder</option>
          <option value="delete">Delete</option>
        </select><br /><br />

    <div class="hidden upFile createF delete downFile
list getLink" id="dropboxPath">Path:<input type="text"
name="dropboxPath" /></div>
    <input type="submit" value="Submit"/>
  </form>
</body>
</html>
```

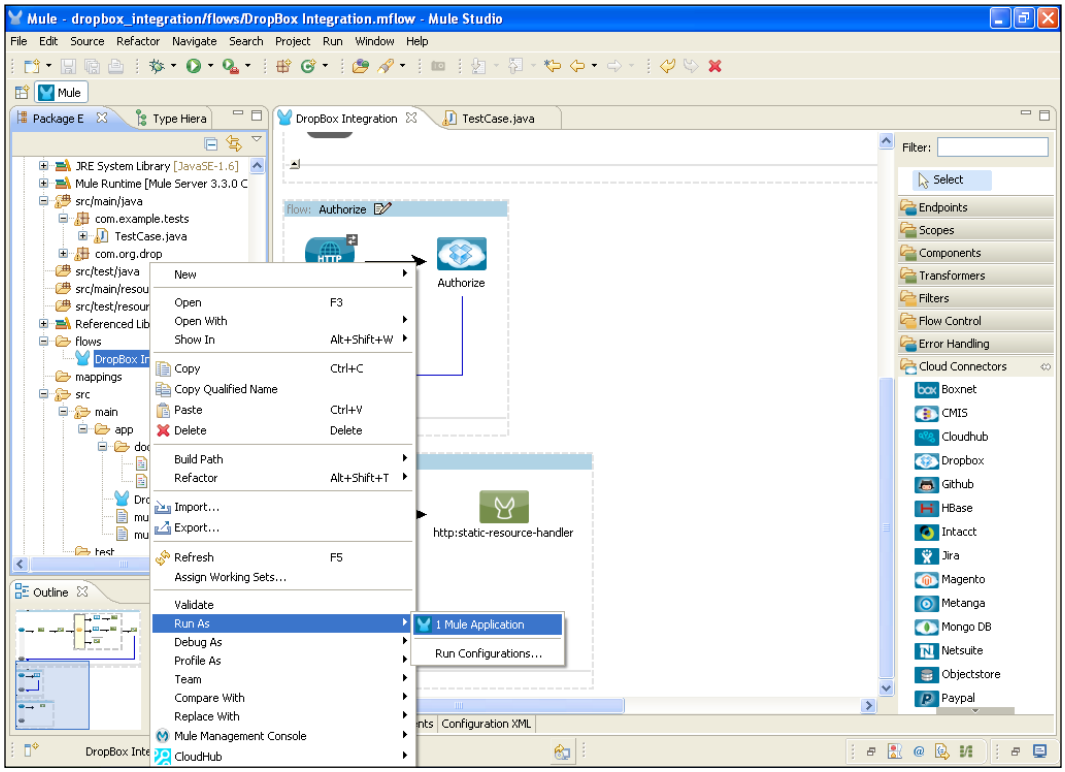
The `index.css` file has the following code snippet in it:

```
form {
background: -webkit-gradient(linear, bottom, left 175px,
from(#CCCCCC), to(#EEEEEE));
background: -moz-linear-gradient(bottom, #CCCCCC, #EEEEEE 175px);
margin:auto;
position:relative;
width:350px;
height:350px;
font-family: Tahoma, Geneva, sans-serif;
font-size: 14px;
font-style: italic;
line-height: 24px;
font-weight: bold;
color: #09C;
text-decoration: none;
-webkit-border-radius: 10px;
-moz-border-radius: 10px;
border-radius: 10px;
padding:10px;
border: 1px solid #999;
border: inset 1px solid #333;
-webkit-box-shadow: 0px 0px 8px rgba(0, 0, 0, 0.3);
-moz-box-shadow: 0px 0px 8px rgba(0, 0, 0, 0.3);
box-shadow: 0px 0px 8px rgba(0, 0, 0, 0.3);
}

textarea#feedback {
width:375px;
height:150px;
}
textarea.message {
display:block;
}
input.button {
width:100px;
position:absolute;
right:20px;
bottom:20px;
```

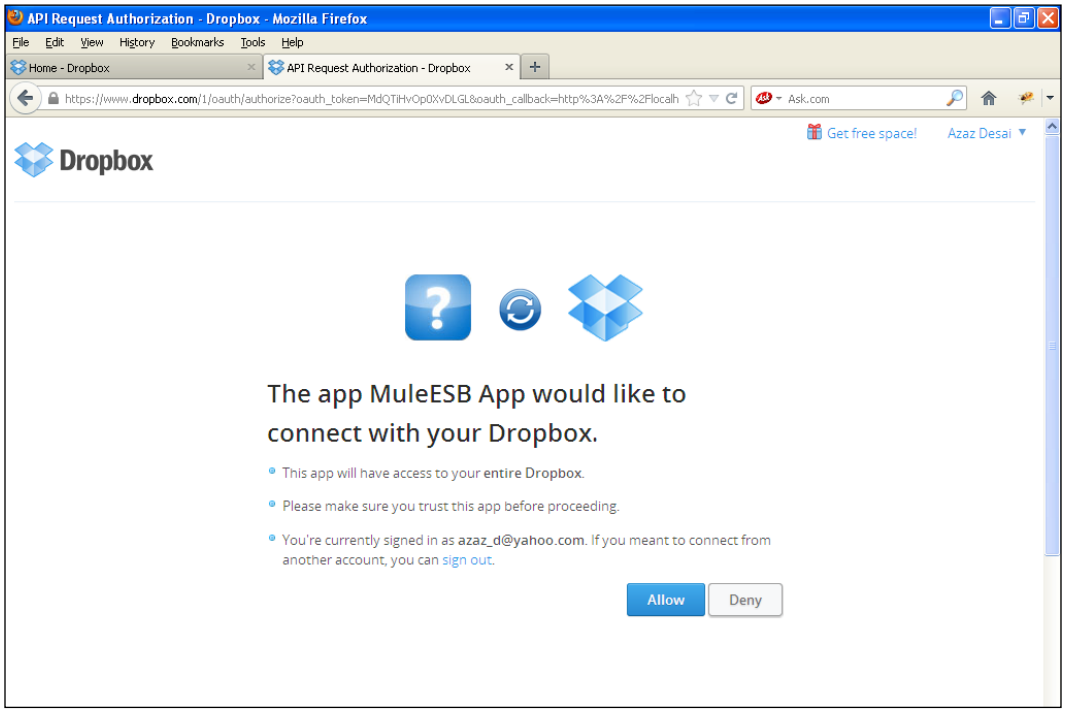
```
background:#09C;
color:#fff;
font-family: Tahoma, Geneva, sans-serif;
height:30px;
-webkit-border-radius: 15px;
-moz-border-radius: 15px;
border-radius: 15px;
border: 1p solid #999;
}
input.button:hover {
background:#fff;
color:#09C;
}
textarea:focus, input:focus {
border: 1px solid #09C;
}

img,a {
display:none;
}

#obj {
display:none;
}

.hidden {
display:none;
}
```

9. Here you have worked with three different flows. First, you deploy `localhost:8081/auth`, and authorize the Dropbox application. Then you can call `localhost:8081/home`.

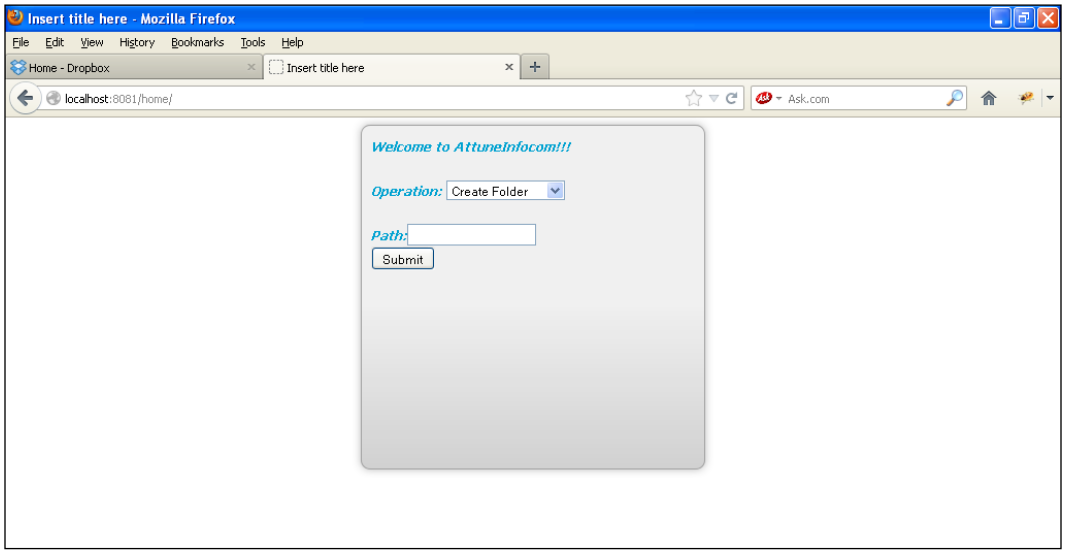
10. To deploy the application code in the Mule server, go to **Run As** | **Mule Application**; the Mule server will deploy your application.



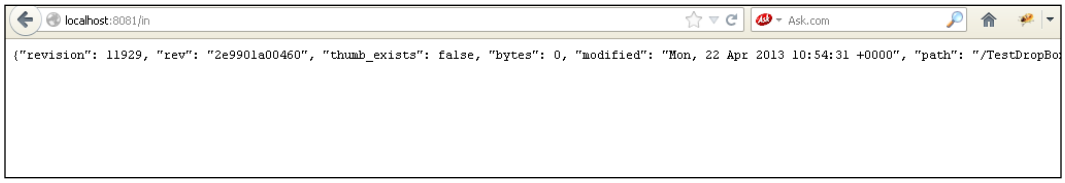
11. Open the browser and enter `localhost:8081/auth`. Once you enter the URL in the browser, it will redirect you to the Dropbox site and open a screen that looks like the following screenshot. Now click on the **Allow** button.



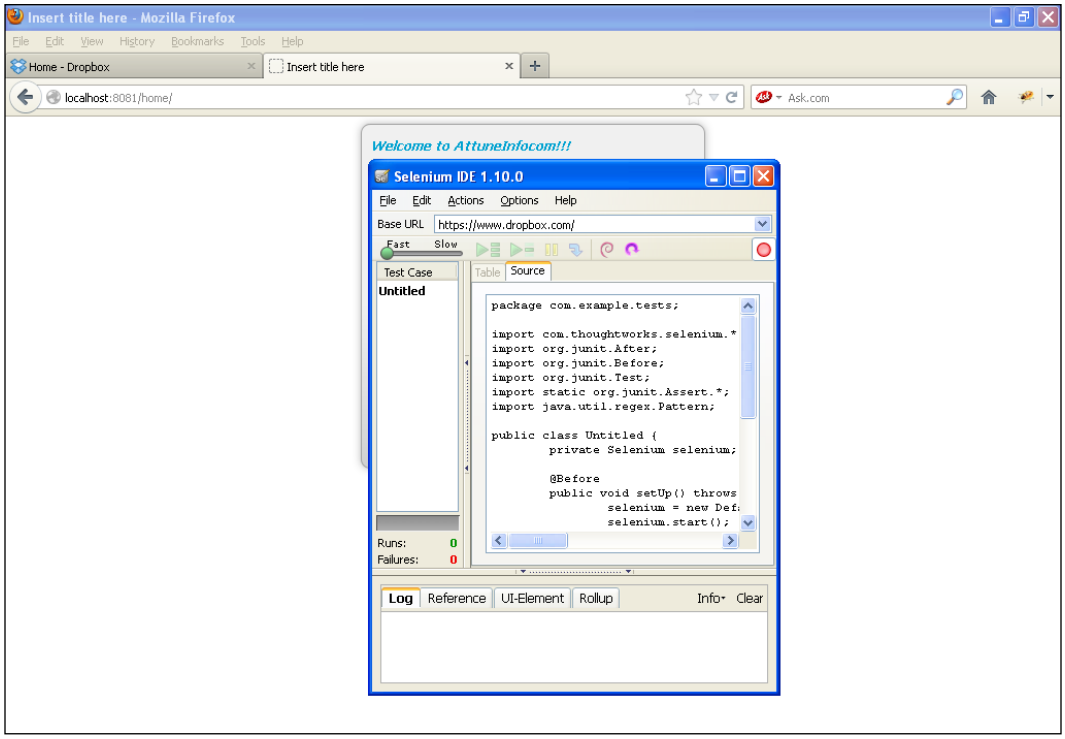
12. Enter the URL `http://localhost:8081/home`. Here, you can create a folder using the selected operation shown in the following screenshot:



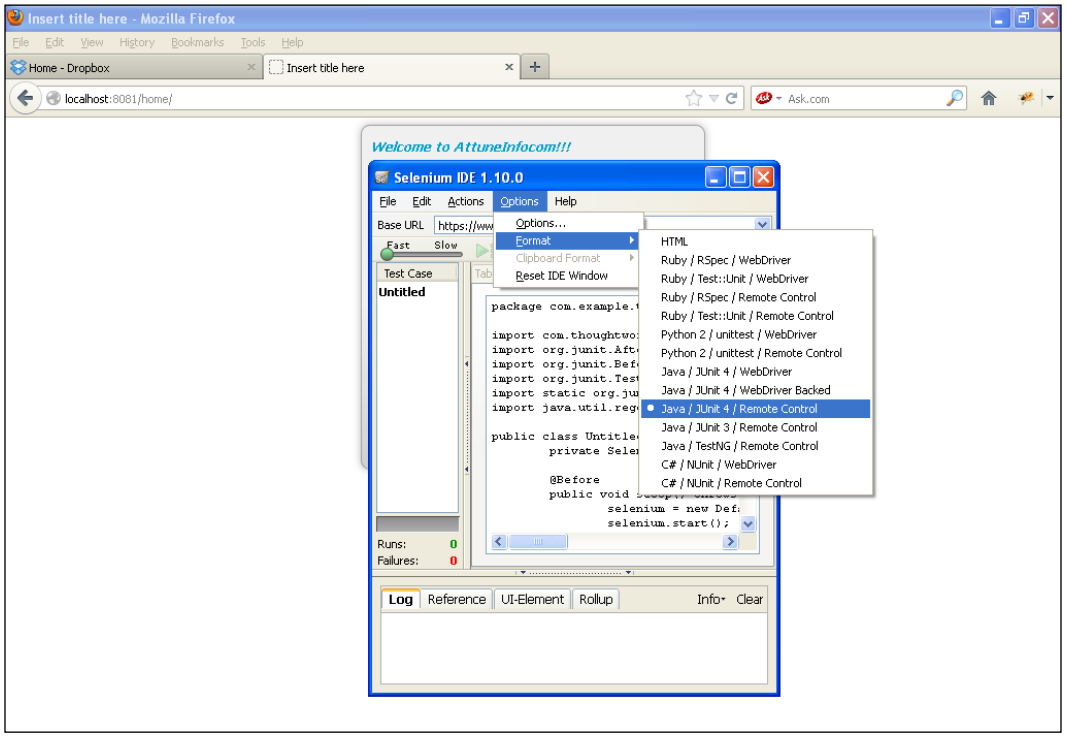
13. Once you click on the **Submit** button, you will see an output similar to the following screenshot:



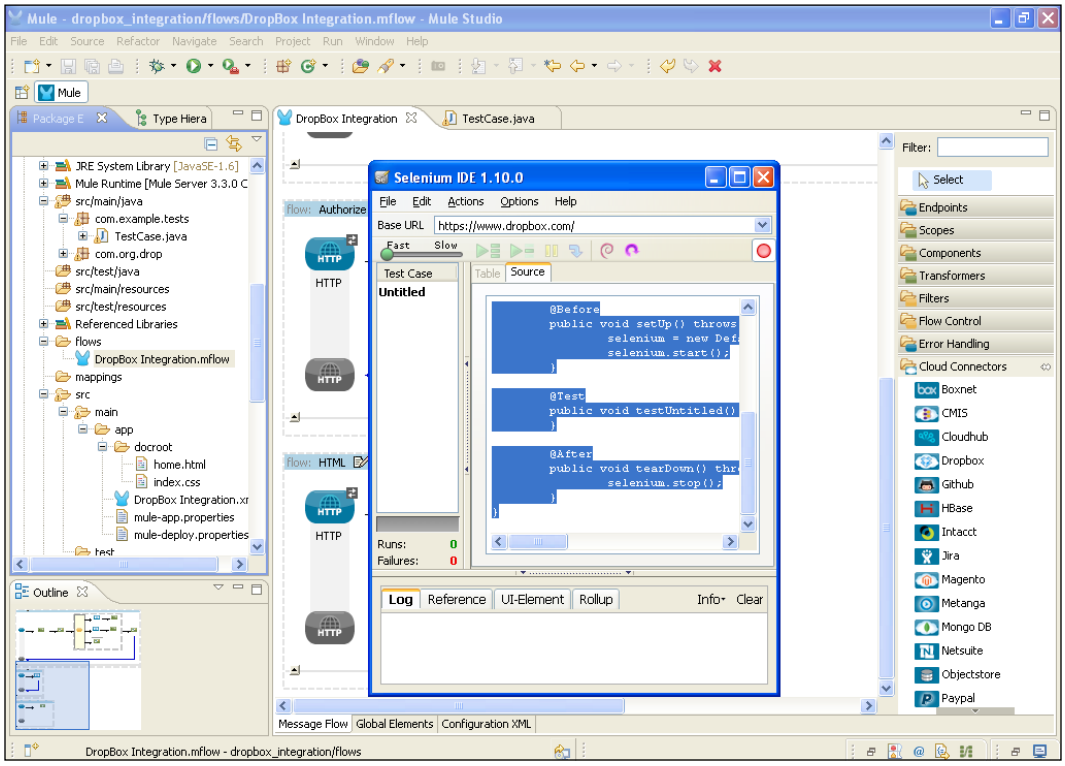
14. Open your Dropbox account. You will see that a folder is created with the name `TestDropBox`.
15. Similarly, you can run the deployment for JUnit 4. For that, you will have to download the Selenium IDE testing plugin on Firefox.
16. Once you download the testing plugin, a part of the IDE is completed. Restart Firefox. After restarting, go to **Tools** and select **Selenium IDE**.
17. Enter `localhost:8081/home`, select the operation, and click on the **Submit** button. Then stop recording in Selenium IDE.



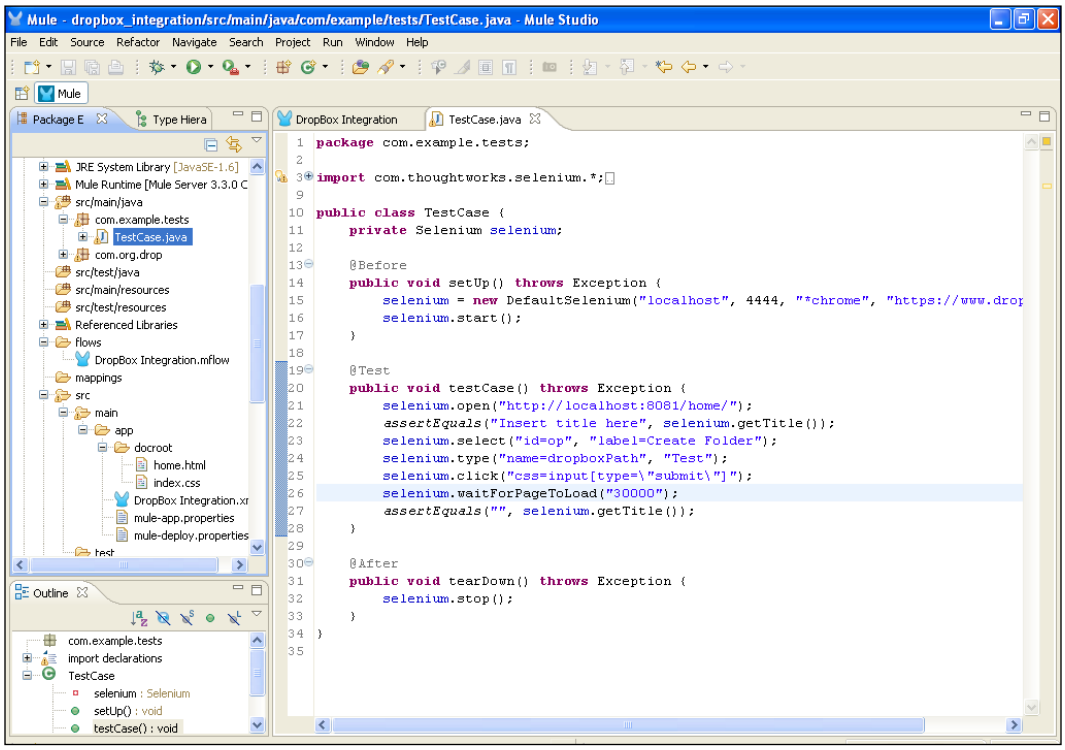
18. Now you have to convert the JUnit test case into an appropriate format. Go to **Options | Format | Java / JUnit 4 / Remote Control**.



19. You will see that the JUnit test case is created. Copy this test case and paste it in your application.



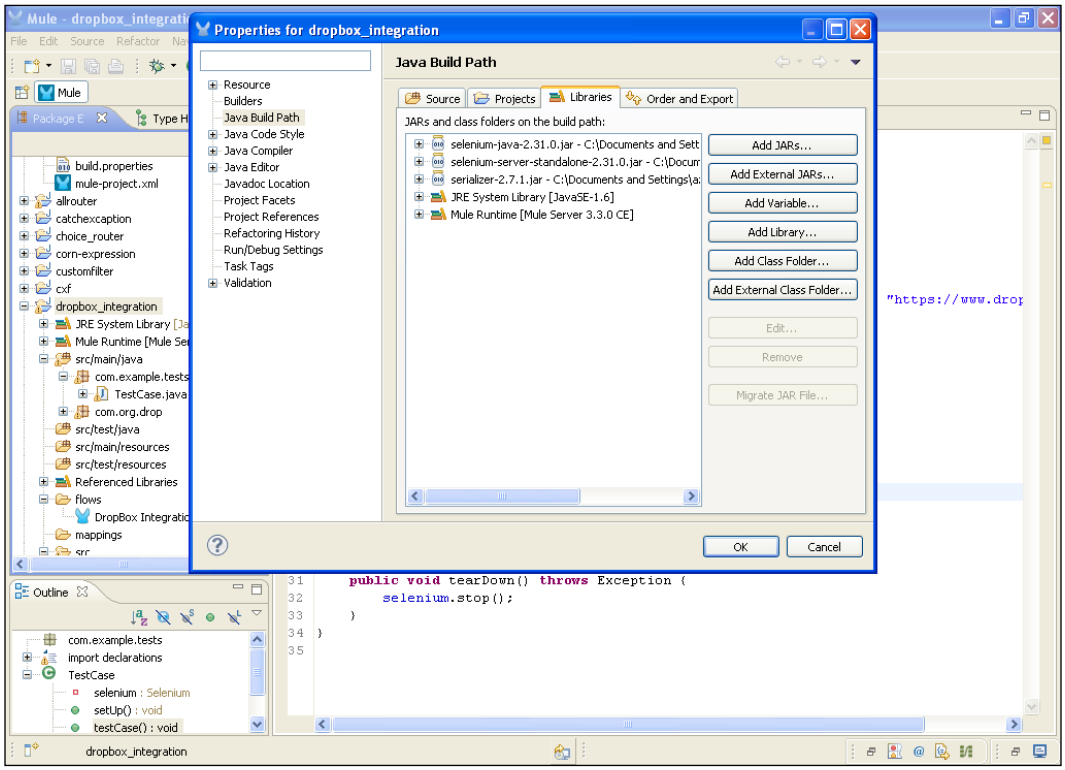
You can create such a test case in your application as well:



How it works...

In this section, you will see how to deploy the JUnit test case in Mule Studio.

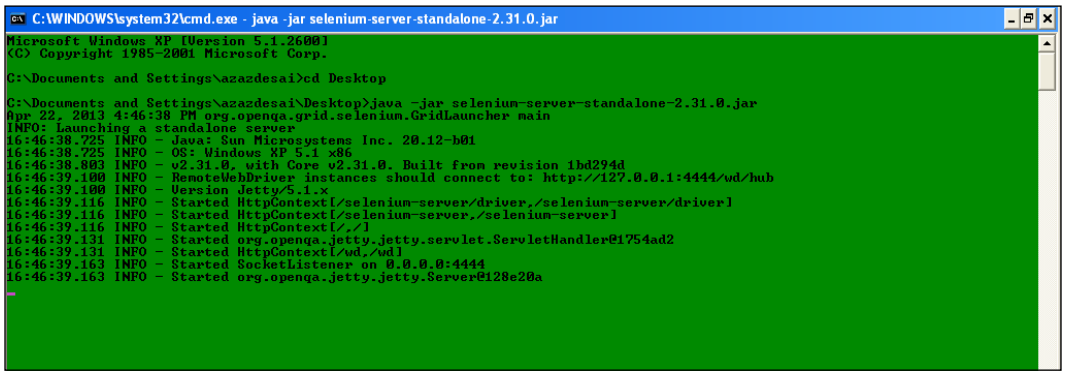
1. First, you have to import three Selenium JAR files: `serializer-2.7.1`, `selenium-java-2.31.0`, and `selenium-server-standalone-2.31.0`.



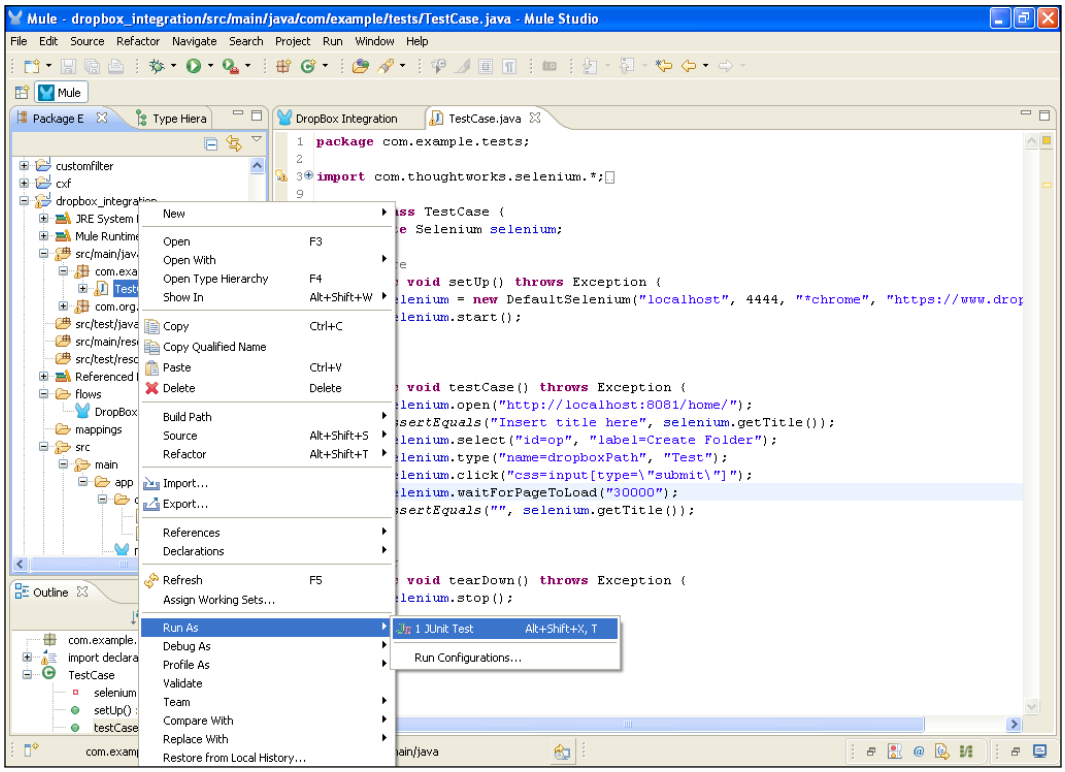
2. Go to the command prompt. Run the `selenium-server-standalone-2.31.0.jar` file using the following command:

```
java -jar selenium-server-standalone-2.31.0.jar
```

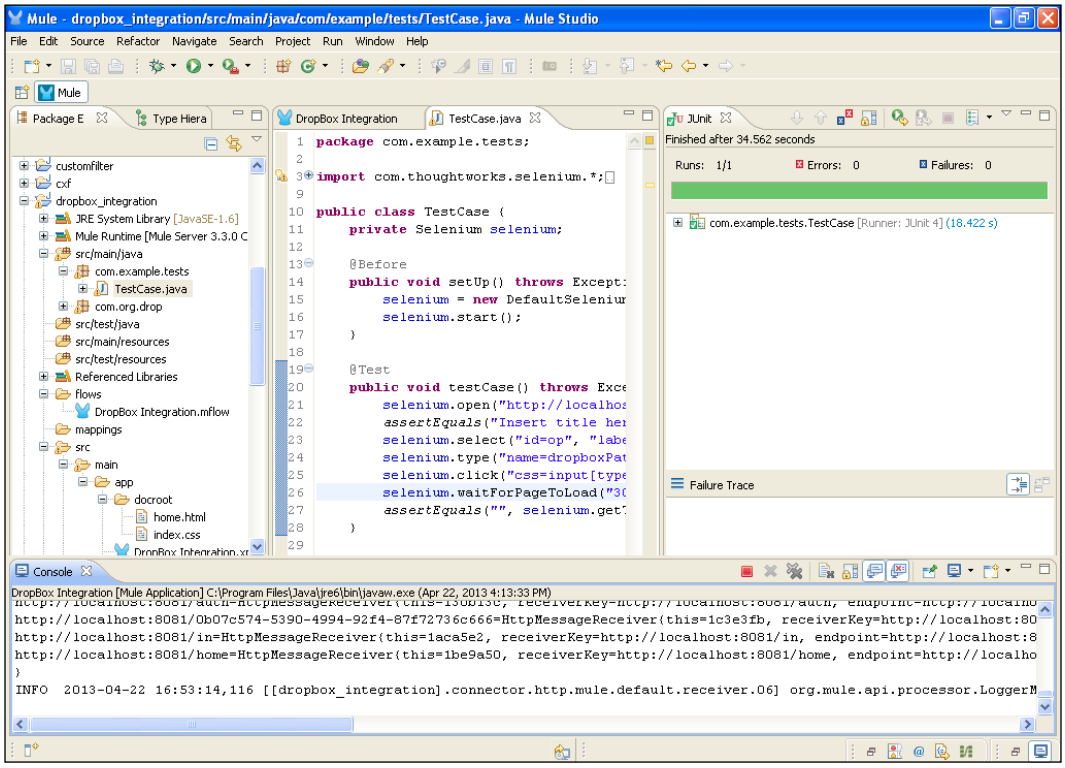
On execution, you will see a screen similar to the following screenshot:



3. Go to your application and right-click on the `TestCase` class. Go to **Run As** | **JUnit**.



4. If your JUnit test case runs successfully, you can see in your Dropbox account. A folder is created automatically in Dropbox.



8
Introducing Web Services

In this chapter, you will learn about web services, integration of web services, and much more. The following topics will be covered in this chapter:

- Proxying web services
- Creating JAX-WS services
- Creating web services using the REST component
- Calling external web services using the SOAP component

Introduction

A **web service** is an application that is written to meet Internet and **Extensible Markup Language** (**XML**) technology standards. It performs a specific task and is made available to other users through a network. In this section, we will learn about the two main types of web services in use: SOAP-based and REST-based. The key characteristics of web services are flexibility, interoperability, and transportability. Today, they are used in a variety of ways, including various web APIs, integration frameworks, and architecture models such as service-oriented architecture. Web services allows different applications to talk to each other and share data and services among themselves. Other applications can also use the services of web services. For example, a VB or .NET application can talk to Java web services and vice versa. So, web services are used to make the application platform and technology independent.

Proxying web services

A proxying web service is a very common application used for different purposes, such as decoupling clients and producers. CXF proxies support working with the SOAP body or the entire SOAP envelope. By default, only the SOAP body is sent as payload, but the payload mode can be set only via the payload attribute to the envelope needed. You can define two types of proxying web services: server-side and client-side.

Getting ready

Mule can perform as a web service proxy. A proxy can perform several useful functions:

- Security enforcement
- WS-Policy enforcement
- Routing to the proper backend service, whether a remote service or a local service
- Protocol bridging, such as HTTP to JMS
- Message transformations, such as converting from old versions of the message format to new versions
- Validation

Mule provides several utilities that help you do these.

In Mule Studio, you can create different types of web services: WS-Security, WS-Proxy, and protocol binding.

- **WS-Security web service**: WS-Security provides means to secure your services above and beyond transport level protocols such as HTTPS through a number of standards such as XML-Encryption, and headers defined in the WS-Security standard.
- **Protocol binding**: This allows you to forward requests from one Endpoint to another. This is generally the best option for proxying web services.
- **WS-Proxy web service**: This allows you to service WSDL files locally while proxying remote web services.

How to do it...

First we will see proxying web services.

Web service proxying

In web service proxying, you have to use CXF proxying for the following:

- To implement WS-Policy assertions
- To easily service a WSDL associated with your service
- To work directly with the SOAP body; for example, adding XML directly to it

A CXF web service standard supports the use of WS-Security and WS-Addressing.

Protocol binding

Protocol binding is used for forwarding a request from one Endpoint to another via service bridging. You can forward the data streams directly or process and transform the XML.

How it works...

The following code snippet is a simple configuration example that forwards a request from one HTTP Endpoint to another:

```
<flow name="HttpProxyService">
      <http:inbound-endpoint address="http://localhost:8888" exchange-
pattern="request-response"/>
<http:outbound-endpoint address="http://www.webservicex.
net#[header:INBOUND:http.request]" exchange-pattern="request-
response"/>
  </flow>
```

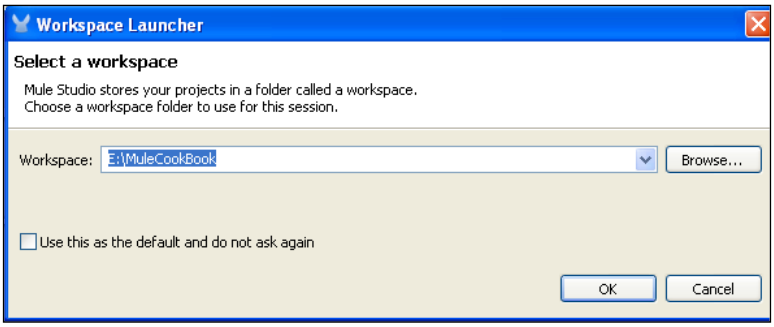
Creating JAX-WS services

In this recipe, you will learn how to create a JAX-WS service, The JAX-WS specification defines a series of APIs and annotations that help you build web services.

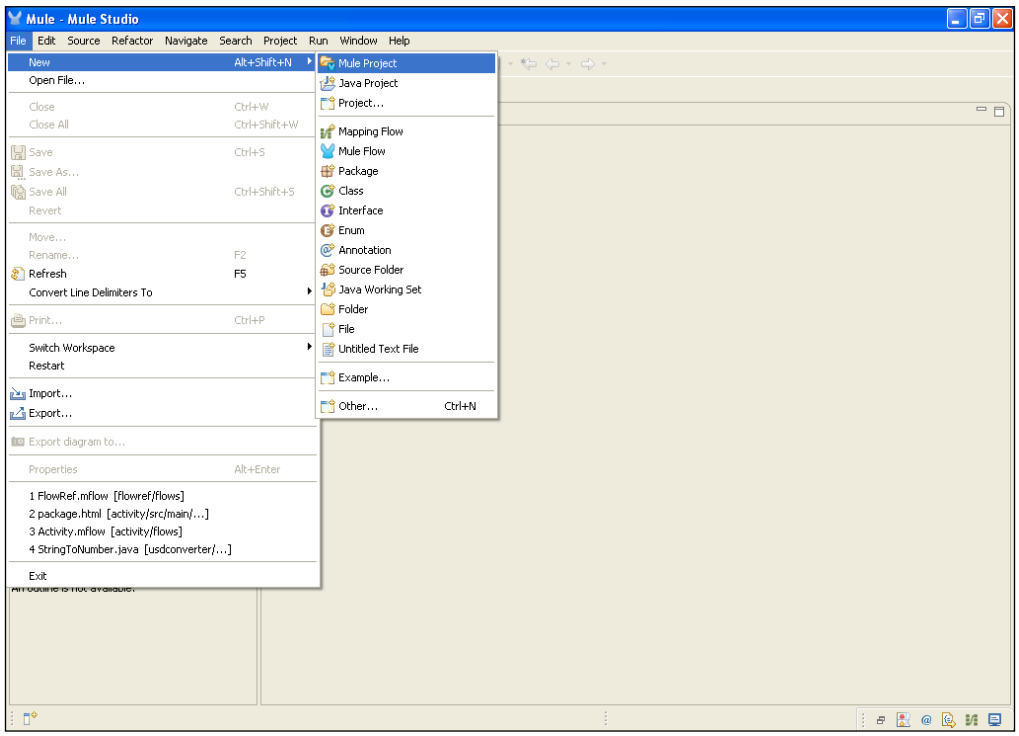
Getting ready

Before creating a JAX-WS service, perform the following steps:

1. Open Mule Studio and enter a name for the workspace as shown in the following screenshot:



2. To create a new project, go to **File** | **New** | **Mule Project**. Enter the project name, `soap-jax-ws`, and click on **Next** and then on **Finish**. Your new project is created. We can start the implementation now.



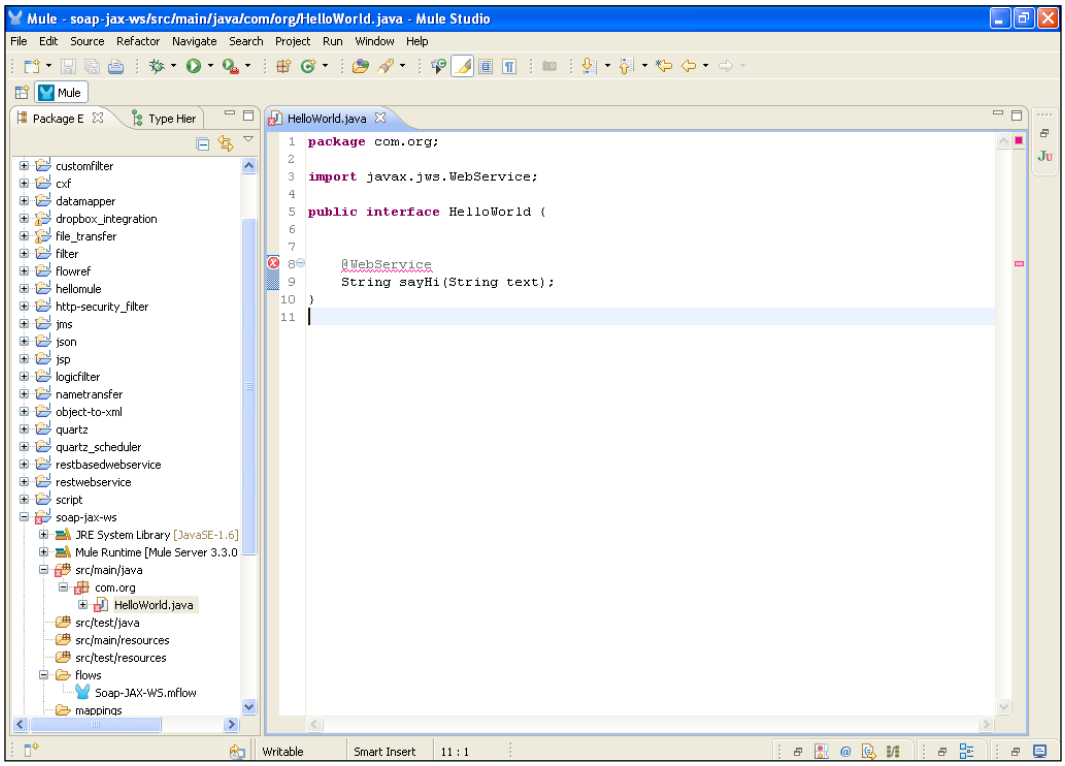
How to do it...

After you're done with the steps mentioned in the *Getting ready* section, perform the following steps to create a JAX-WS service:

1. You begin by writing the service interface. Firstly, go to `src/main/java`, right-click on it, and create an interface called `HelloWorld`. You can write an operation called `sayHello` to greet anyone who submits his or her name through a web browser.

```
import javax.jws.WebService;
public interface HelloWorld {

  @WebService
  String sayHi(String text);
}
```

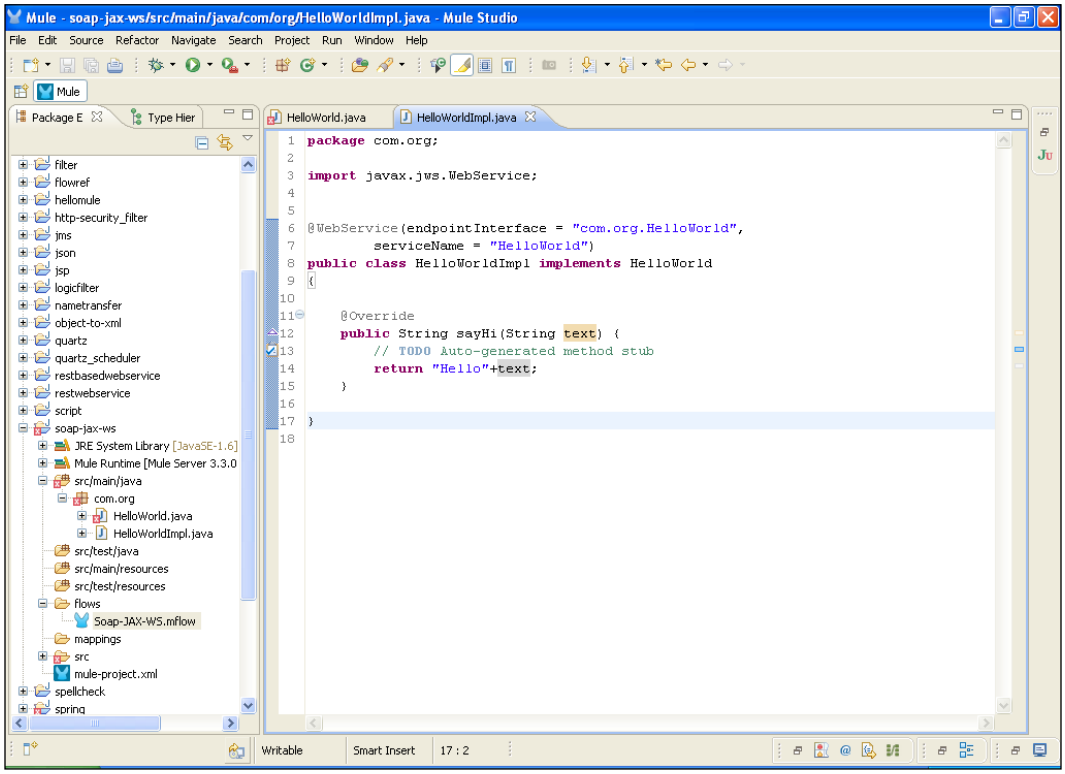
   The following screenshot shows this process:



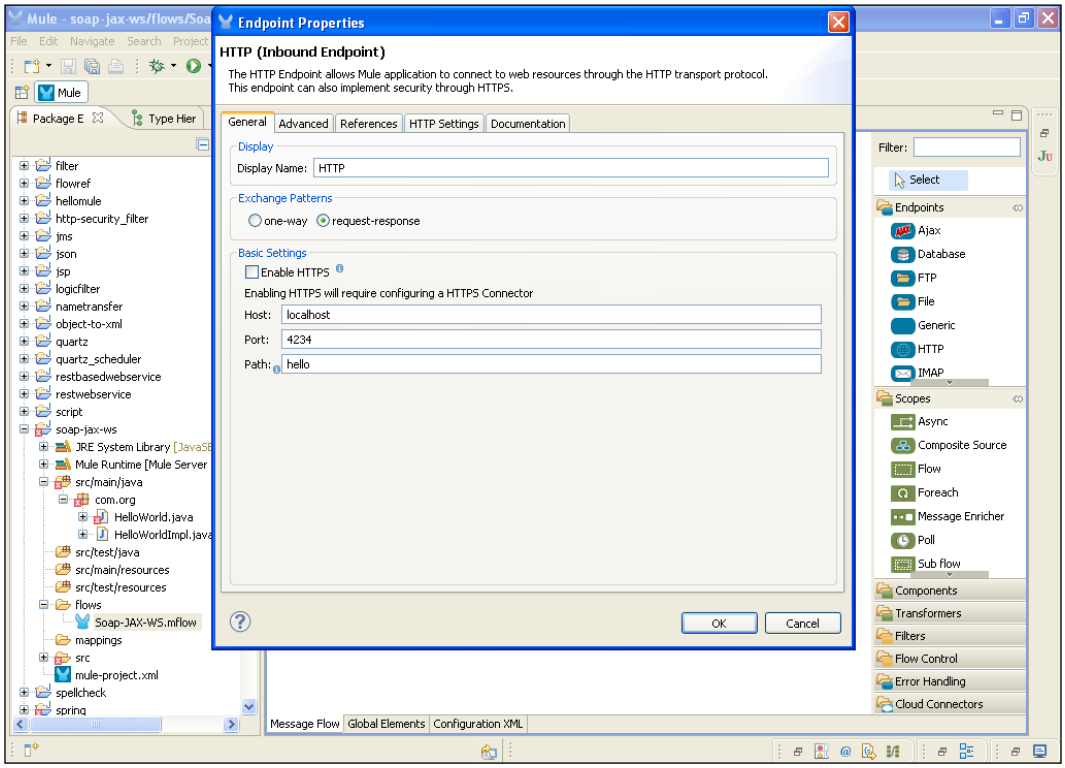
2. To create a class, right-click on `src/main/java` and create a class called `HelloWorldImpl`; implement it with the interface `HelloWorld`.

```
import javax.jws.WebService;

@WebService(endpointInterface = "com.org.HelloWorld",
        serviceName = "HelloWorld")
public class HelloWorldImpl implements HelloWorld
{

  @Override
  public String sayHi(String text) {
    // TODO Auto-generated method stub
    return "Hello"+text;
  }

}
```

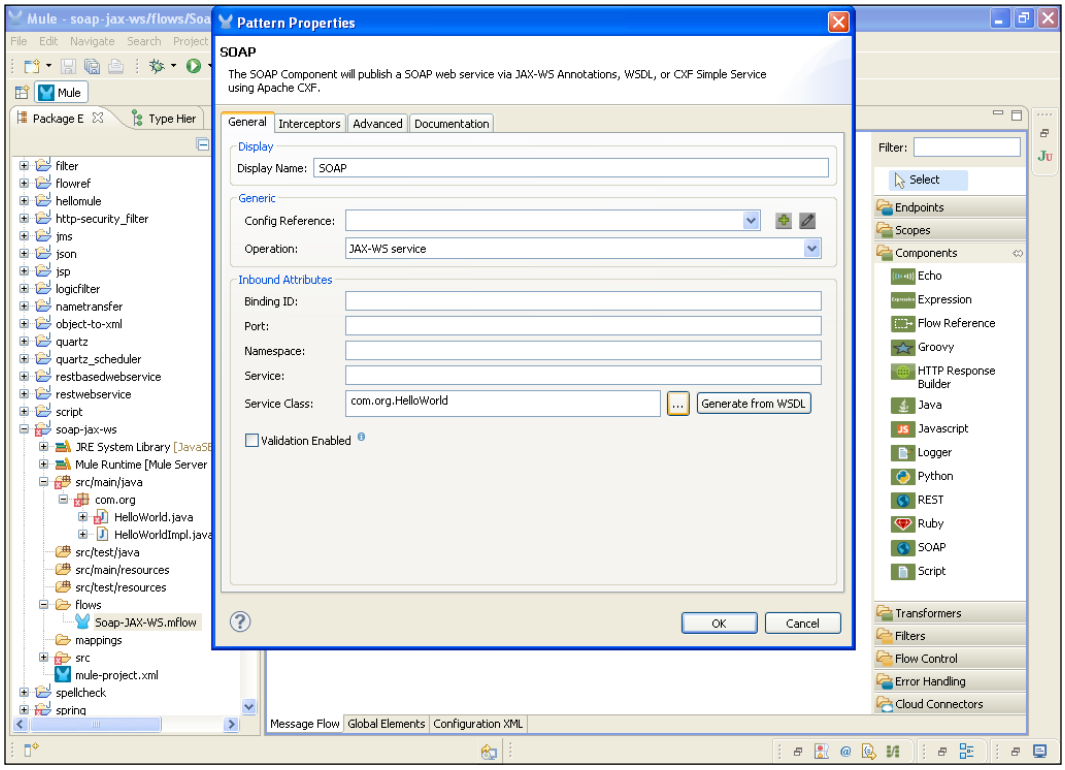
The following screenshot shows the implementation of `HelloWorldImpl` with `HelloWorld`:



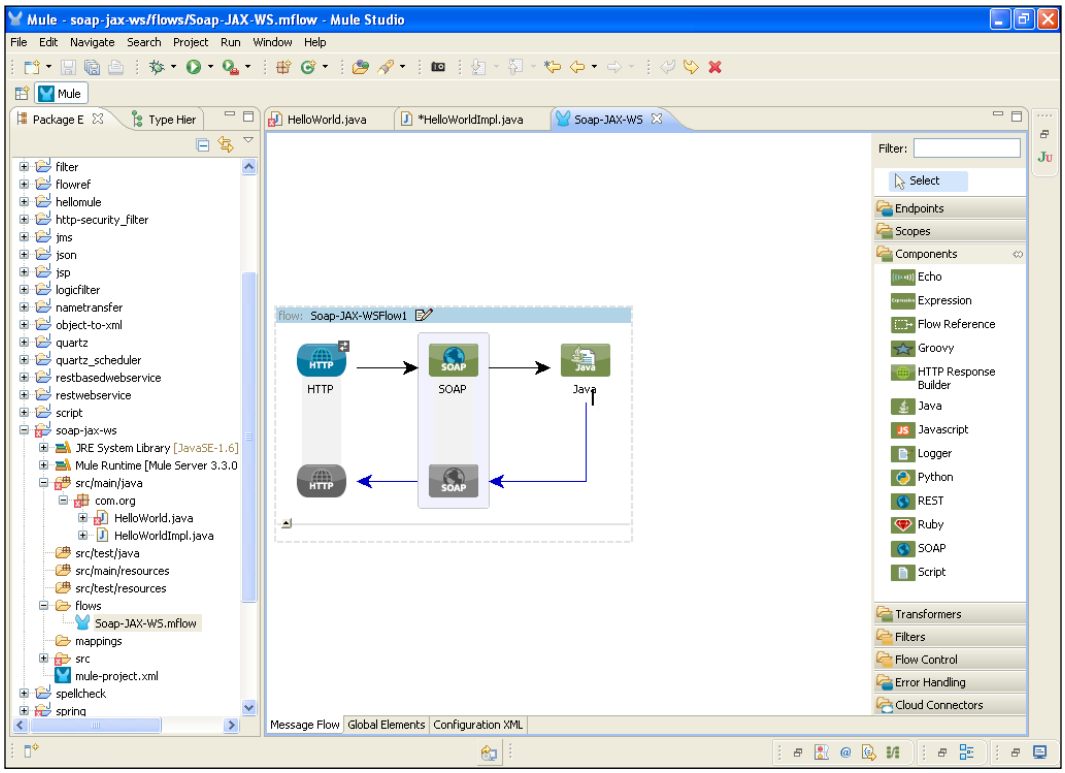
3. Go to the `SOAP-JAX-WS.mflow` file. Drag the **HTTP** Endpoint onto the canvas. Double-click on the **HTTP** Endpoint to configure it. Enter the port number and path.



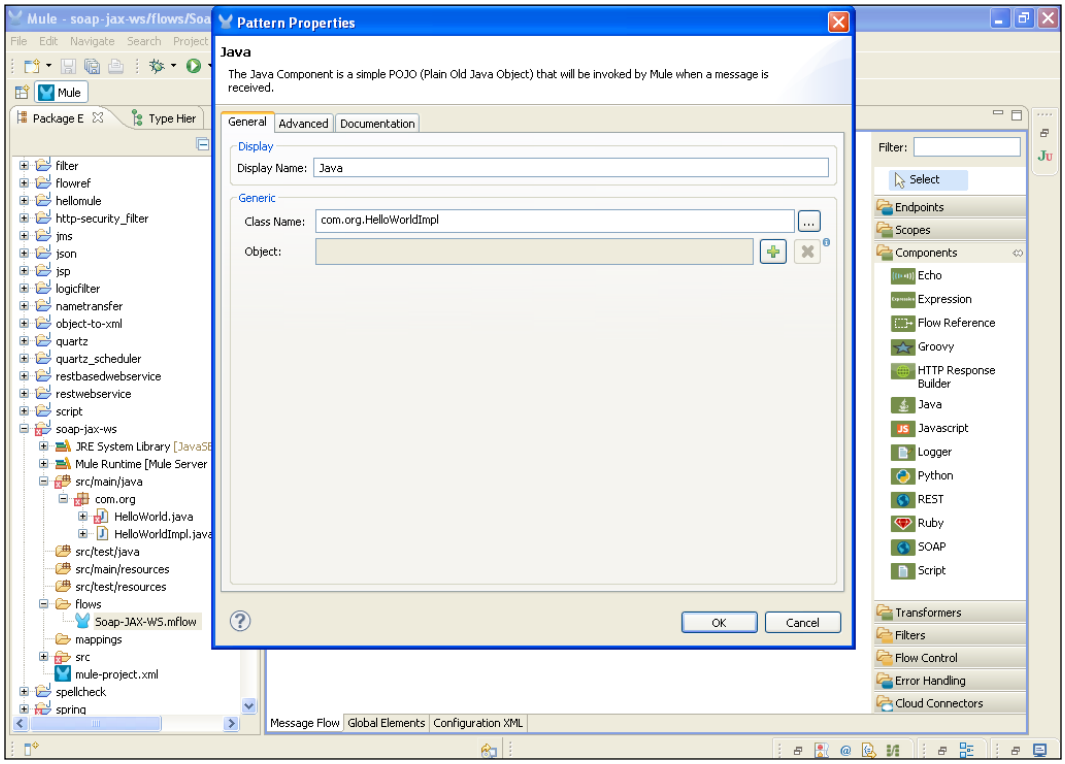
4. To configure the interface, drag the **SOAP** component onto the canvas. Double-click on the **SOAP** component and configure the service class, and choose the operation **JAX-WS service**.



5. Drag the **Java** component onto the canvas. Your flow will look like this:



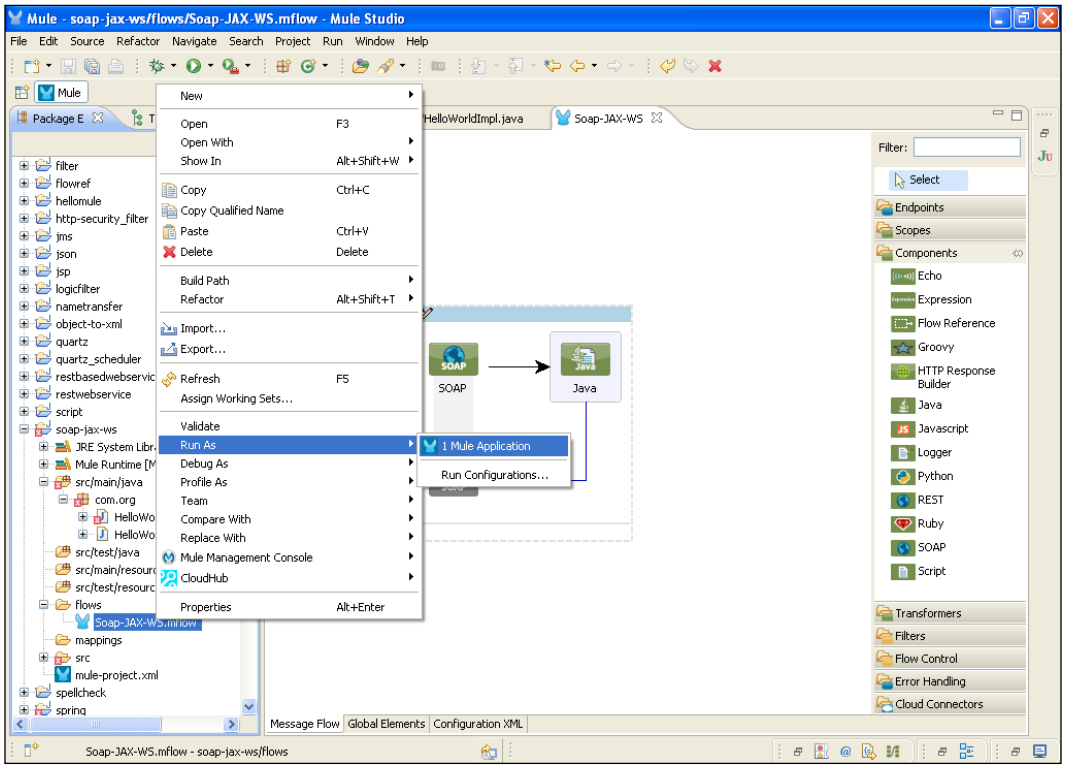
6. Double-click on the **Java** component to configure it. Here, you import the `HelloWorldImpl` class that was created before.



How it works...

In this section, you will see how to deploy your application using Mule Studio.

1. To deploy the application code in the Mule server, go to **Run As** | **Mule Application**; the Mule server will deploy your application.



2. Open a browser and paste this URL: `http://localhost:4234/hello?wsdl`. This will display the WSDL generated by CXF.

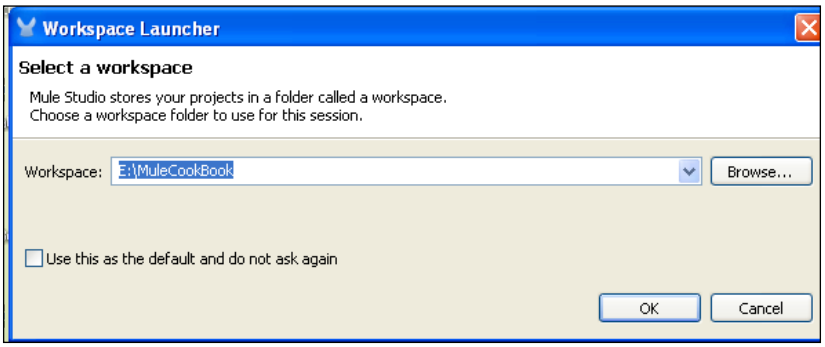
Creating web services using the REST component

Using **REST**, which stands for **Representational State Transfer**, applications can transmit the information needed to function as web services using only the native functions of a given protocol. In the context of web services, this generally means that RESTful web services communicate via pure HTTP using XML or JSON to encapsulate the data and metadata.

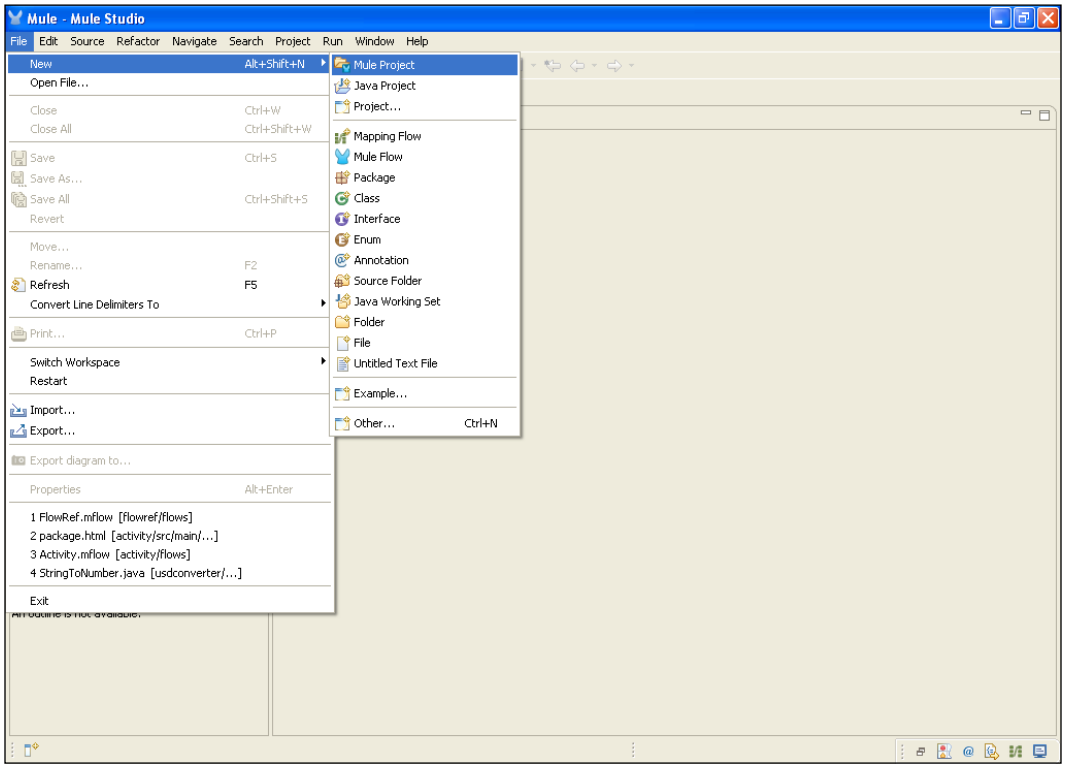
Getting ready

In this section, you will use three components: the HTTP Endpoint, the Logger component, and the REST component. The REST component is used for creating a REST-based web service.

1. Open Mule Studio and enter a name for the workspace as shown in the following screenshot:



2. To create a new project, go to **File** | **New** | **Mule Project**. Enter the project name, `restbasedwebservice`, and click on **Next** and then on **Finish**. Your new project is created. We can start the implementation now.



How to do it...

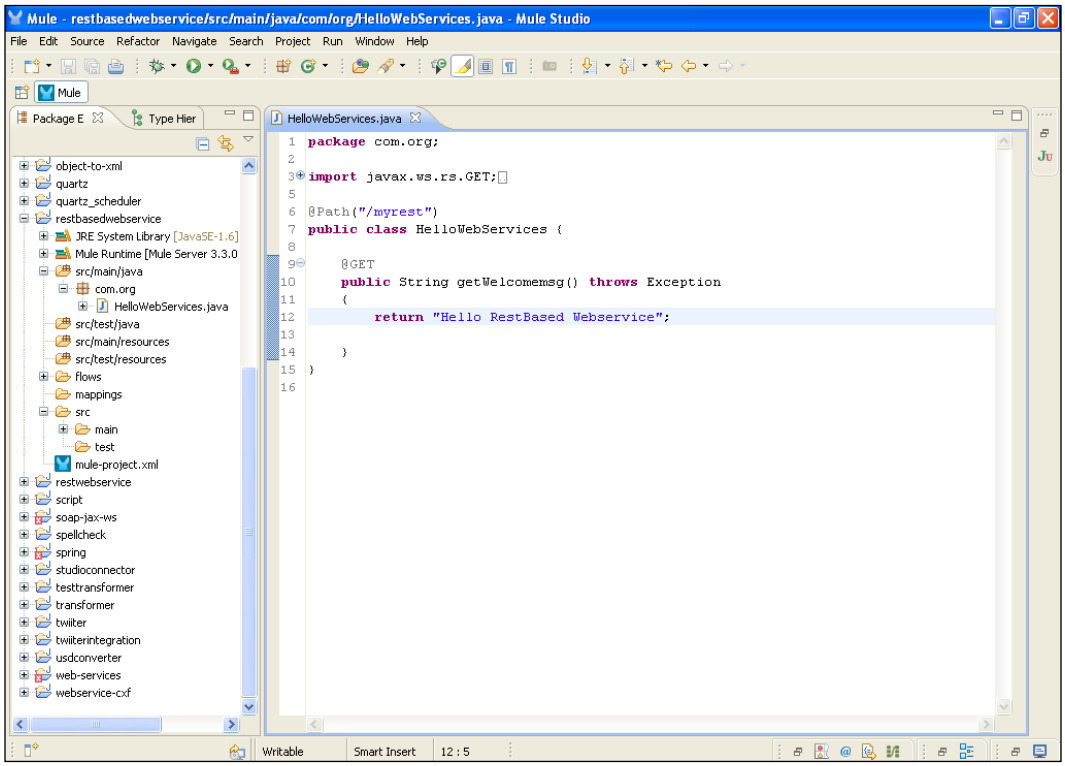
In this section, you will see how to create a REST-based web service using Java annotations. Java annotations are one of the main development features that was introduced in JDK 5. Annotations are like meta tags that you can add to your code and apply to package declarations. Annotations carry data over a runtime.

1. Go to `src/main/java`, right-click on it, and create a class called `HelloWebService`.

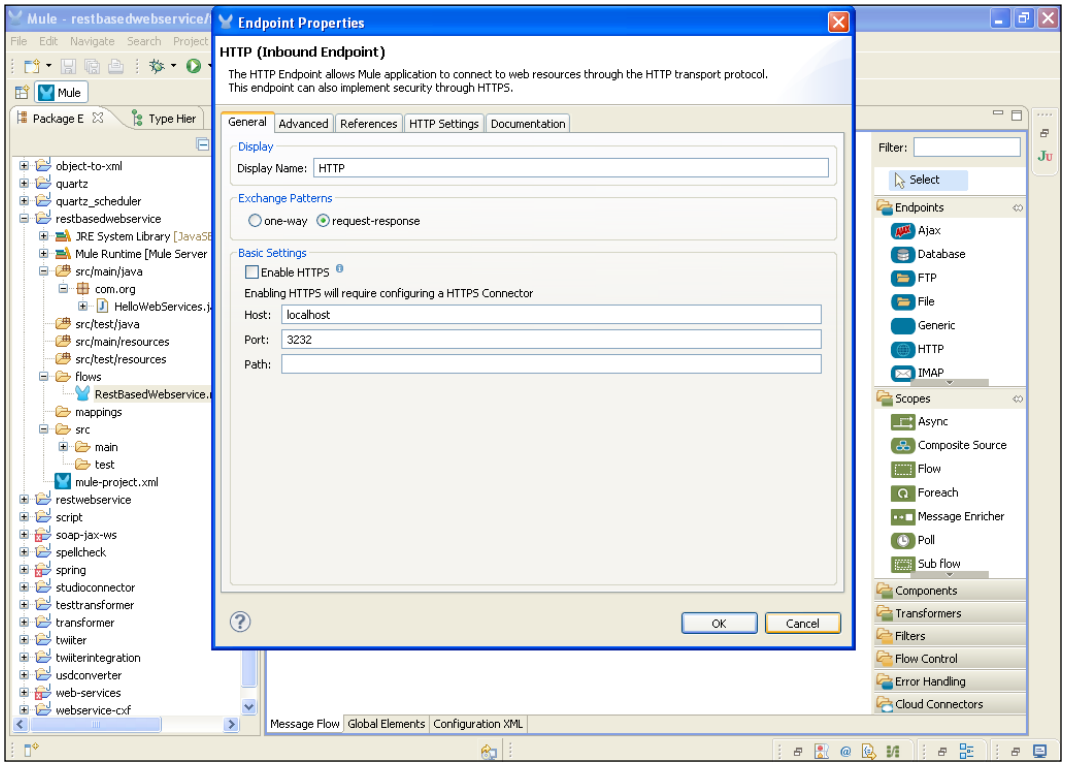
```
import javax.ws.rs.GET;
import javax.ws.rs.Path;

@Path("/myrest")
public class HelloWebServices {

  @GET
```

```
    public String getWelcomemsg() throws Exception
    {
      return "Hello RestBased Webservice";

    }
}
```

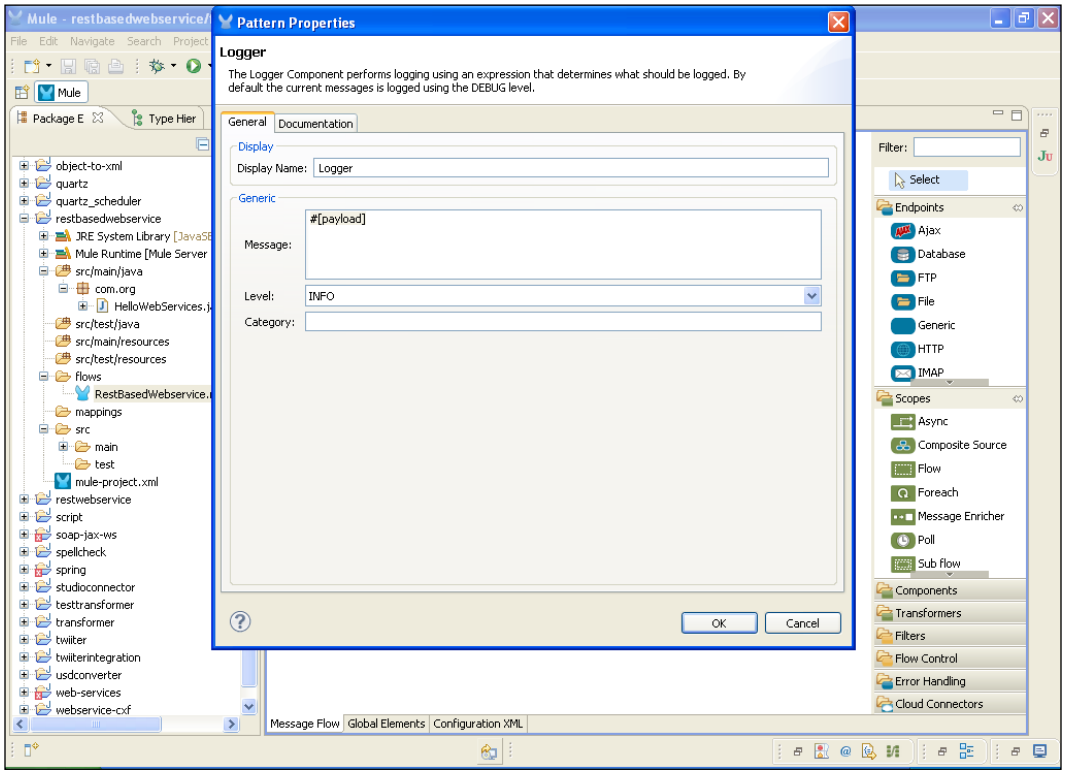
The following screenshot shows the creation of this class:



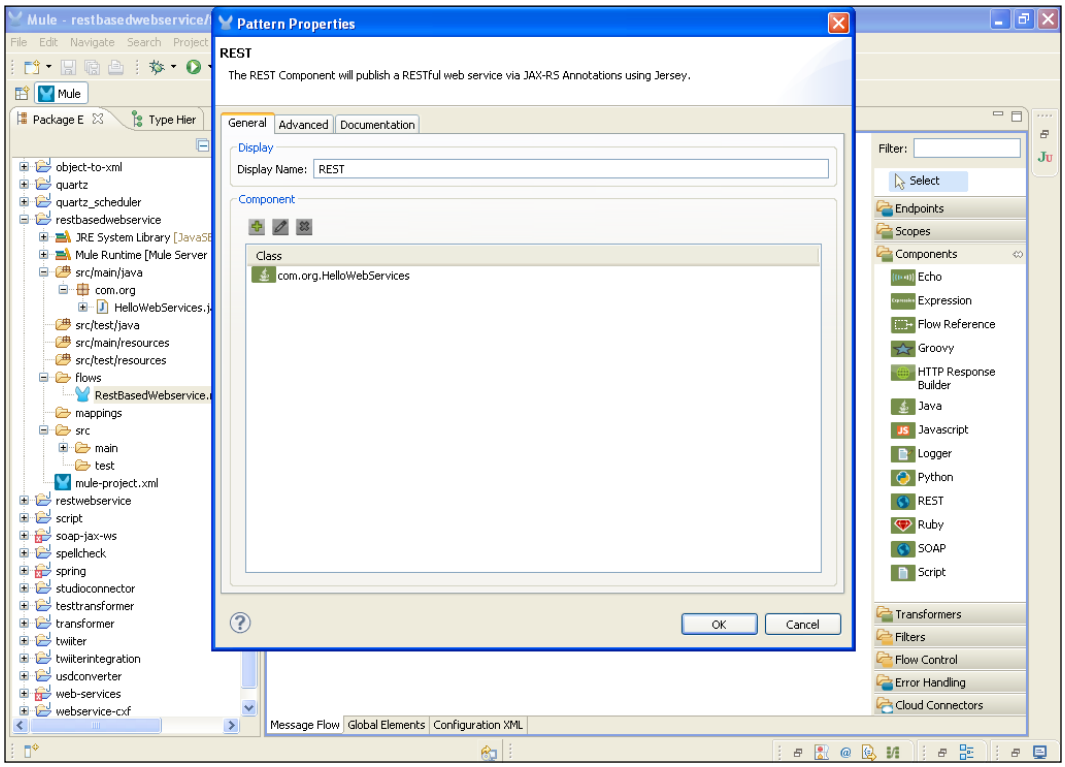
2. Drag the **HTTP** Endpoint onto the canvas. Double-click on the **HTTP** Endpoint to configure it. Enter the port number as `3232`.



3. Drag the **Logger** component onto the canvas. The **Logger** component is used for displaying logs on the console. Double-click on the **Logger** component to configure it.



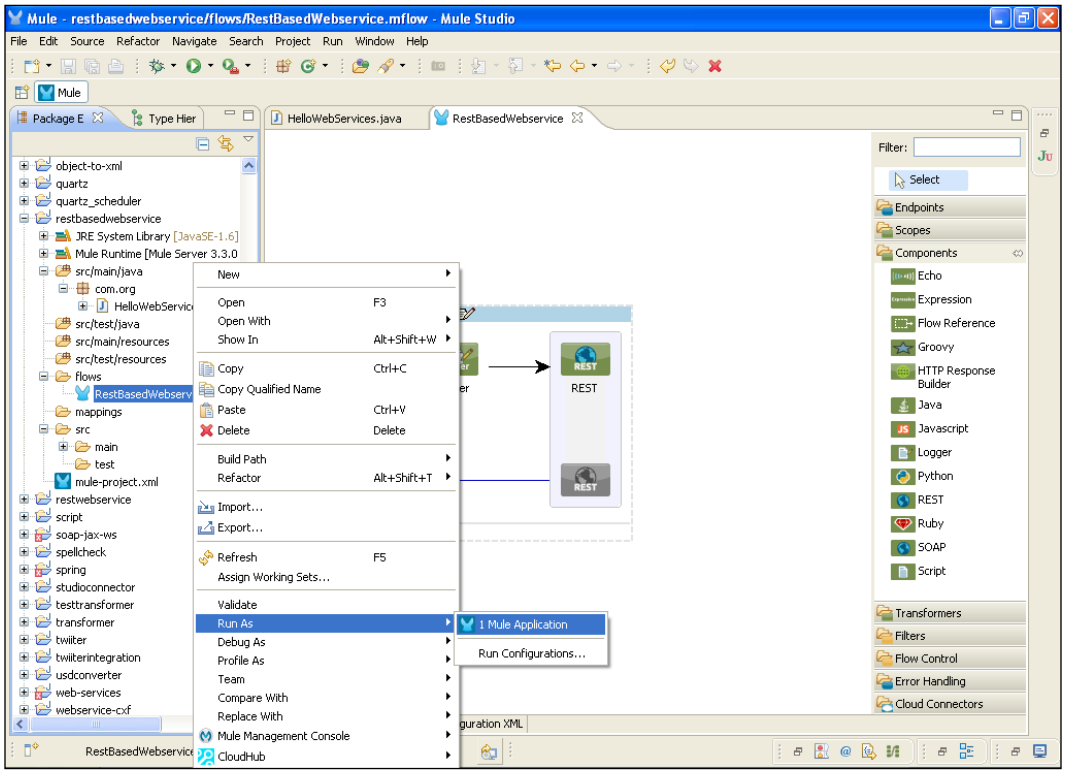
4. Drag the **REST** component onto the canvas. Double-click on it and configure it.



How it works...

In this section, you will see how to deploy applications in Mule Studio.

1. To deploy the application code in the Mule server, go to **Run As** | **Mule Application**; the Mule server will deploy your application.



2. Open a browser and paste the URL `http://localhost:3232/myrest` in the browser.

Calling external web services using the SOAP component

In this recipe, you will see how to create a simple web service using Eclipse in Java; this web service will be called in Mule Studio using the SOAP component. You will see how to call external web services in Mule. You will then create a simple web service that will display the current date and time and the reverse string.

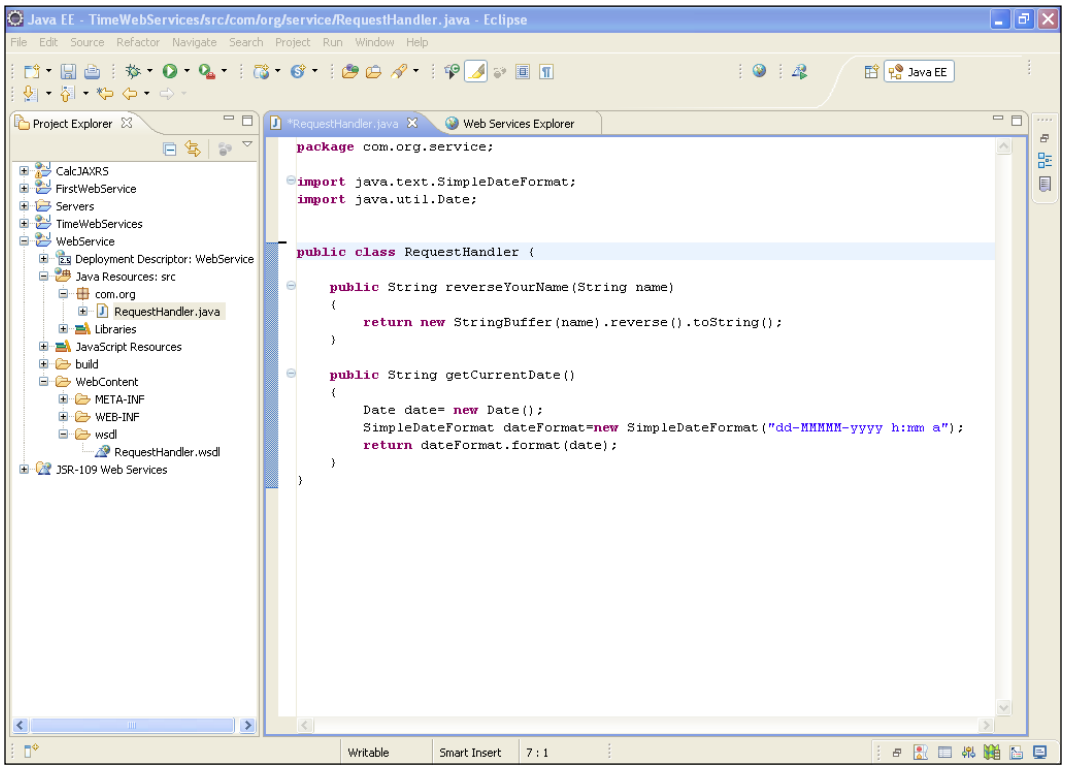
Getting ready

In this section, we will create a web service in Eclipse.

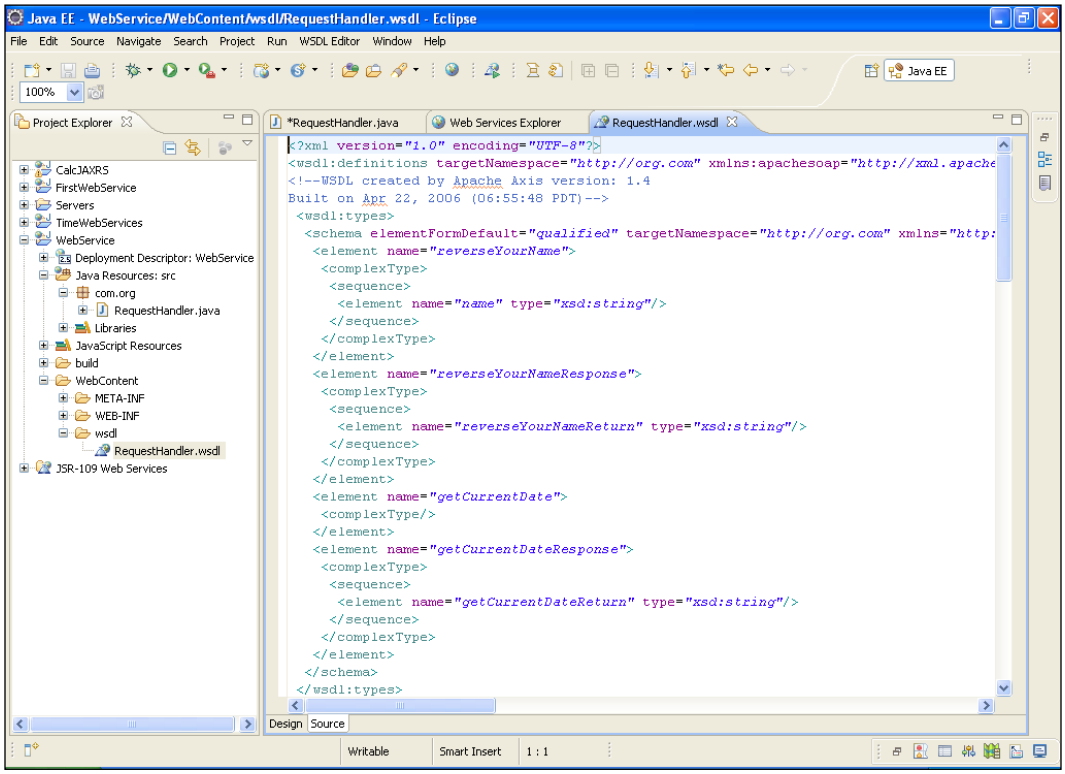
1. Open Eclipse and enter a name for the workspace. Create a class called `RequestHandler`. In this class, we create two methods: one to display the current time and date and the other to generate the reverse string.

```
 import java.text.SimpleDateFormat;
 import java.util.Date;

public class RequestHandler {

  public String reverseYourName(String name)
  {
    return new StringBuffer(name).reverse().toString();
  }

  public String getCurrentDate()
  {
    Date date= new Date();
    SimpleDateFormat dateFormat=new SimpleDateFormat("dd-MMMMM-
yyyy h:mm a");
    return dateFormat.format(date);
  }
}
```

The following screenshot shows the creation of this class:



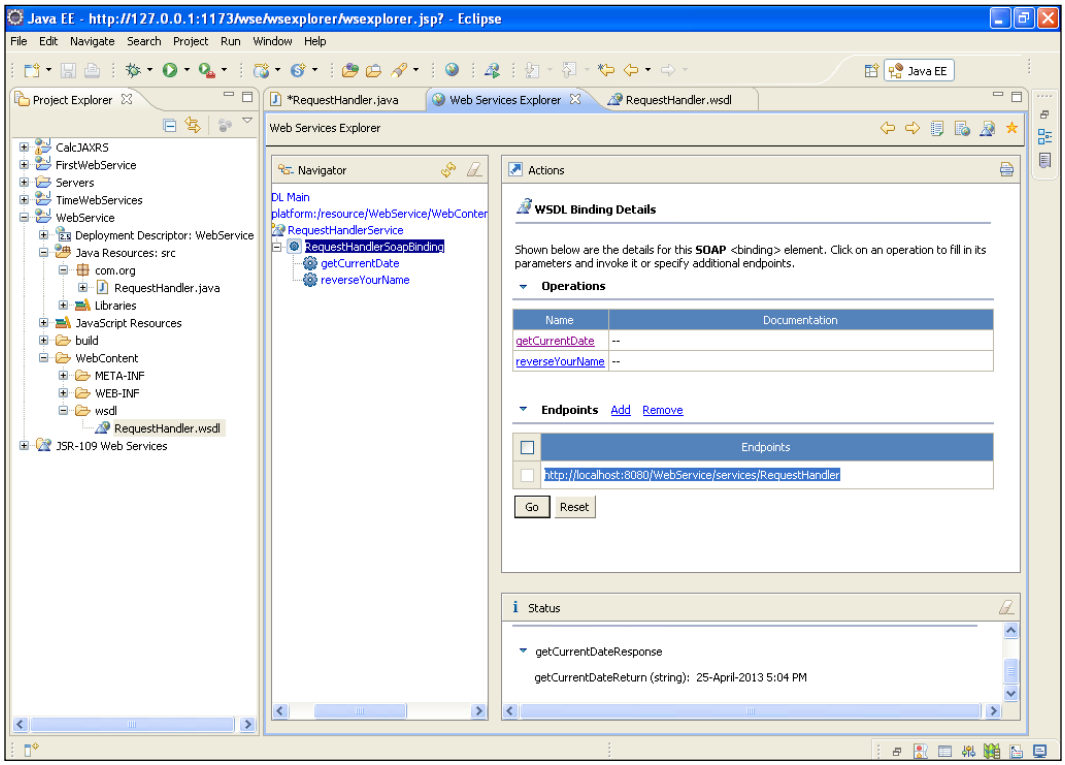
2. The following screenshot shows the class you created as a web service. We will generate the WSDL file as well.



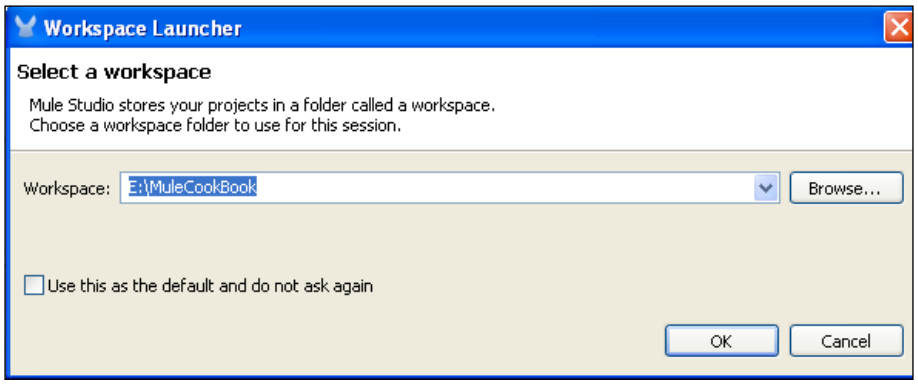
How to do it...

In this section, you will see how to run the WSDL file and see the output on the WSDL page.

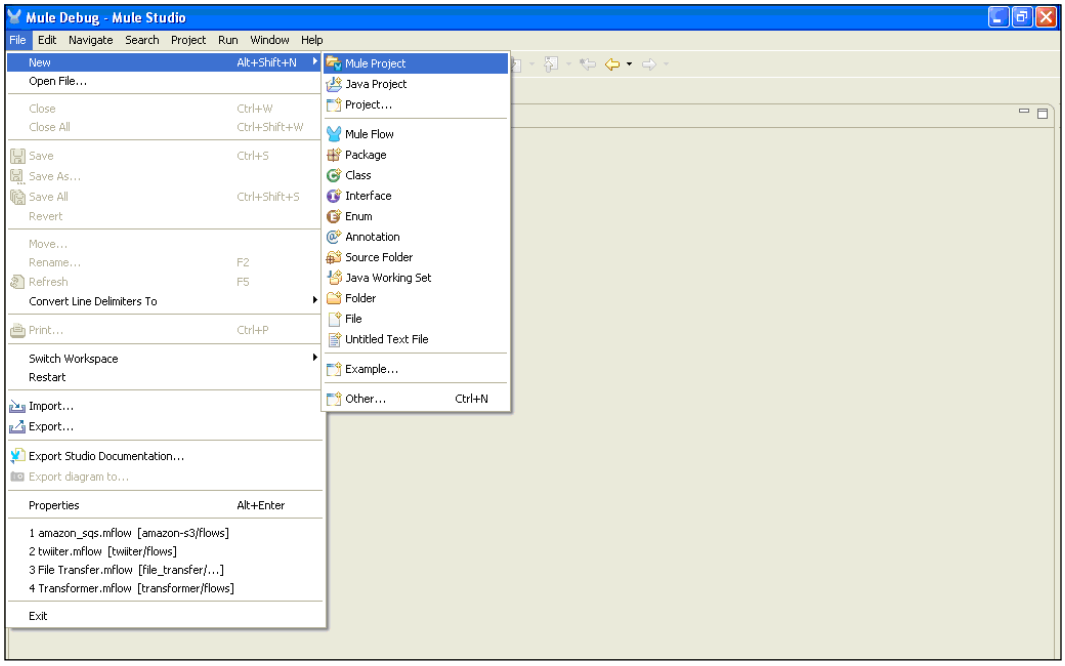
1. Right-click on the `RequestHandler.wsdl` file and publish the WSDL file. Once the file is successfully published, you will see something similar to the following screenshot on your system. You can run the `.wsdl` file in two ways: right-click on the `.wsdl` file or right-click on the project and run the `.wsdl` file. You will see the output on the **Status** tab.



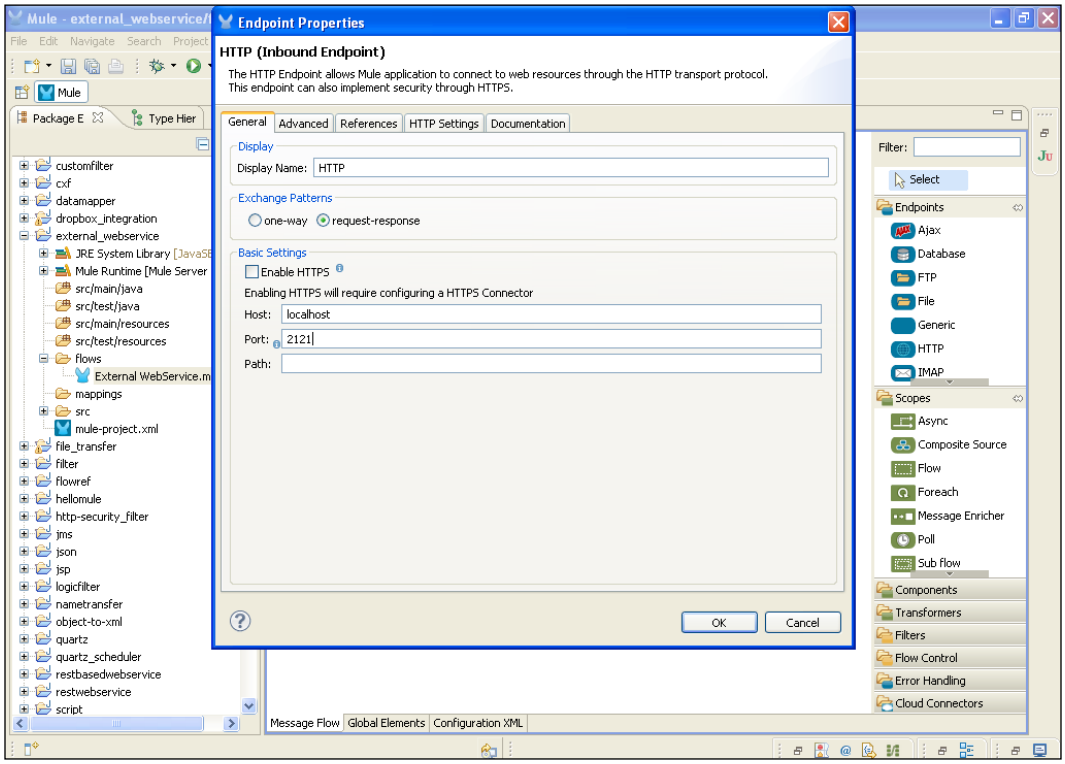
2. Open Mule Studio and enter the workspace name as shown in the following screenshot:



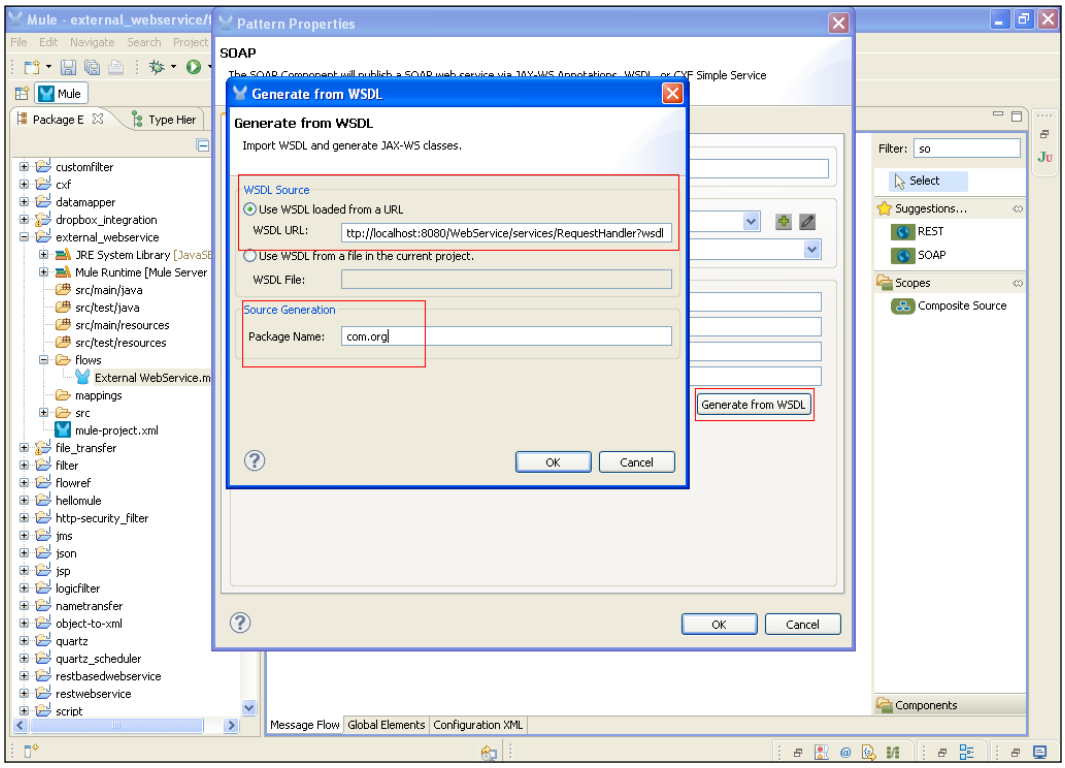
3. To create a new project, go to **File** | **New** | **Mule Project**. Enter the project name, `external_webservice`, and click on **Next** and then on **Finish**. Your new project is created. We can start the implementation now.



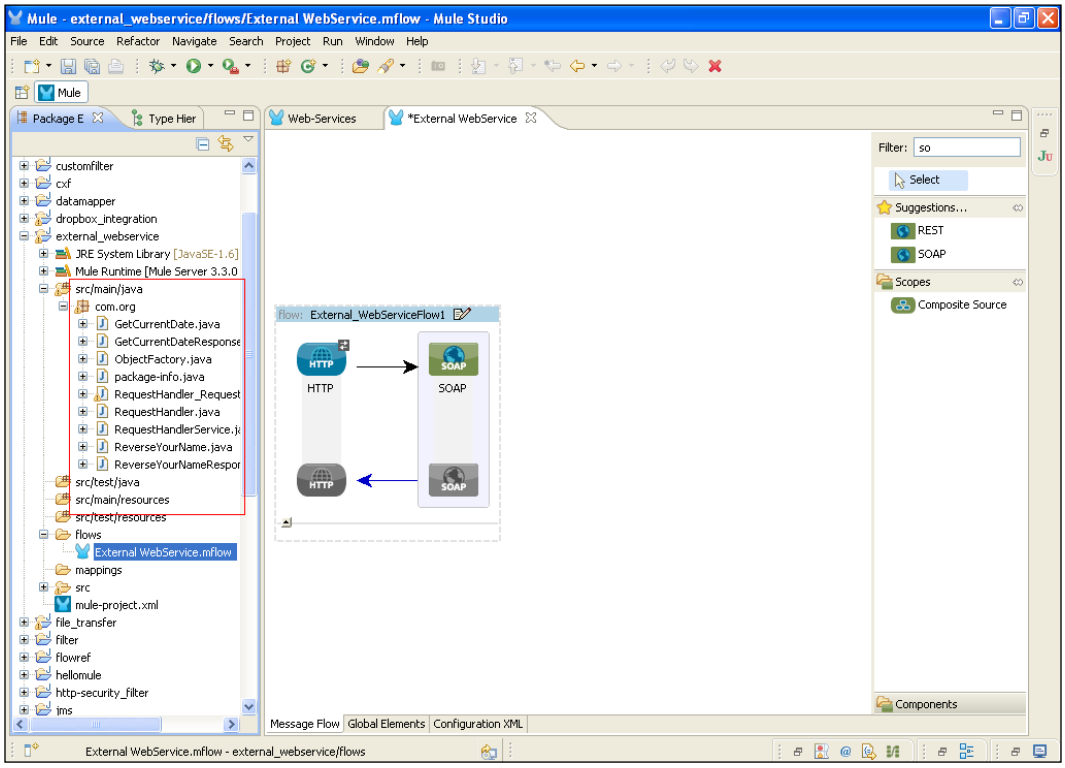
4. Go to the `External WebService.mflow` file. Drag the **HTTP** Endpoint onto the canvas and double-click on it to configure it.



5. Drag the **SOAP** component onto the canvas. Double-click on the **SOAP** component to configure it. First click on the **Generate from WSDL** button; you will have to enter the WSDL URL, `http://localhost:8080/WebService/services/RequestHandler?wsdl`; this URL was created in the external web service in Eclipse. Enter the package name as `com.org`. This package name comes from the external web service. Click on the **OK** button.



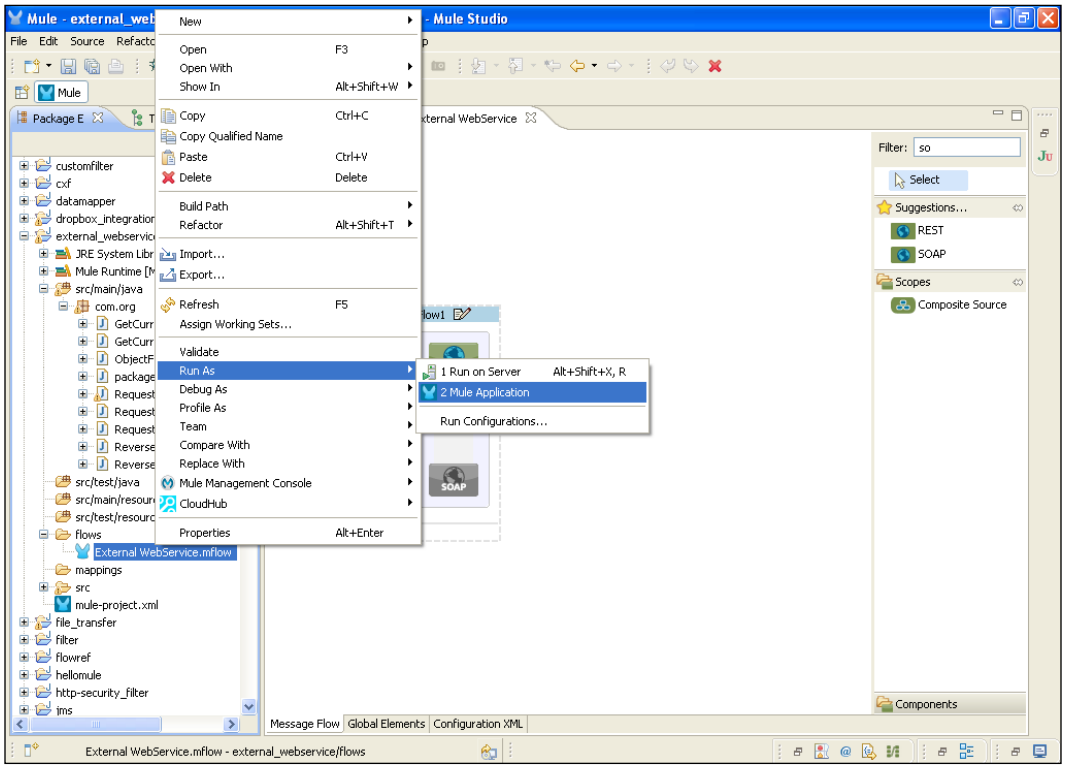
6. You will see that, under `src/main/java`, classes are created automatically, as shown in the following screenshot:



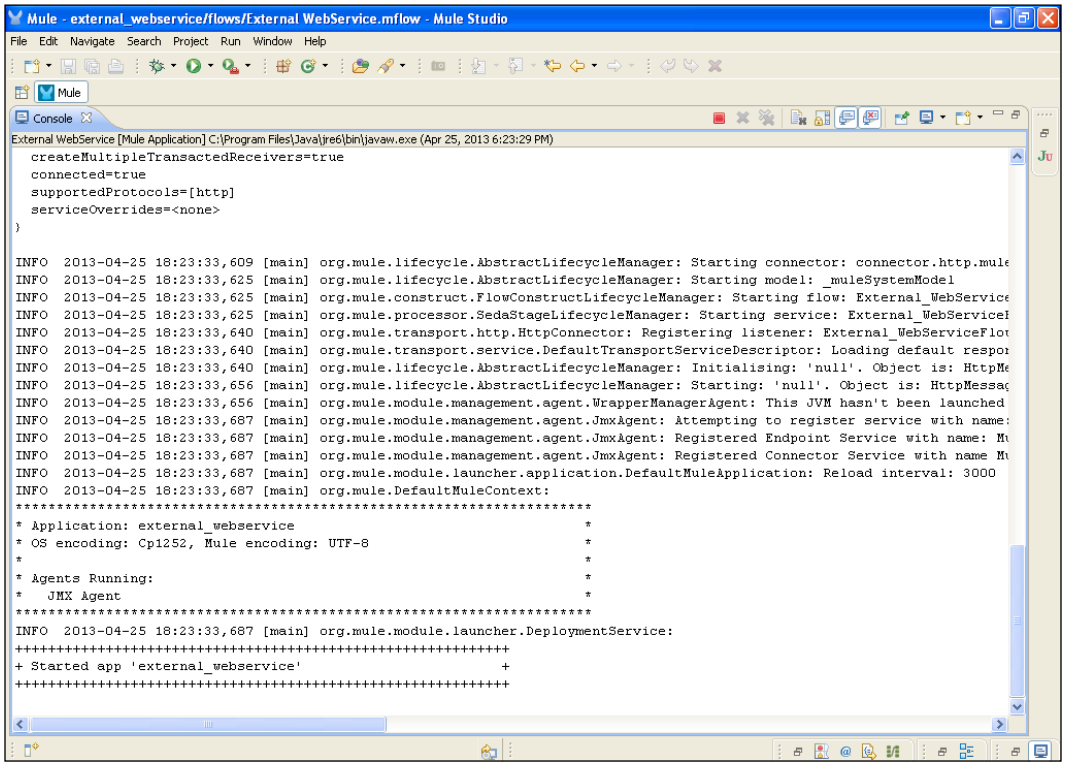
How it works...

In this section, you will see how to deploy the application.

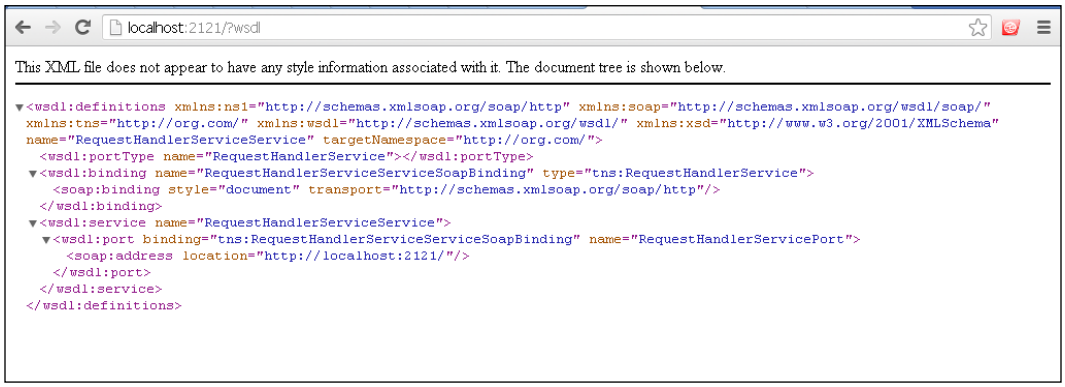
1. To deploy the application code in the Mule server, go to **Run As** | **Mule Application**; the Mule server will deploy your application. At the same time, you should run the Tomcat server in Eclipse.



2. Once you have successfully deployed the application, you will see the following output on your console:



3. Open a browser and paste the URL `http://localhost:2121/?wsdl` in your browser bar.



9

Understanding Flows, Routers, and Services

In this chapter you will learn different types of Routers/Flow Controls, and also the following:

- Configuring the All Router/Flow Control
- Configuring the Choice Router/Flow Control
- Configuring the Splitter Flow Control

Introduction

Flow Controls route messages to various destinations in a Mule flow. Some Flow Controls in business logic are implemented to study and possibly transform messages before routing takes place. Through Flow Controls, you will see how messages are sent and received within a Mule flow. In this chapter, you will see all types of Routers/Flow Controls. There are different types of Routers in Mule Studio: the **Choice Router**, the **All Router**, and the **Splitter**.

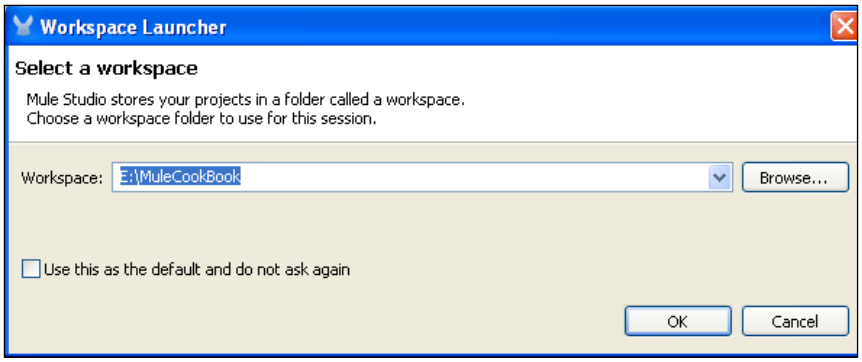
Configuring the All Router/Flow Control

The All Router/Flow Control is used for sending a message to multiple targets. It is also used to route the same message to more than one processor component.

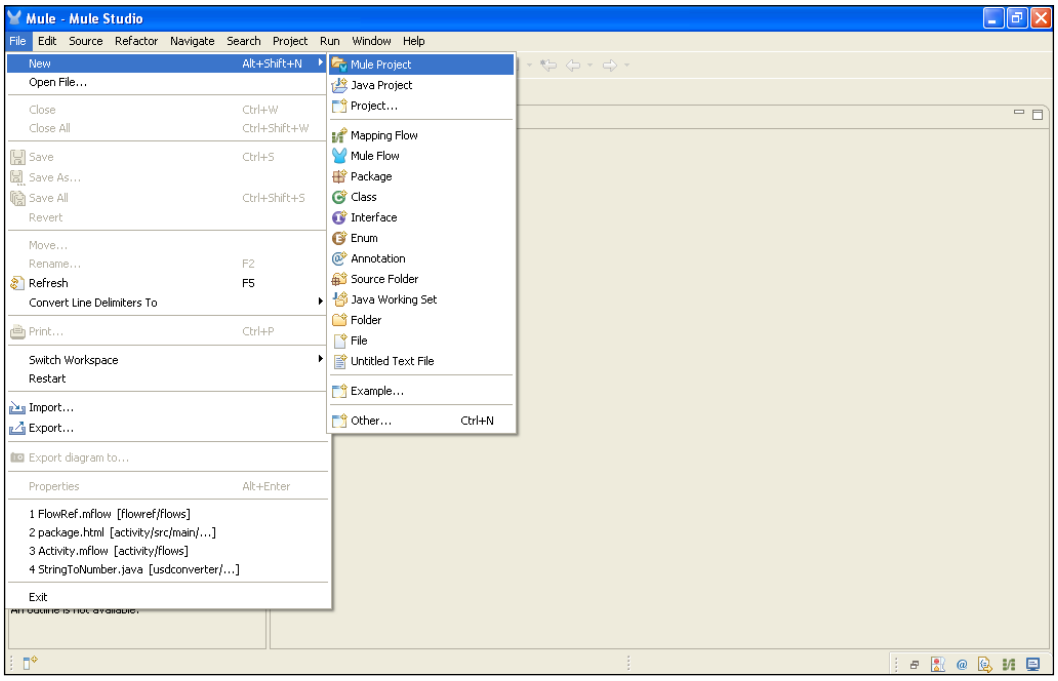
Getting ready

In this example, you will use four components: the File Inbound Endpoint, the File Outbound Endpoint, the All Router, and the Echo component.

1. Open Mule Studio and enter the workspace name as shown in the following screenshot:



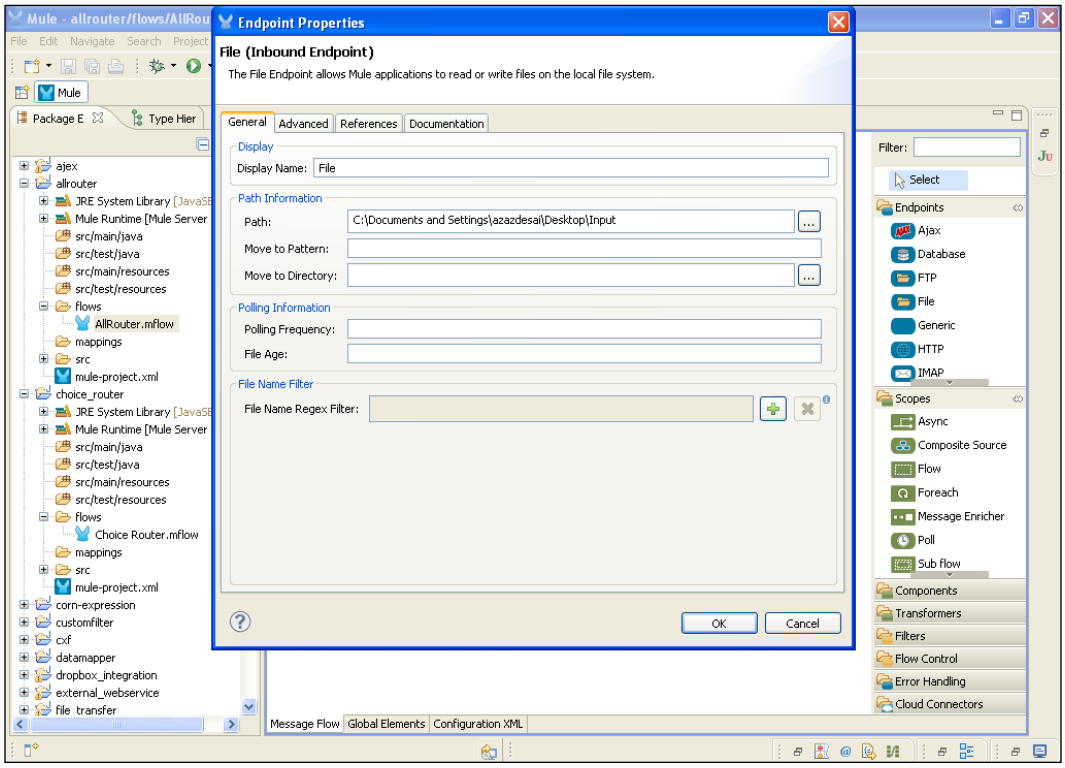
2. To create a new project, go to **File** | **New** | **Mule Project**. Enter the project name, `allrouter`, and click on **Next** and then on **Finish**. Your new project is created, now you have to start the implementation.



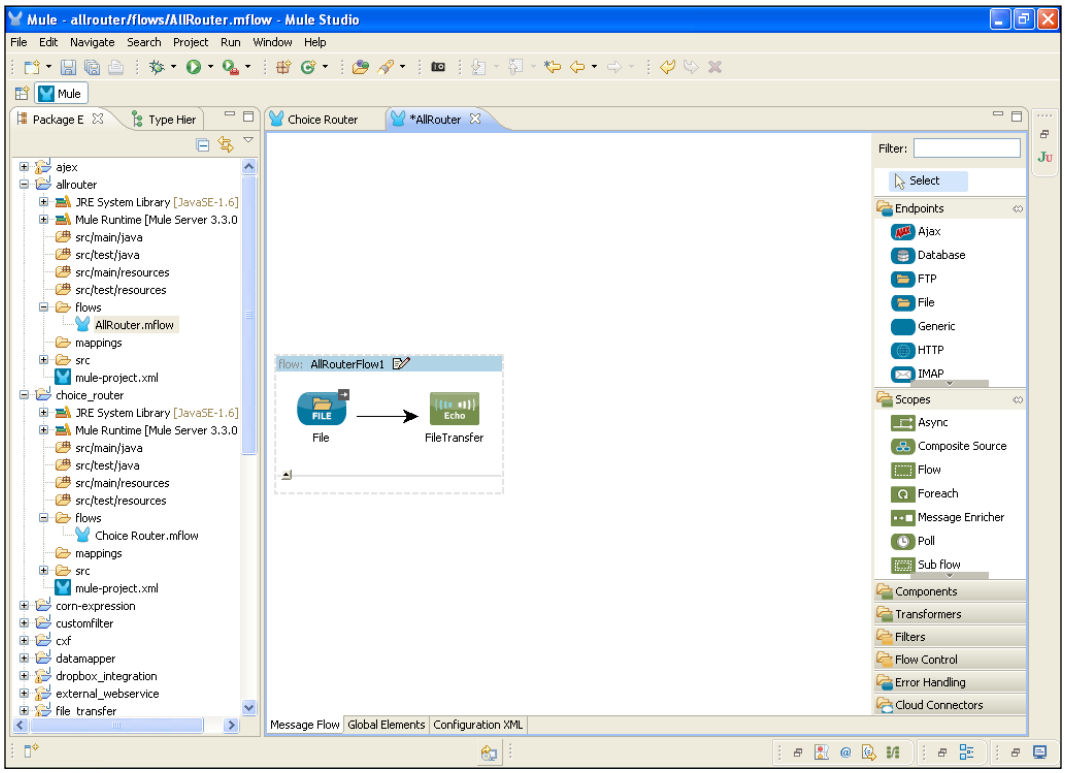
How to do it...

In this section, you will see how to use components in a graphical flow and how it works.

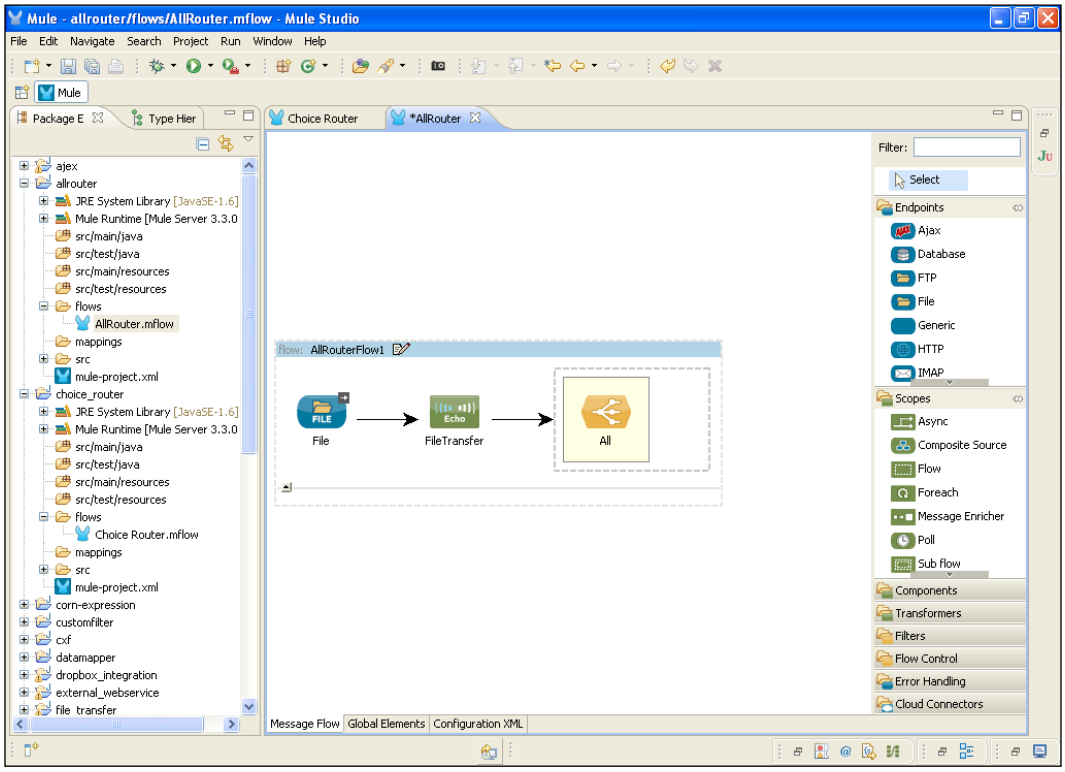
1. Go to the `allrouter.mflow` file and drag the **File** Inbound Endpoint. Double-click on the **File** Endpoint and configure it. Provide a full path name.



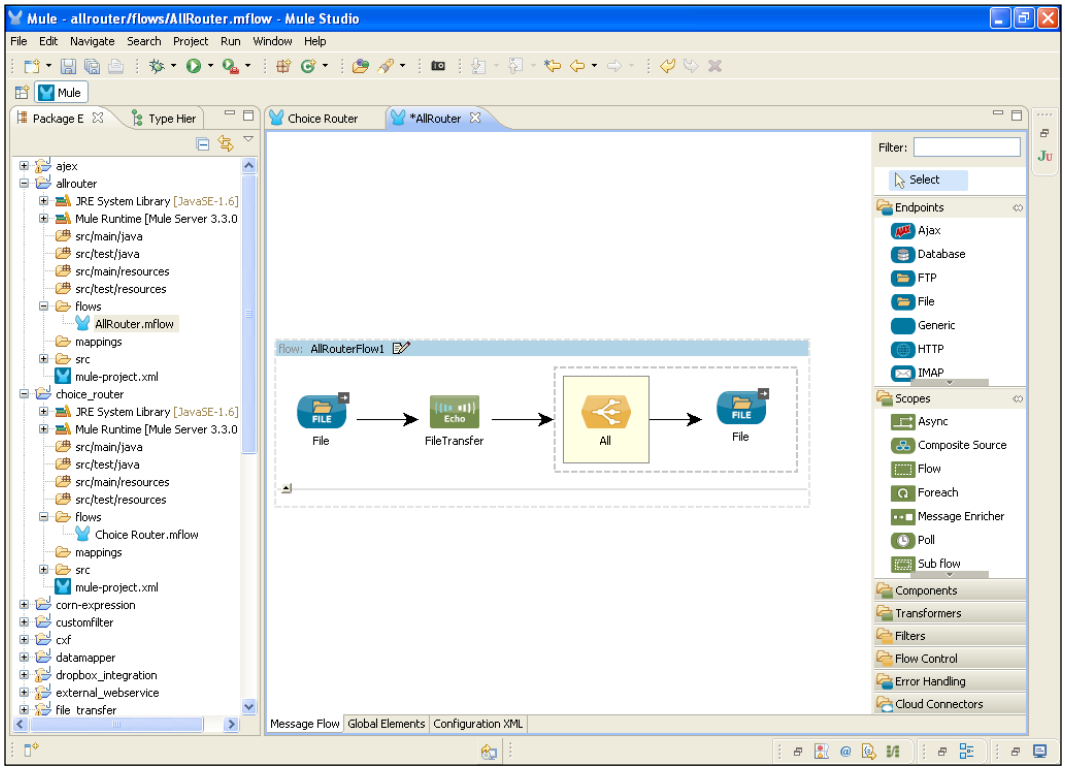
2. Drag the **Echo** component onto the canvas; it's used for display purposes.



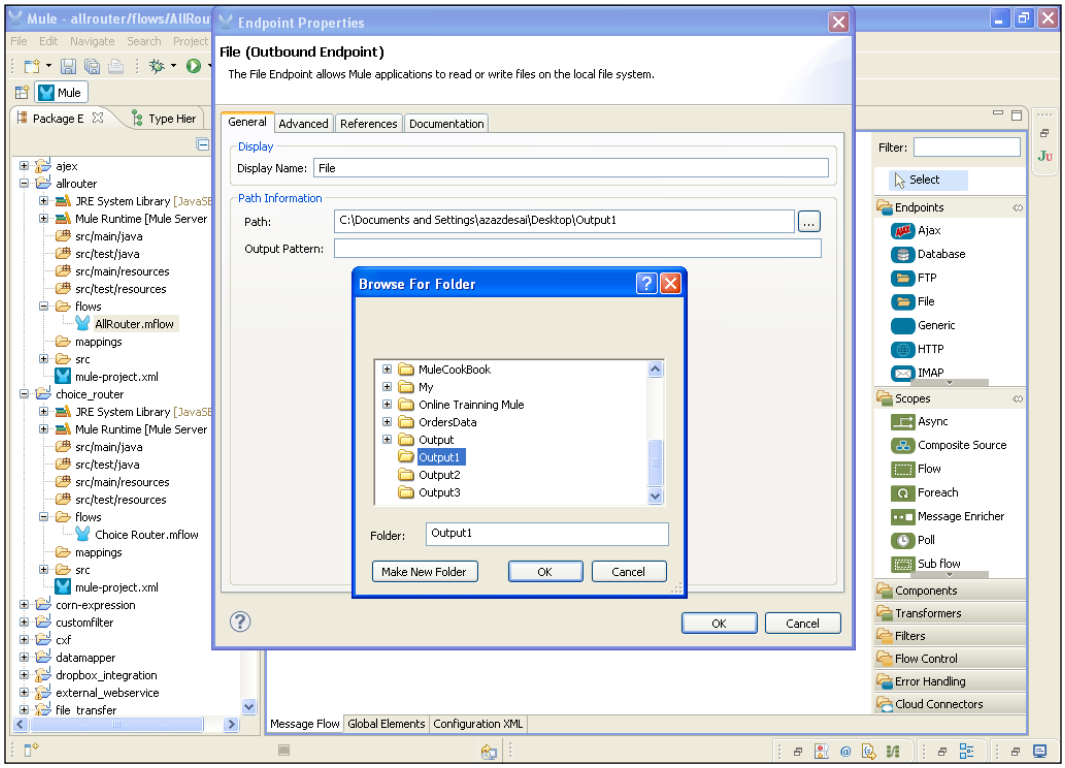
3. Drag the **Choice** Router/Flow Control onto the canvas; it's used for sending messages to multiple targets.



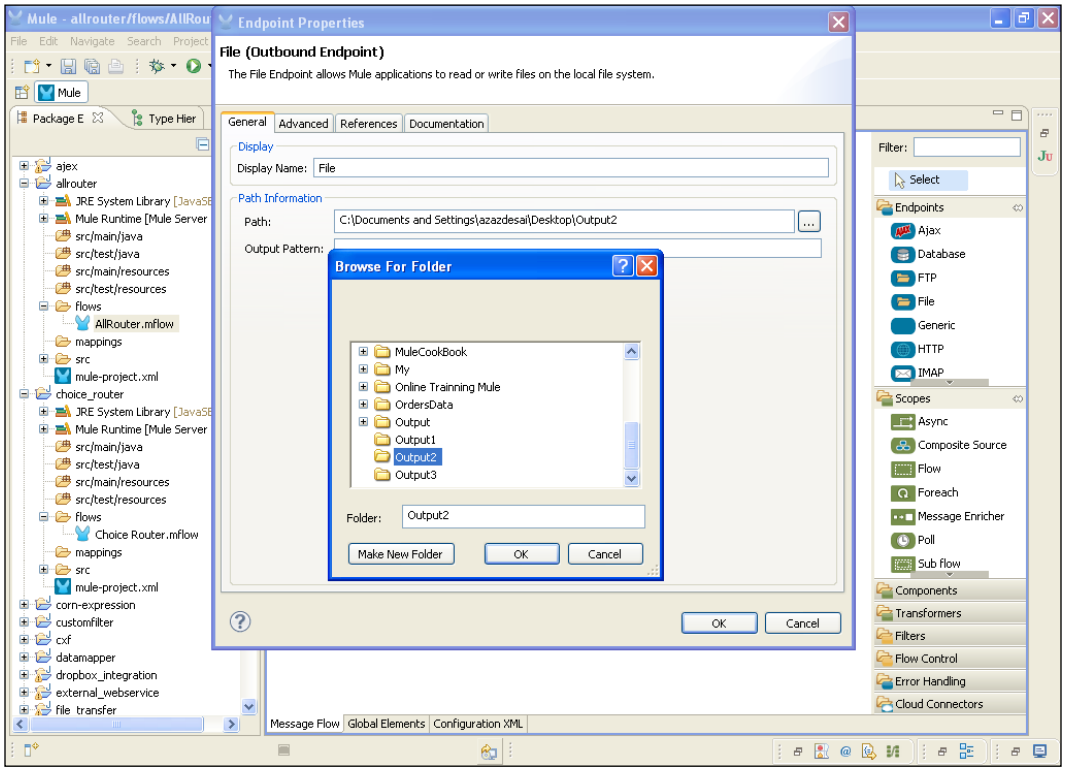
4. Drag the **File** Outbound Endpoint onto the canvas; the **File** Endpoint is chosen as the destination path.



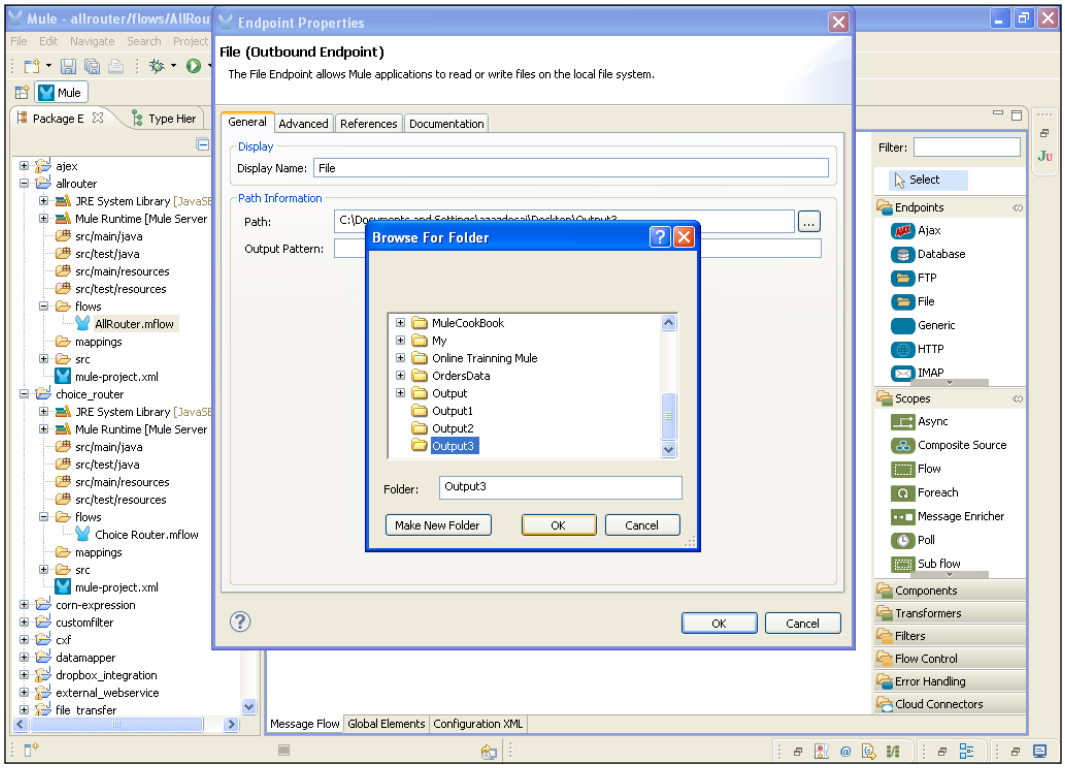
5. Double-click on the **File** Endpoint to configure it. Click on the **...** button and configure the destination path.



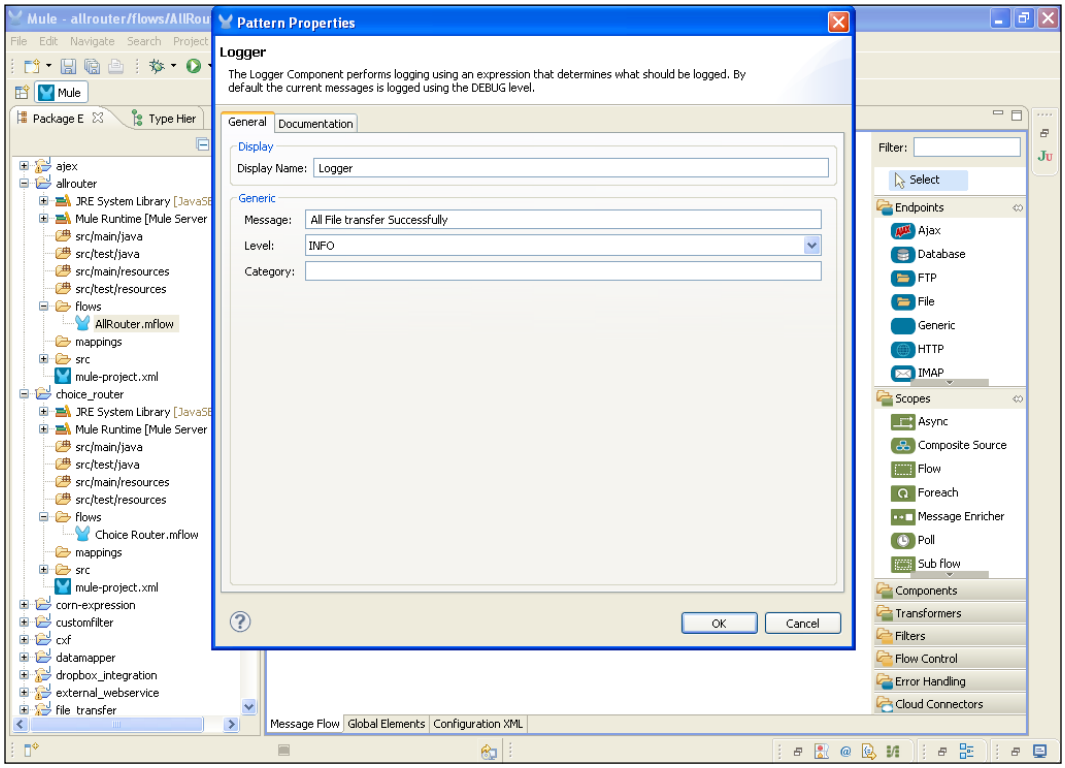
6. Drag the second **File** Outbound Endpoint onto the canvas. Double-click to configure it. Configure it in the same way as the previous one.



7. Drag the third **File** Endpoint. As you send the file to multiple targets, you use three **File** Outbound Endpoints. The file is sent to three different locations.



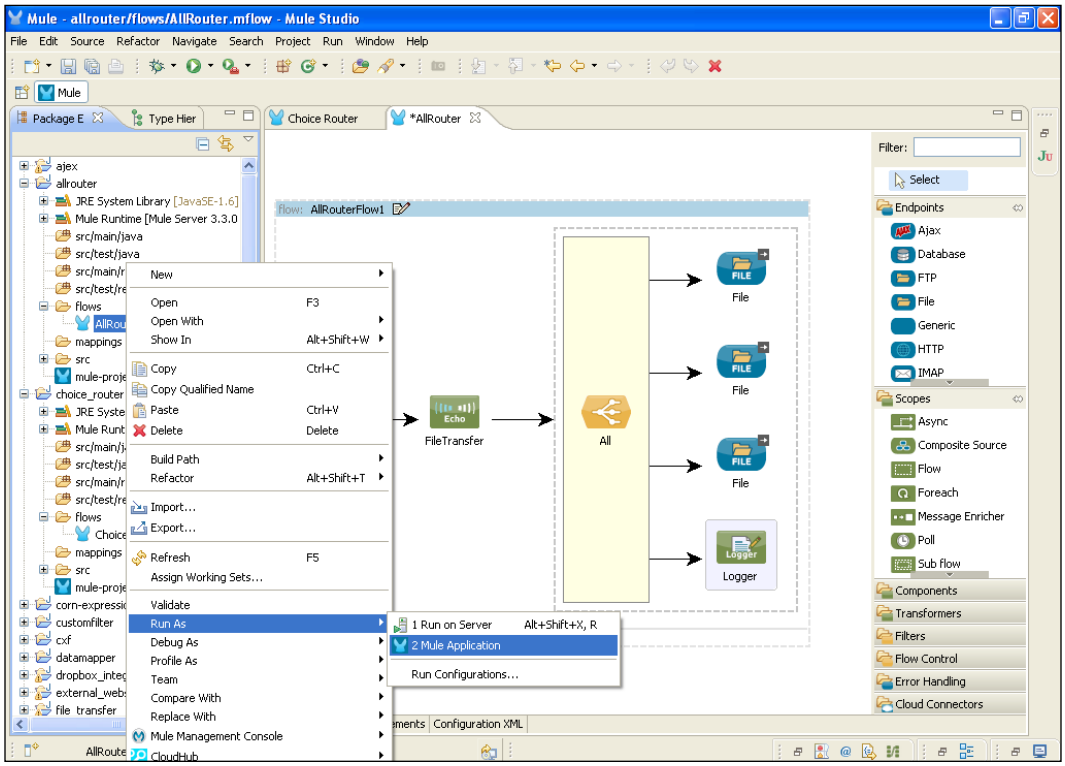
8. Drag the **Logger** component onto the canvas. Double-click on the **Logger** component to configure it. It's used for displaying a log on the console.



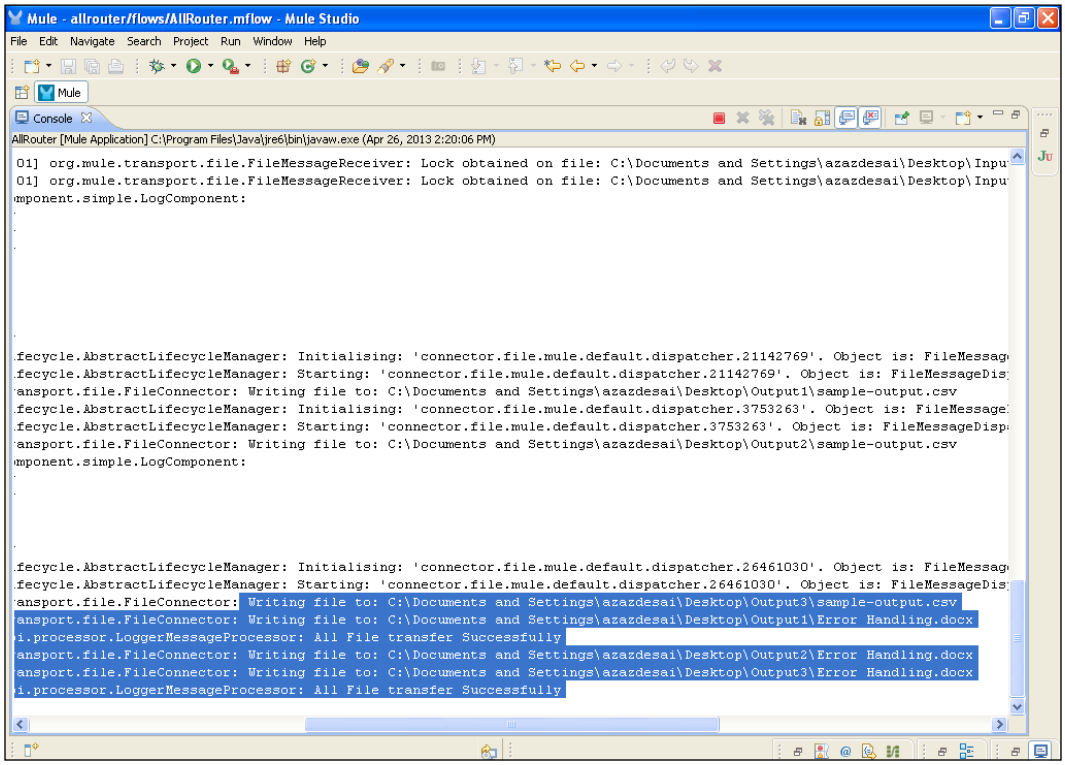
How it works...

In this section you will learn how to deploy the application.

1. To deploy the application code in the Mule server, go to **Run As** | **Mule Application**; the Mule server will deploy your application.



2. Once you have successfully deployed the application, you will see the following output on your console. You will also see a log on the console, which states that all the files are transferred to four different locations.



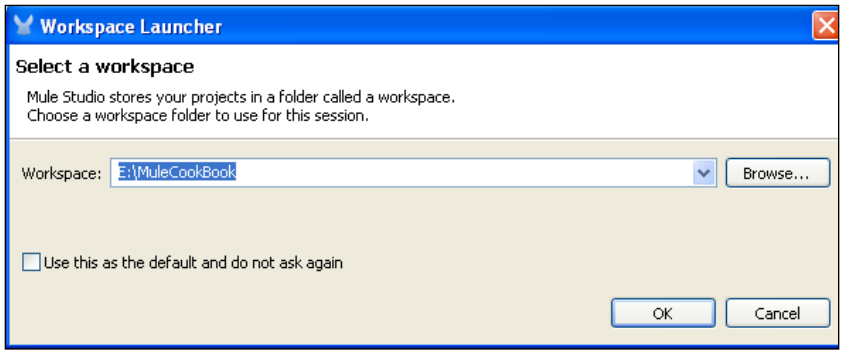
Configuring the Choice Router/Flow Control

Choice Router allows us to route a request to a specific path based on an expression. If the expression matches, it will move forward to the next Endpoint. In this recipe, you will see an example of how the Choice Router/Flow Control works.

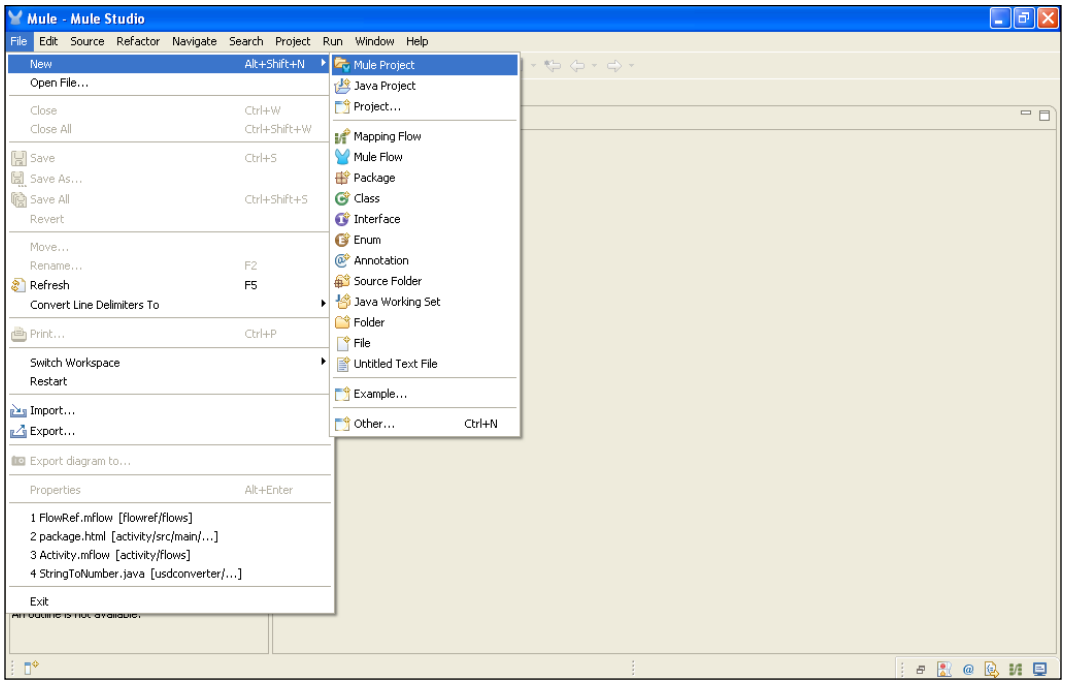
Getting ready

In this example, you will use the File Inbound Endpoint, Choice Router, and the File Outbound Endpoint.

1. Open Mule Studio and enter the workspace name as shown in the following screenshot:



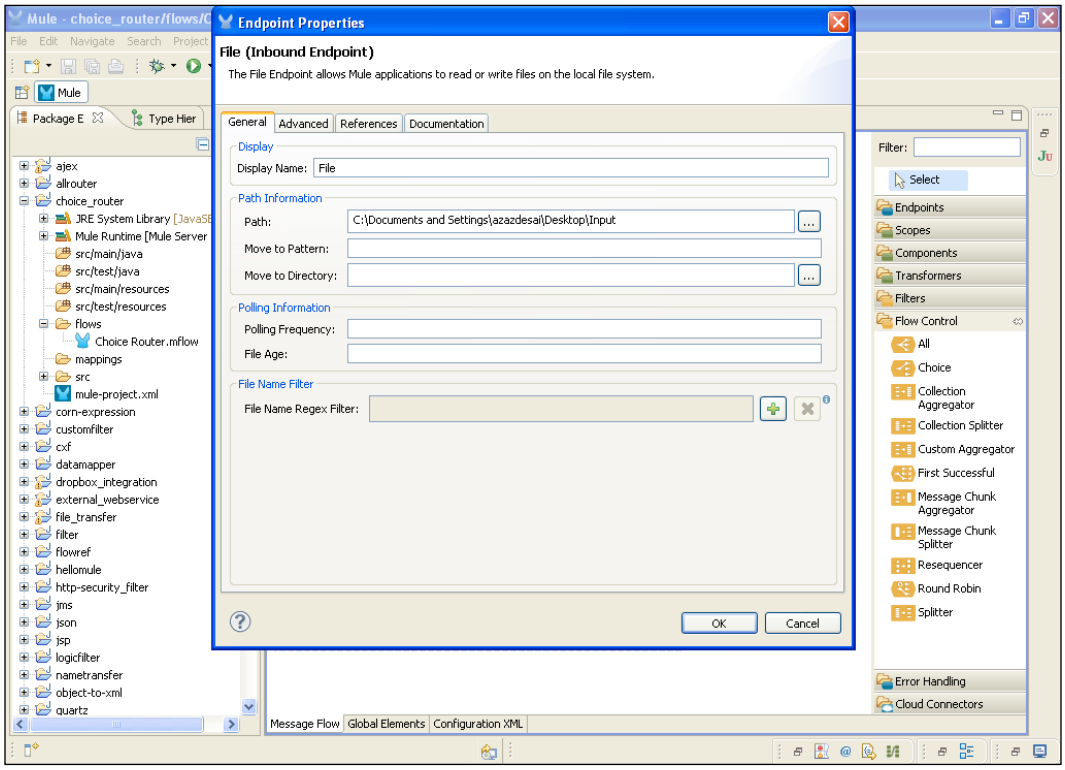
2. To create a new project, go to **File** | **New** | **Mule Project**. Enter the project name, `choice_router`, and click on **Next** and then on **Finish**. Your new project is created; you now have to start the implementation.



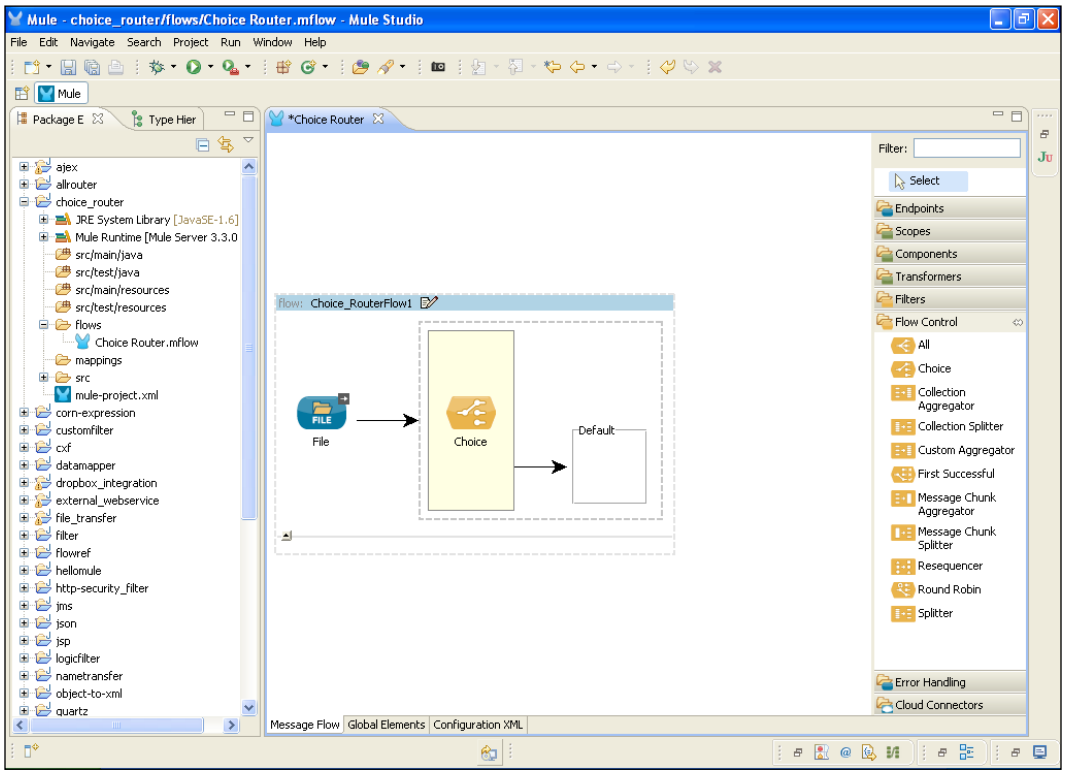
How to do it...

In this section, you will see how to configure the File Inbound Endpoint, Choice Router, and the File Outbound Endpoint.

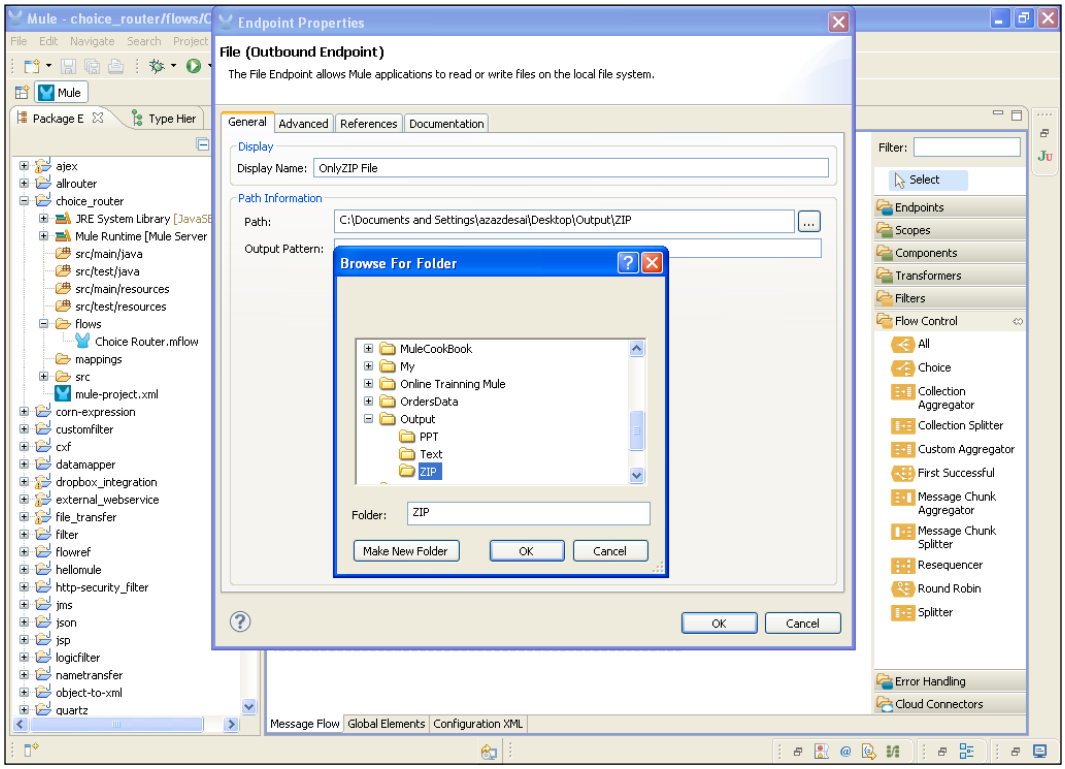
1. Go to the `Choice Router.mflow` file and drag the **File** Inbound Endpoint onto the canvas. Double-click and configure it, and provide the path.



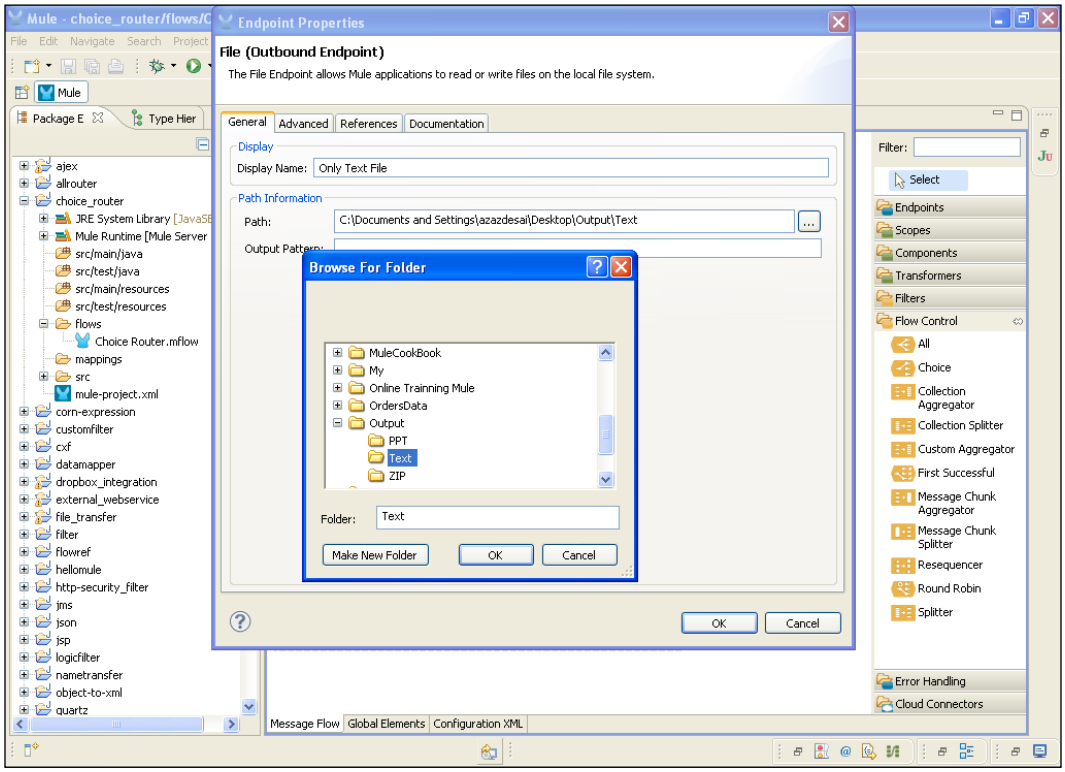
2. Drag the **Choice** Router onto the canvas. This Flow Control will be configured later.



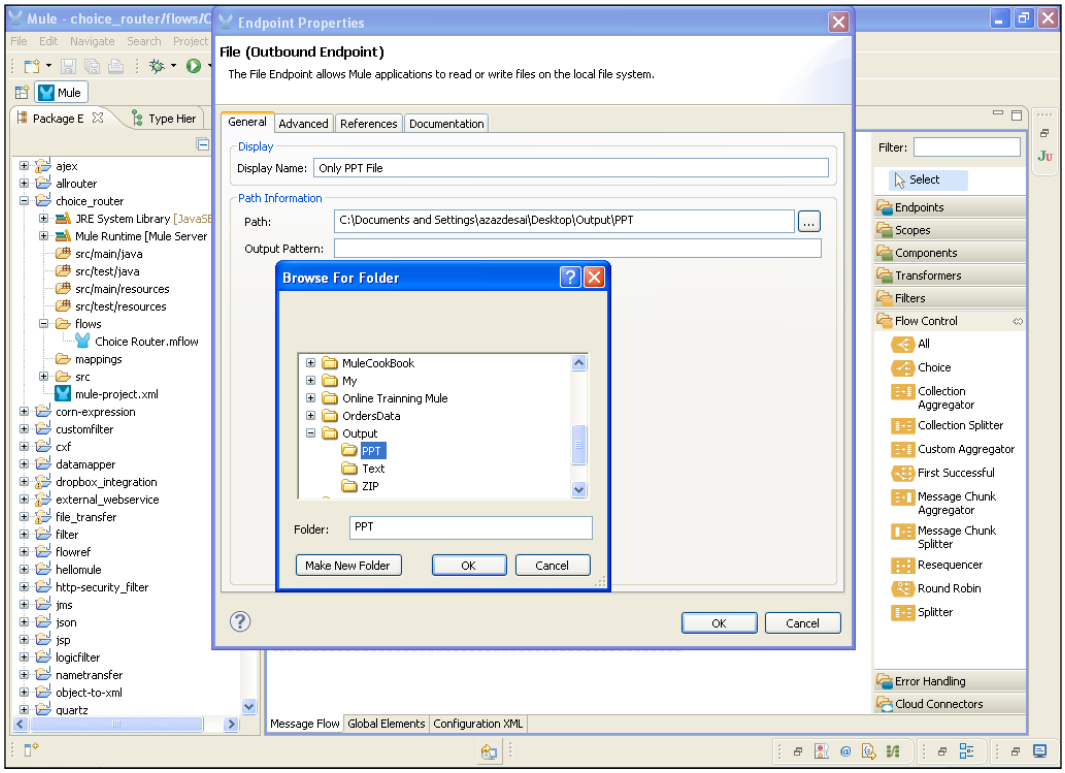
3. Drag the **File** Outbound Endpoint onto the canvas; double-click and configure it, and provide the path. You have to create a folder called `Output`. Inside the `Output` folder, you need to create three different folders: `Text`, `PPT`, and `ZIP`. Firstly, you have to provide the path for the `ZIP` folder.



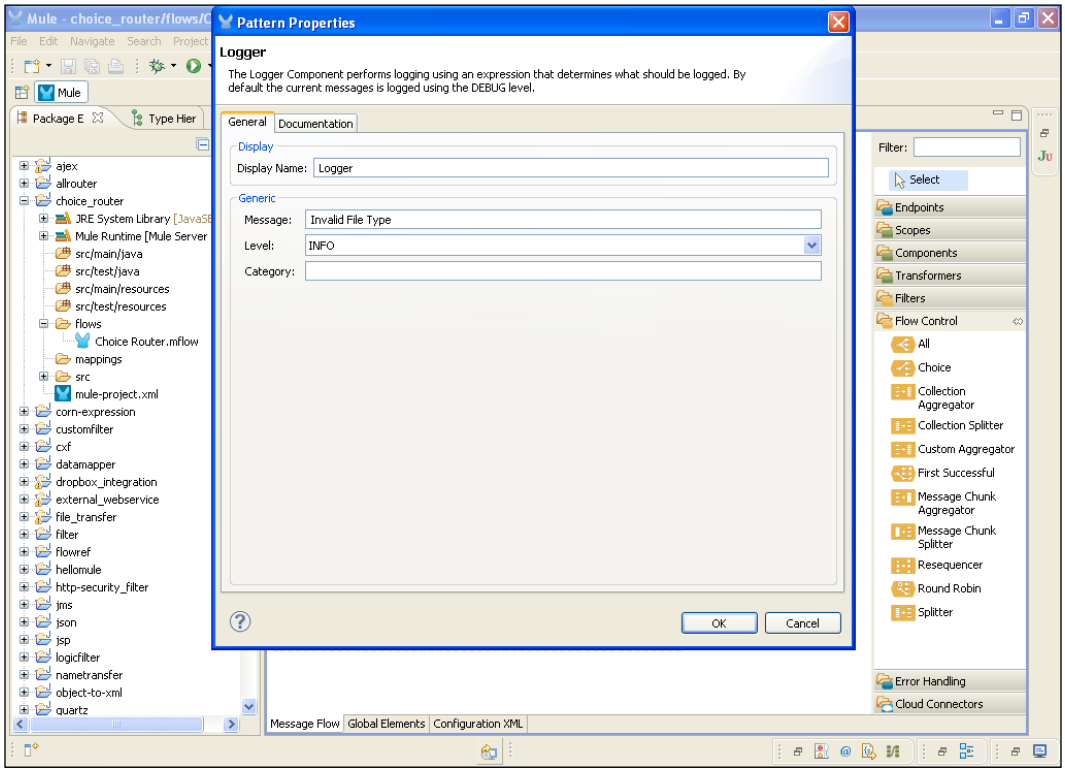
4. Drag the second **File** Outbound Endpoint onto the canvas. Configure it just like the previous one, but this time you have to provide the path for the `Text` folder.



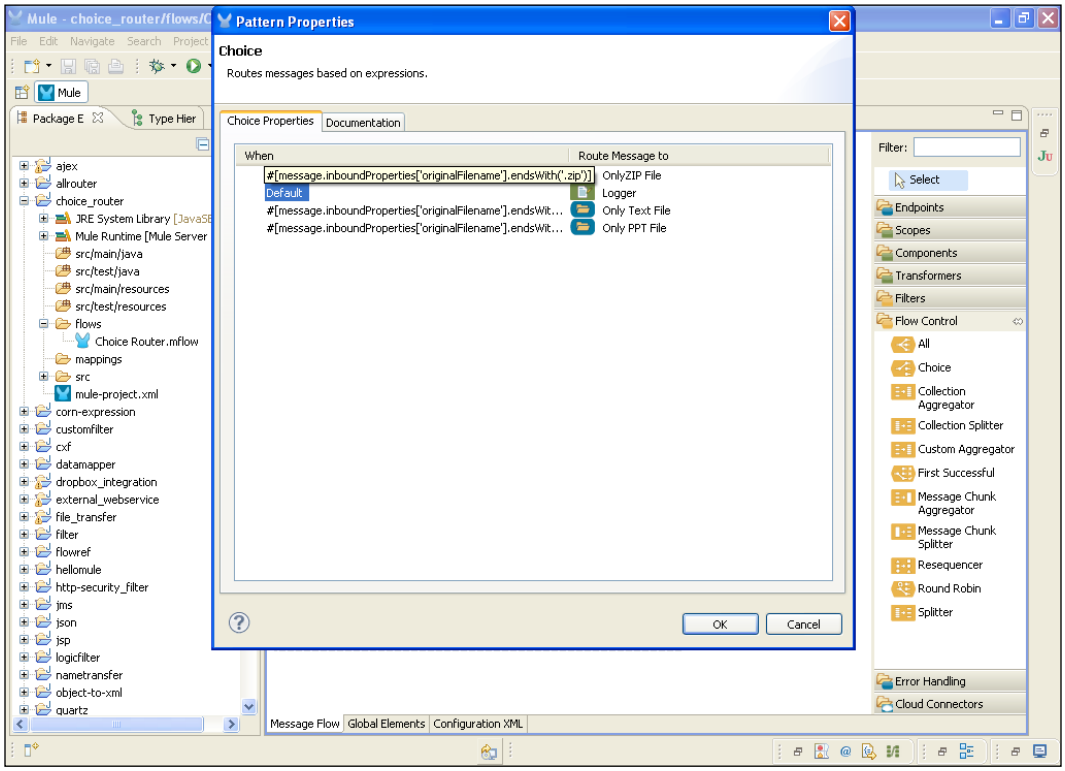
5. Drag the third **File** Output Endpoint onto the canvas. Configure it just as the first one and provide the path for the `PPT` folder.



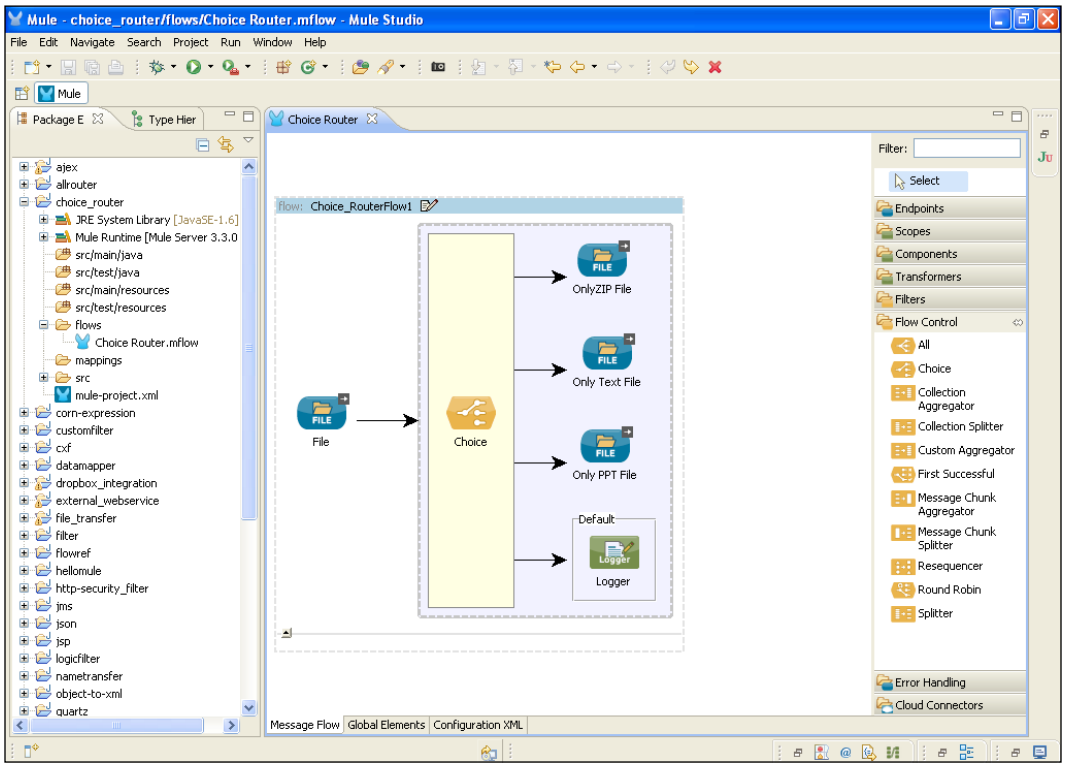
6. Drag the **Logger** component onto the canvas. Double-click and configure it. If an expression doesn't match, that file will be sent to the **Logger** component.



7. Double-click on the **Choice** Router to configure it. You will write an expression for all the files. The `#[message.inboundProperties['originalFilename'].endsWith('.zip')]` expression is for ZIP files. Through this expression, only ZIP files are transferred to the destination folder.



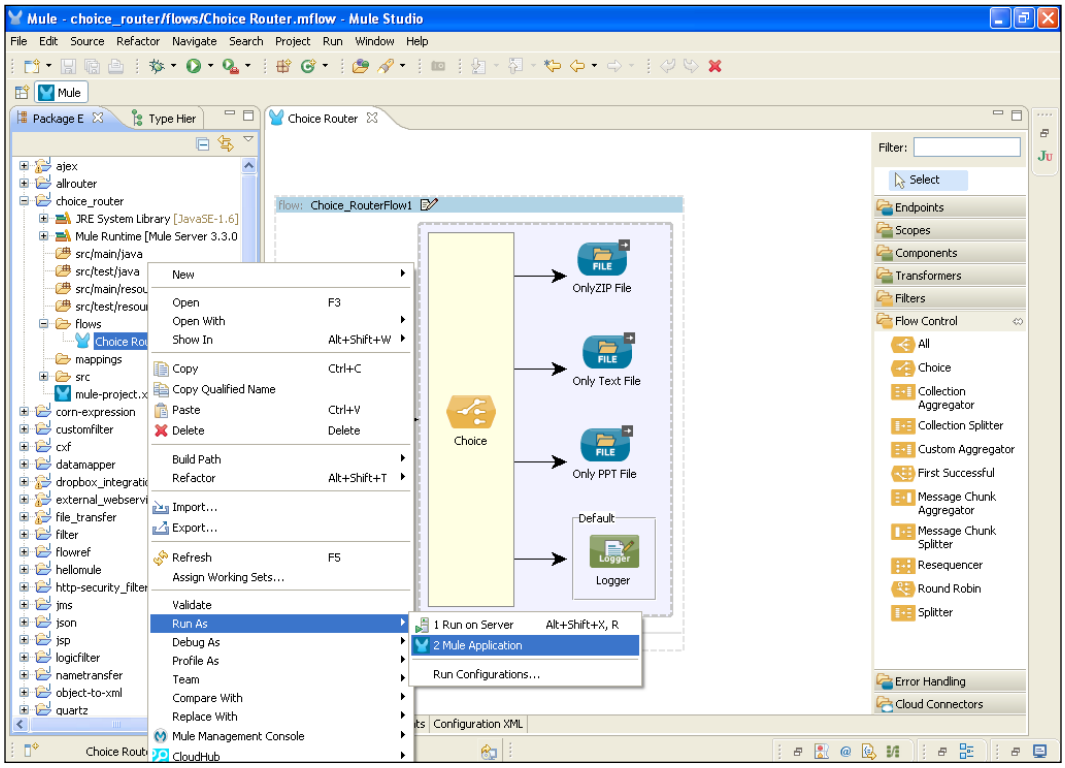
Your final flow will look like the following screenshot:



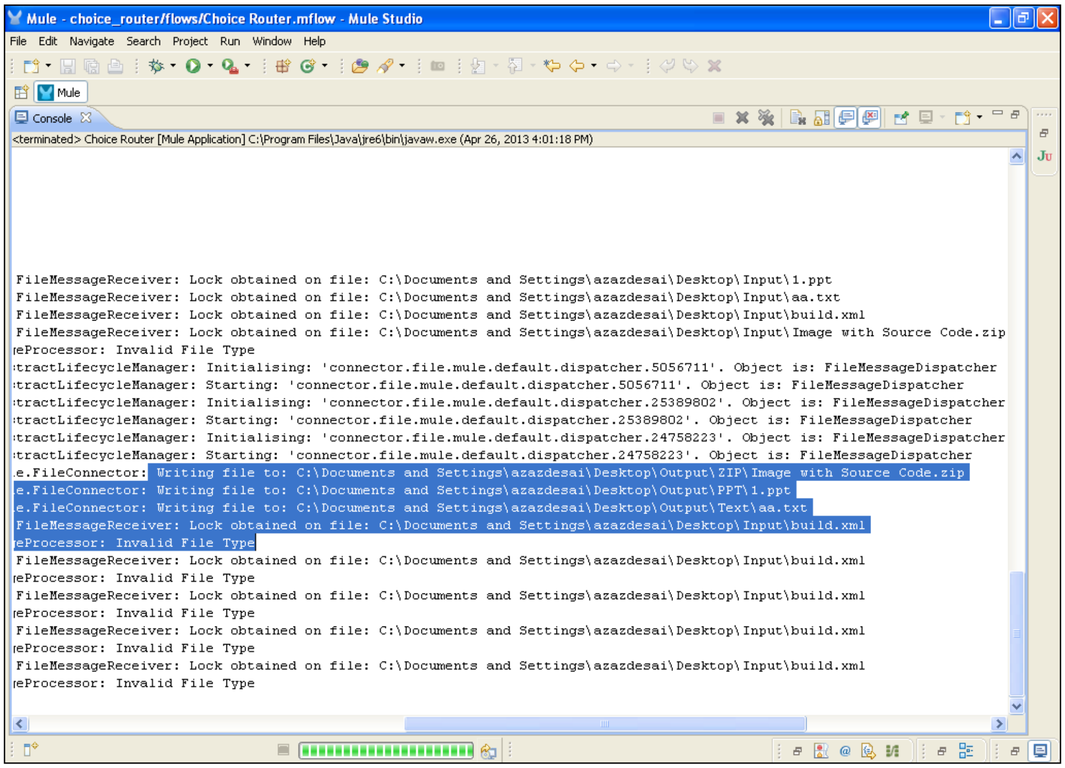
How it works...

In this section, you will get an idea of deploying an application in Mule Studio.

1. To deploy the application code in the Mule server, go to **Run As** | **Mule Application**; the Mule server will deploy your application. At the same time you should run the Tomcat server in Eclipse.



2. From the log on the console, you will see that all files are transferred to the specific folder. Only XML files are not transferred because we didn't use the XML file expression. That's why it shows `Invalid File Type` as the error description.



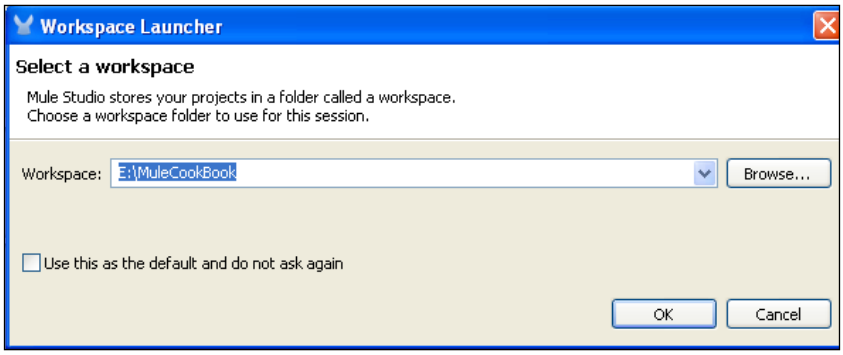
Configuring the Splitter Flow Control

A **Splitter** splits incoming messages into parts using the configured expression, which in turn is fed into the next message processor. In this recipe, you will see how to use the Splitter Flow Control.

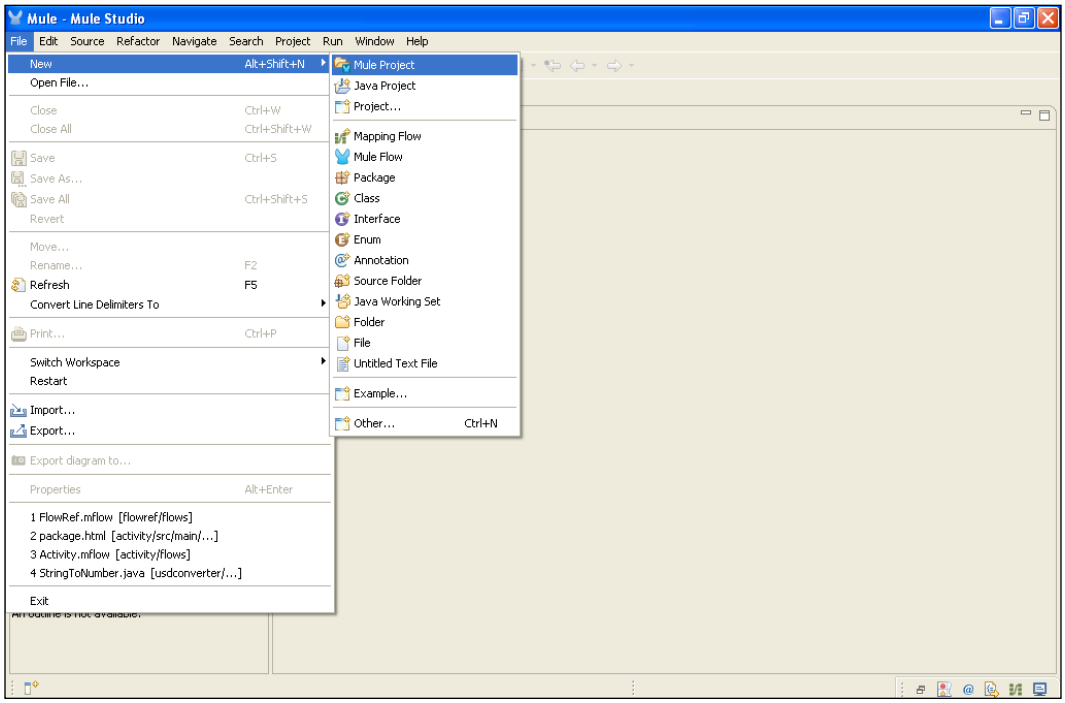
Getting ready

In this section, you will use the File Inbound Endpoint, the Splitter, and the File Outbound Endpoint.

1. Open Mule Studio and enter the workspace name as shown in the following screenshot:



2. To create a new project, go to **File** | **New** | **Mule Project**. Enter the project name, `splitter`, and click on **Next** and then on **Finish**. Your new project is created, so you can now start the implementation.



How to do it...

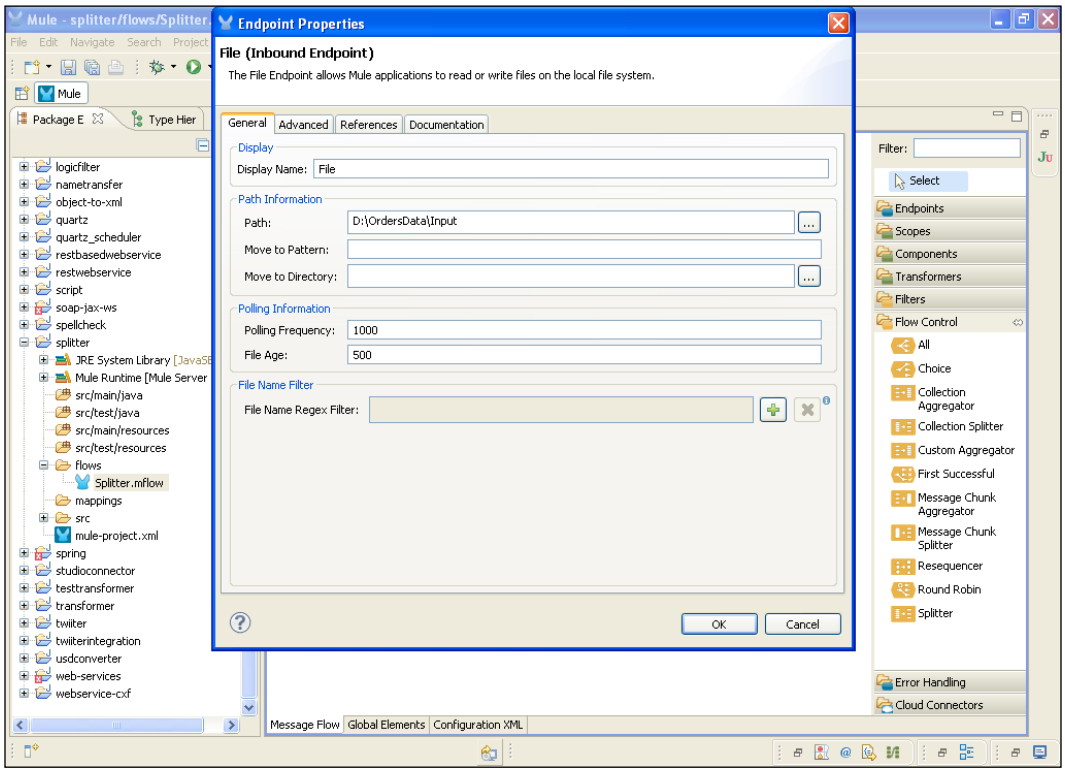
In this section, you will see how to configure Splitter, File Inbound Endpoint, and File Outbound Endpoint.

1. Go to the `splitter.mflow` file and drag the **File** Endpoint onto the canvas. Double-click and configure it, and then provide an XML file path. You split a city name in that XML file.

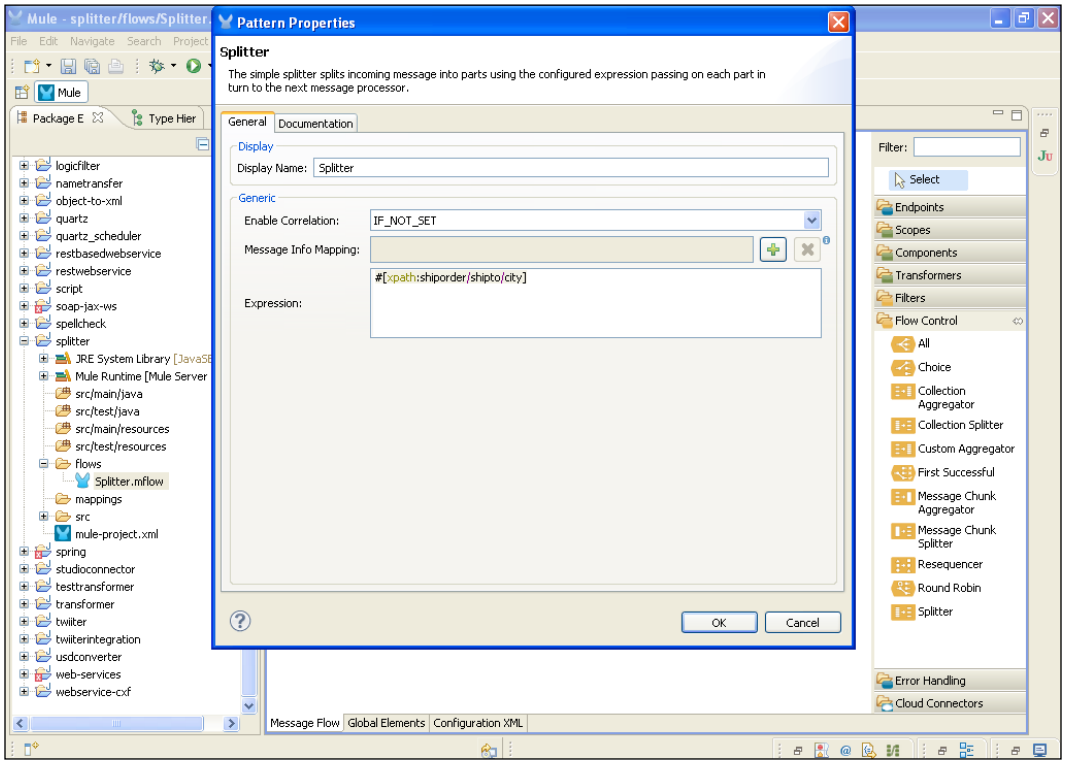
   The following code snippet is the `Shipping.xml` file:

```
<shiporder xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
orderid="555-66-7777">
  <orderperson>Derek Adams</orderperson>
  <shipto>
    <name>Azaz Desai</name>
    <address>123 Test Drive</address>
    <city>Ahmedabad</city>
    <country>India</country>
  </shipto>
  <item>
    <title>Laptop</title>
    <note>Some piece of Mac crap!</note>
    <quantity>1</quantity>
    <price>99.97</price>
  </item>
  <item>
    <title>Memory Chips</title>
    <note>1 GB</note>
    <quantity>4</quantity>
    <price>49.99</price>
  </item>
</shiporder>
```

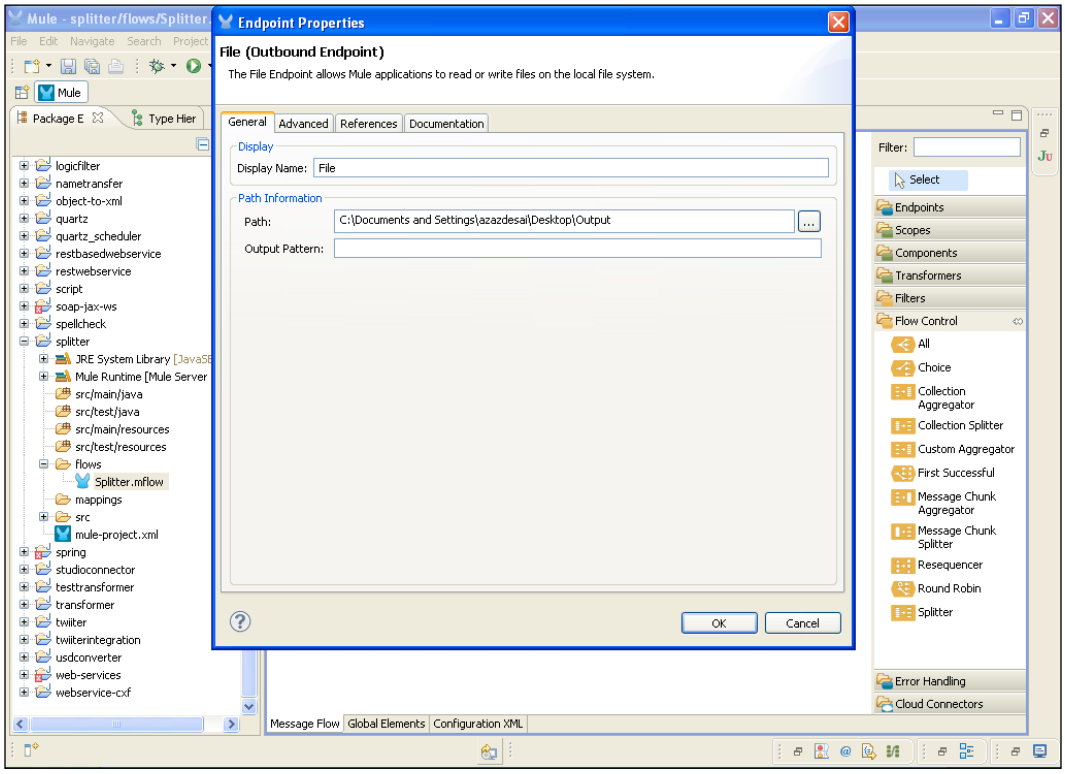
Here you configure the input folder path.



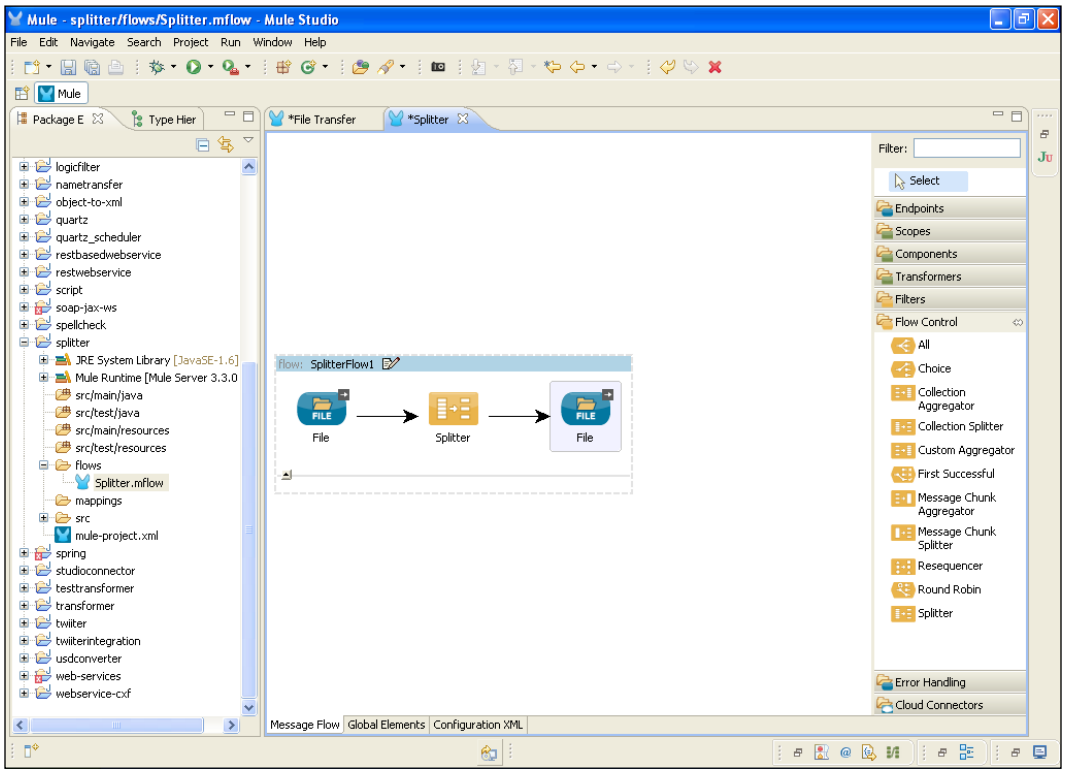
2. Drag the **Splitter** Flow Control onto the canvas. Double-click and configure it. To split the city name, you write an expression `#[xpath:shiporder/shipto/city]`.



3. Drag the **File** Outbound Endpoint on the canvas. Double-click and configure it, and provide a destination path.



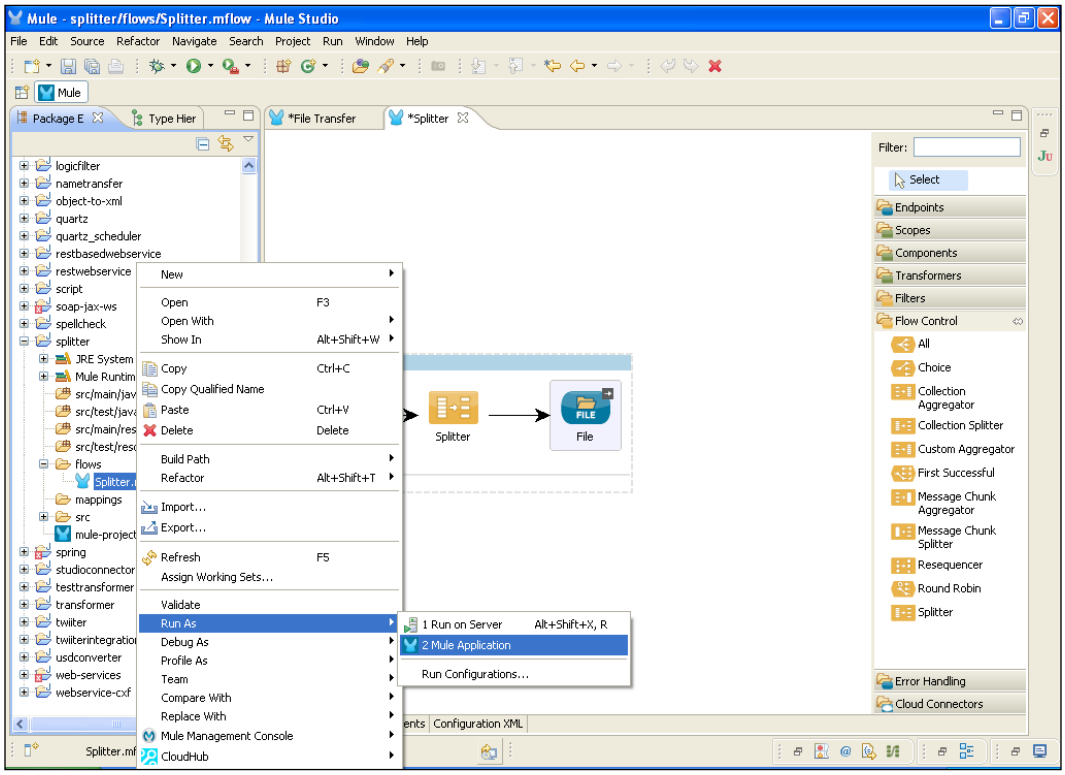
Your final flow will look similar to the one shown in the following screenshot:



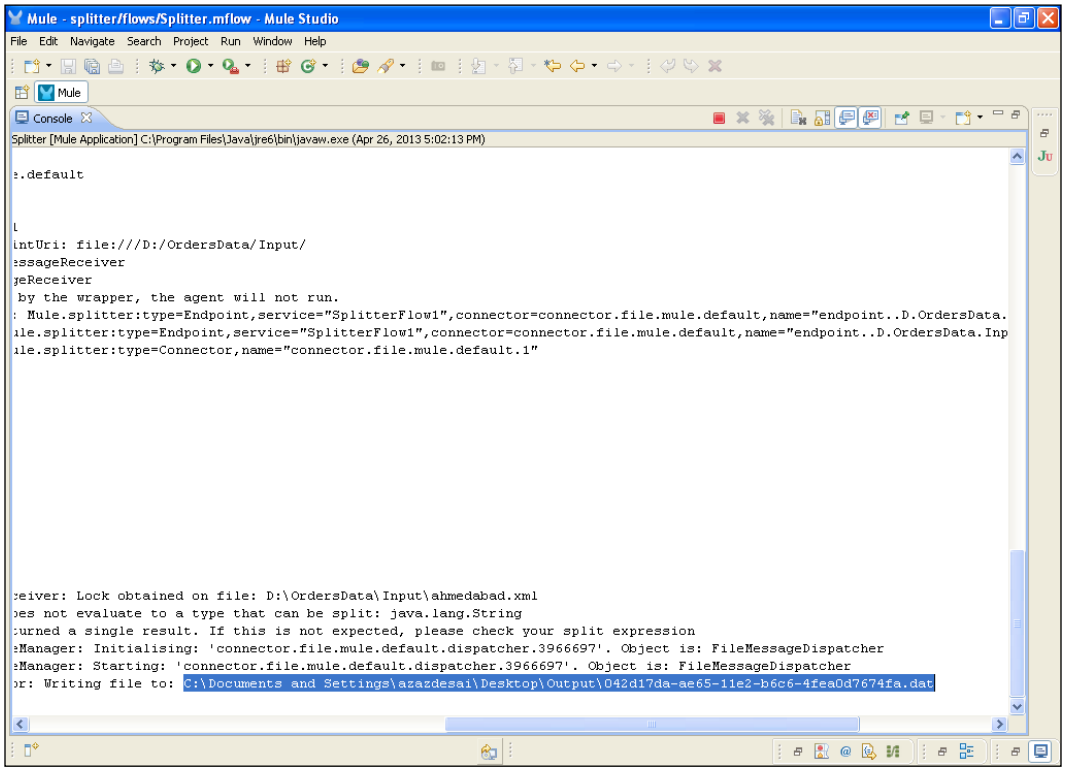
How it works...

In this section, you will see how to deploy the application using Mule Studio.

1. To deploy the application code in the Mule server, go to **Run As** | **Mule Application**; the Mule server will deploy your application.



2. Once you have successfully deployed the application, you will see the following output on your console. Use the path highlighted in the following screenshot to see that only the city name will be written into the file.



10
Configuring Cloud Connectors

In this chapter, you will learn what a Cloud Connector is. We will also look at the following recipes:

- Configuring the Twitter Cloud Connector
- Configuring the DropBoxIntegration folder

Introduction

Through Cloud Connectors, easy integration of your third-party web APIs is possible. Cloud Connectors are mainly used for integration purposes. In this chapter, you will see how to integrate third-party APIs through different connectors.

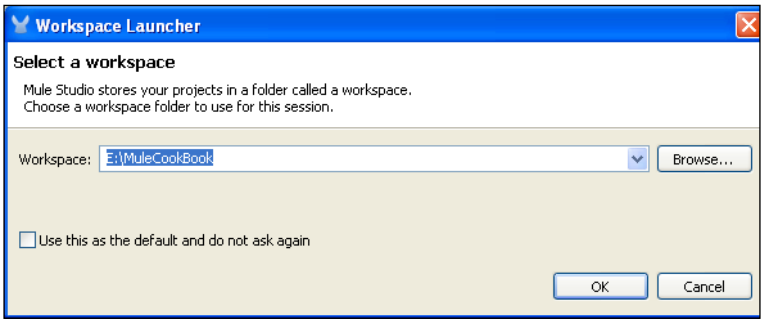
Configuring the Twitter Cloud Connector

The **Twitter Cloud Connector** is used for integrating the Twitter API. Through this API, you will perform different operations such as updating the status on your Twitter account, and you can retweet the message, search and show statuses. In this recipe, you will see how to send a tweet using Mule Studio.

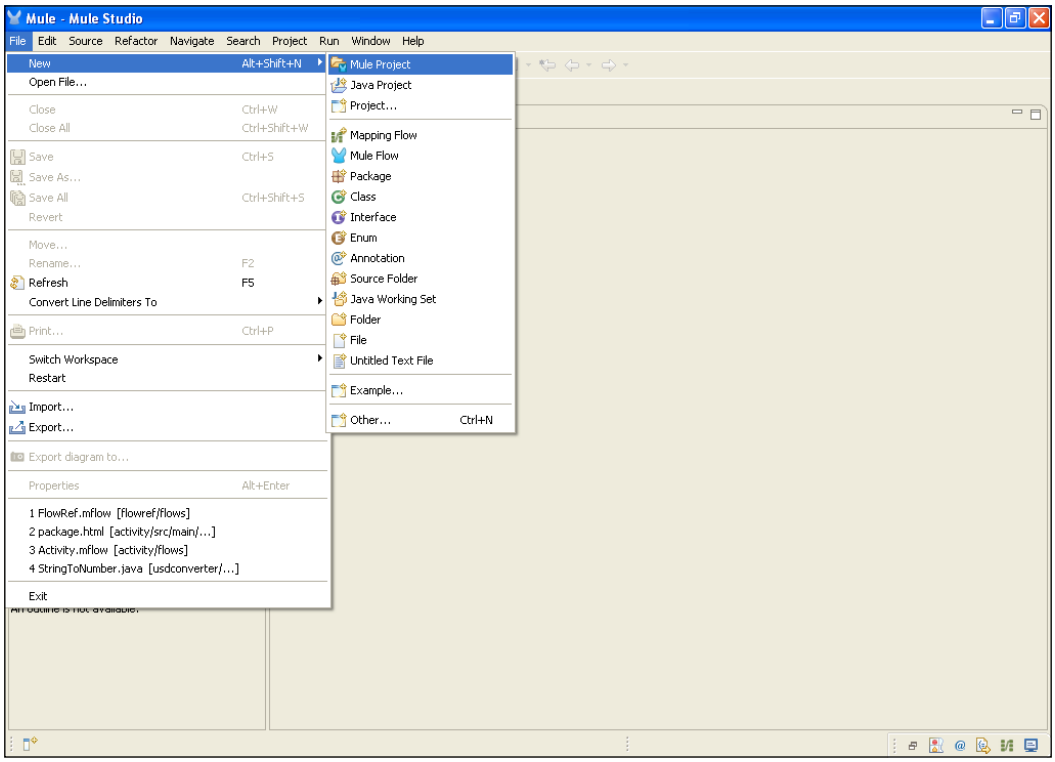
Getting ready

In this example, you will use the HTTP Endpoint and the Twitter Cloud Connector.

1. Open Mule Studio and enter the workspace name as shown in the following screenshot:



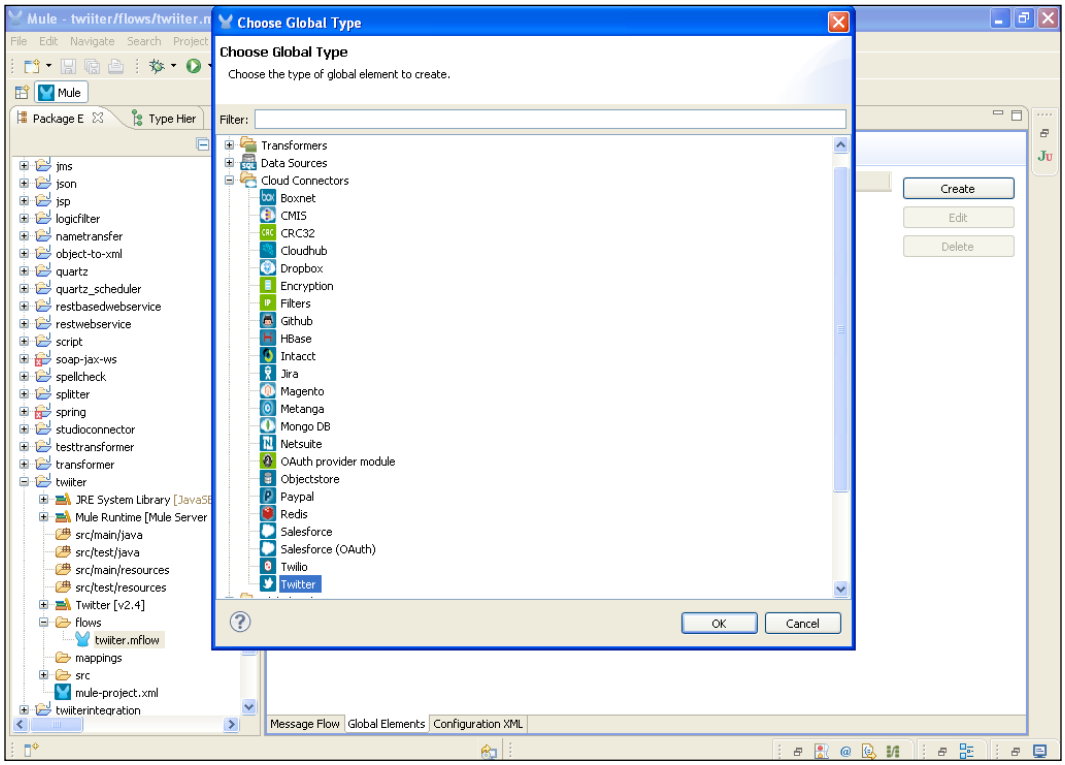
2. To create a new project, go to **File** | **New** | **Mule Project**. Enter the project name, `Twitter`, click on **Next** and then on **Finish**. Your new project is created, and you now have to start the implementation.



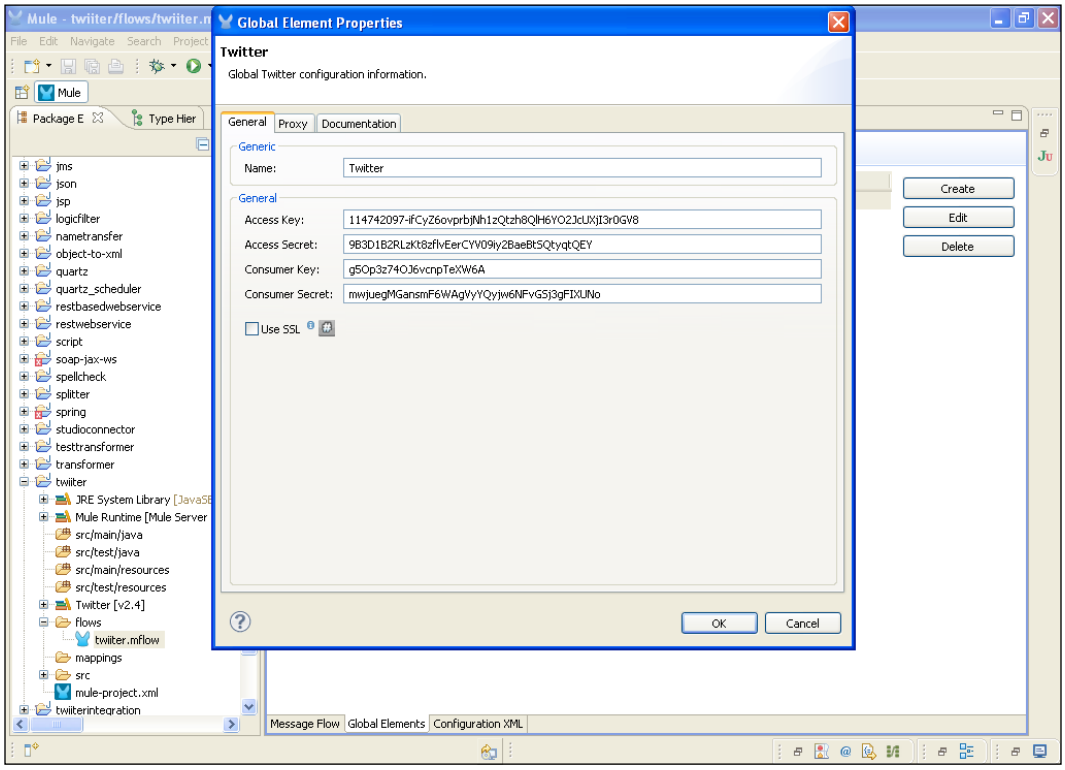
How to do it...

In this section, you will see how to configure the Twitter Cloud Connector and how to use it in a flow.

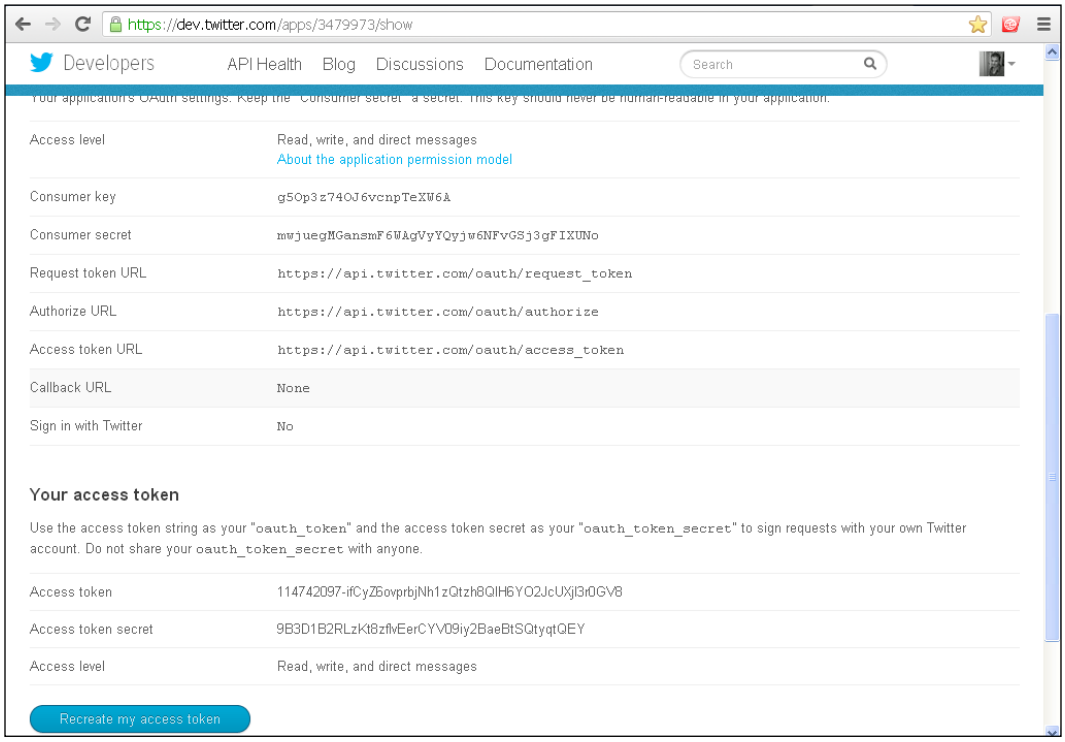
1. Go to the `twitter.mflow` file and click on the **Global Elements** tab. Go to **Cloud Connectors** | **Twitter**.



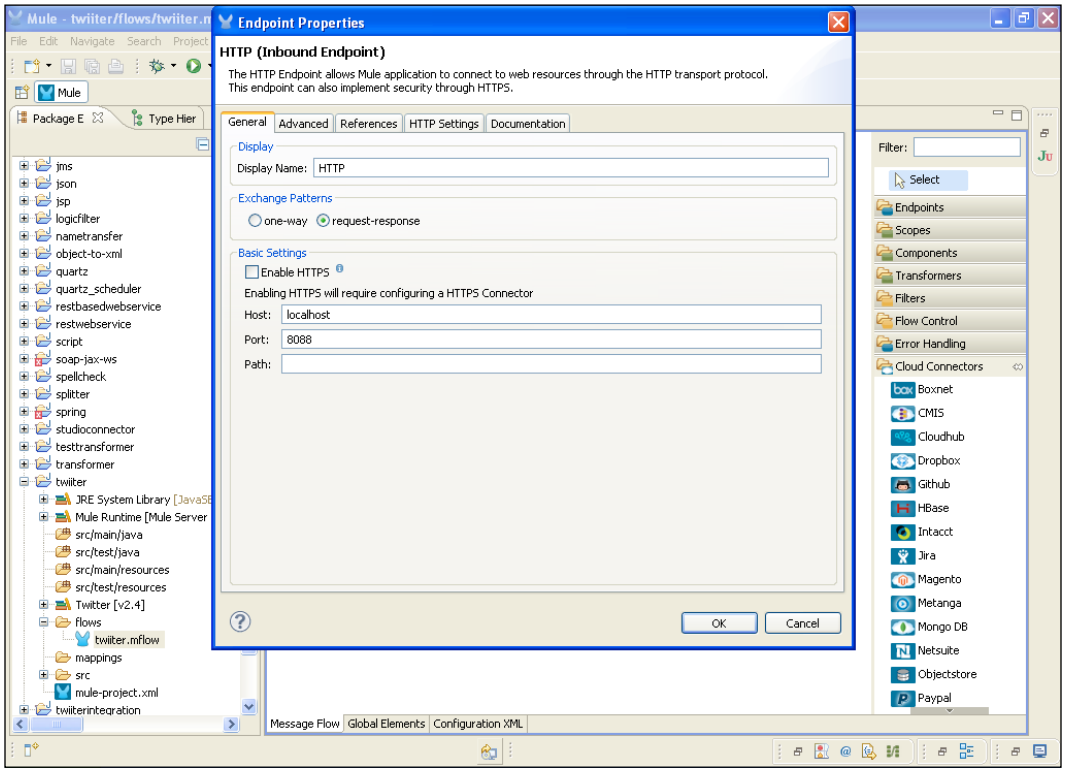
2. Once you click on the **Twitter** Cloud Connector, you will see a similar screen on your system. Here, you generate the values for the **Access Key:**, **Access Secret:**, **Consumer Key:**, and **Consumer Secret:** fields.



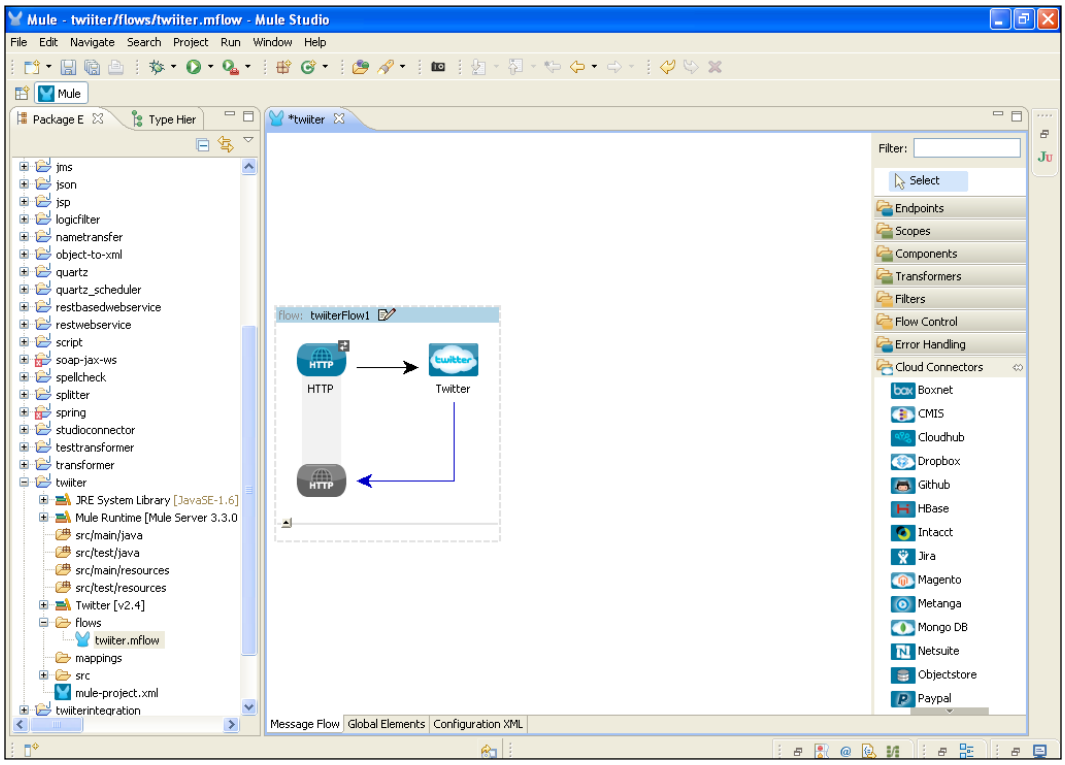
3. Go to `https://dev.twitter.com/apps`. Here you can click on **Create application**. Then you have to fill up the application form. Once you create the application, you will find a key in that application. Paste that key into your **Twitter** Cloud Connector.



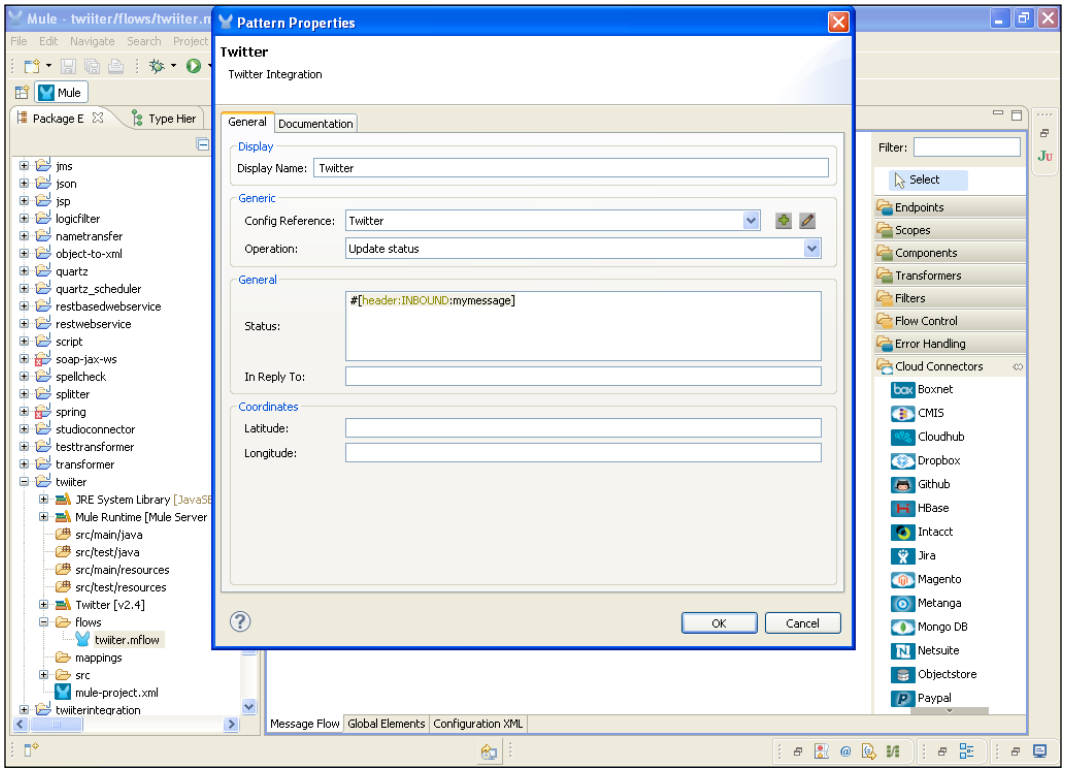
4. Click on the **Message Flow** tab and drag the **HTTP** Endpoint onto the canvas. To configure the Endpoint, double-click on it. Enter the port number.



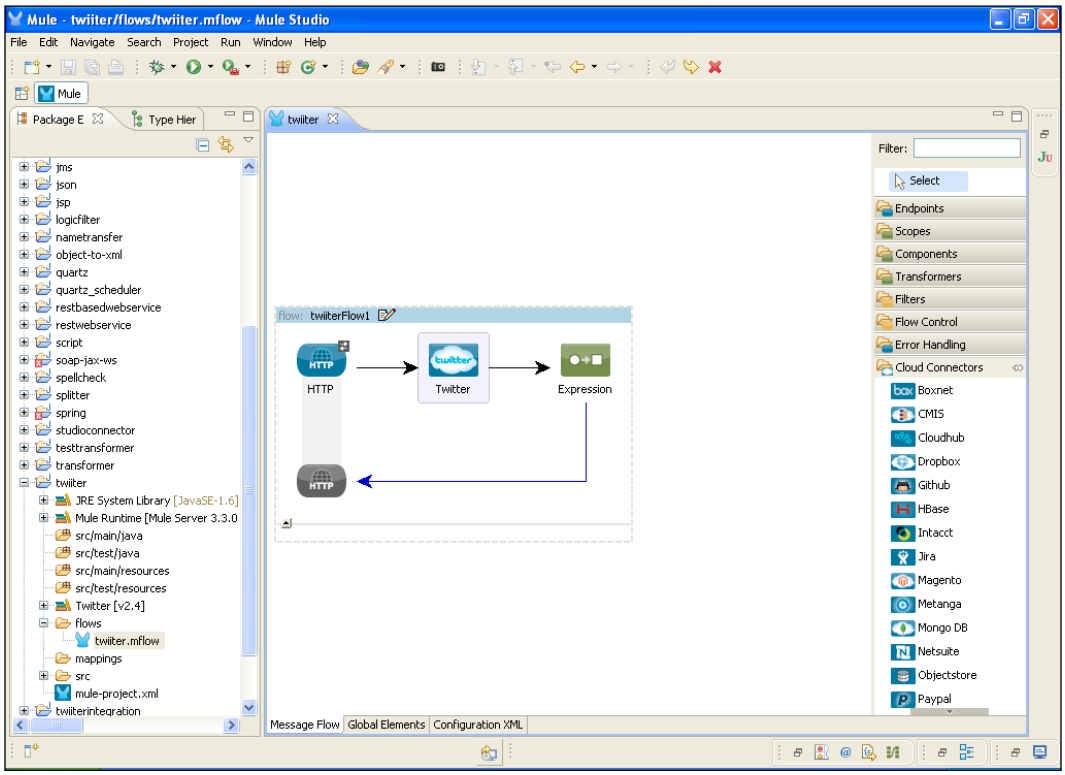
5. Drag the **Twitter** Cloud Connector onto the canvas. To configure it, double-click on it.



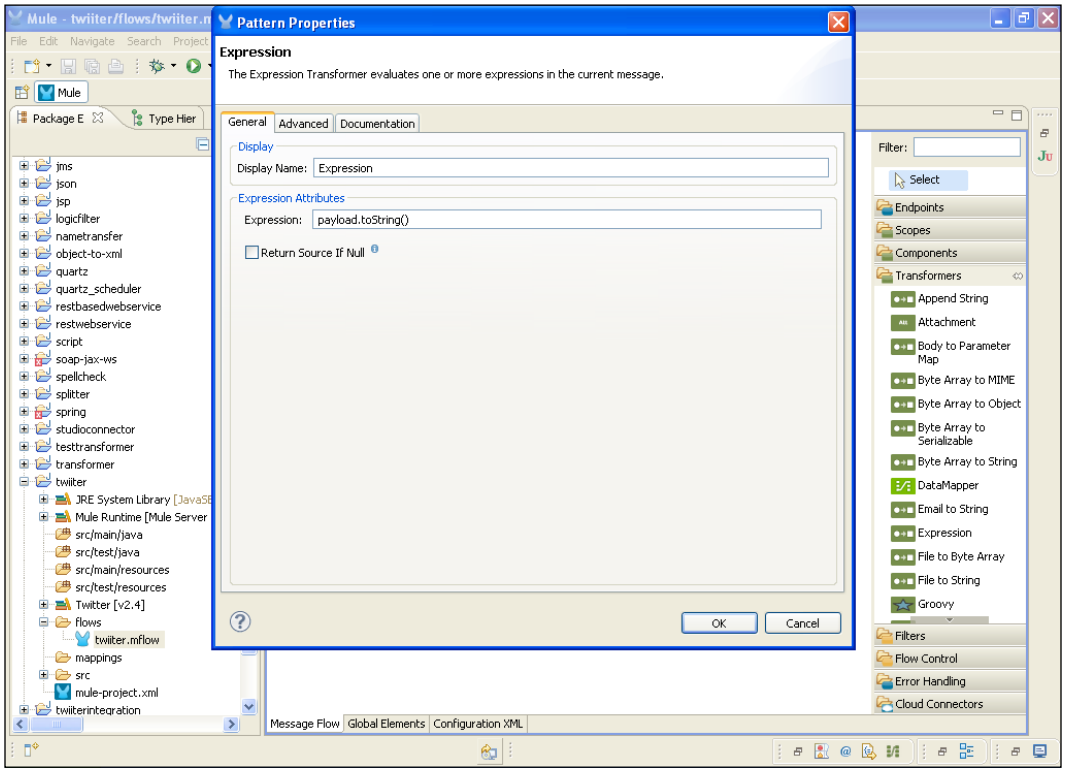
6. Once you click on the Connector, you will see a similar screen, as shown in the following screenshot, on your system. You will select the configuration reference name that was created before in the **Global Elements** tab—`Twitter`. Then, you select operation **Update Status**. In the **Status:** textbox, write an expression to send a tweet to your account, `#[header:INBOUND:mymessage]`.



7. Drag the **Expression** transformer; it will convert a payload into a string.



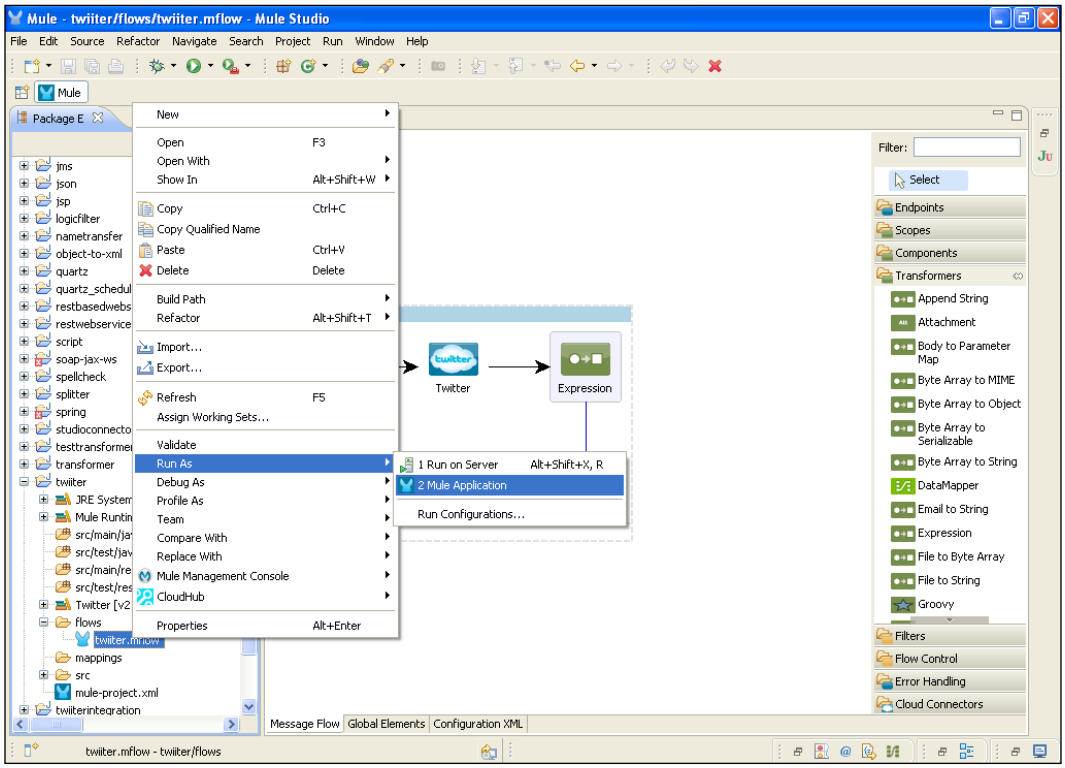
8. Double-click on the **Expression** transformer to configure it. Write an expression to convert it into a string, `payload.toString()`.



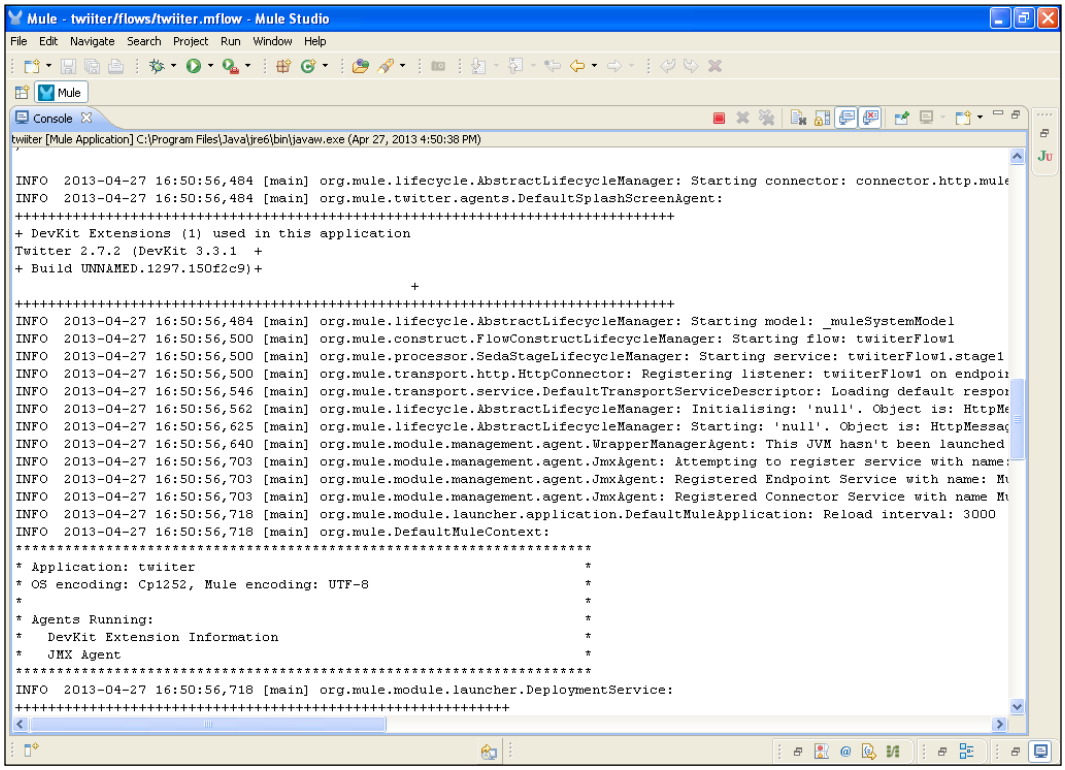
How it works...

In this section, you will have a look at how to deploy the application in Mule Studio.

1. To deploy the application code in the Mule server, go to **Run As** | **Mule Application**; the Mule server will deploy your application.



2. Once you have successfully deployed the application, you will see the following output on your console. You will see a log on the console, which states that all files are transferred to different locations.



3. Open your browser and paste the URL in it `http://localhost:8088/?mymessage=HelloMule`. You will receive a JSON output. When you receive a similar output on your browser (as shown in the following screenshot), your message will be successfully sent to your Twitter account:

4. You can now see the message displayed on your Twitter account:

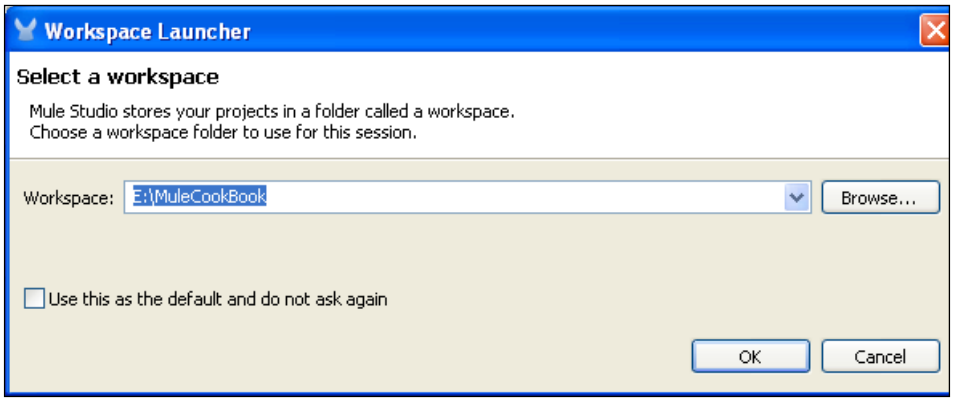
Configuring the DropBoxIntegration folder

Dropbox is a file hosting service operated by Dropbox Inc. It is a free service that lets you bring your photos, documents, and videos anywhere and share them easily. Through Dropbox Connectors, you can access the Dropbox API and you can also insert, upload, download, and delete files and folders.

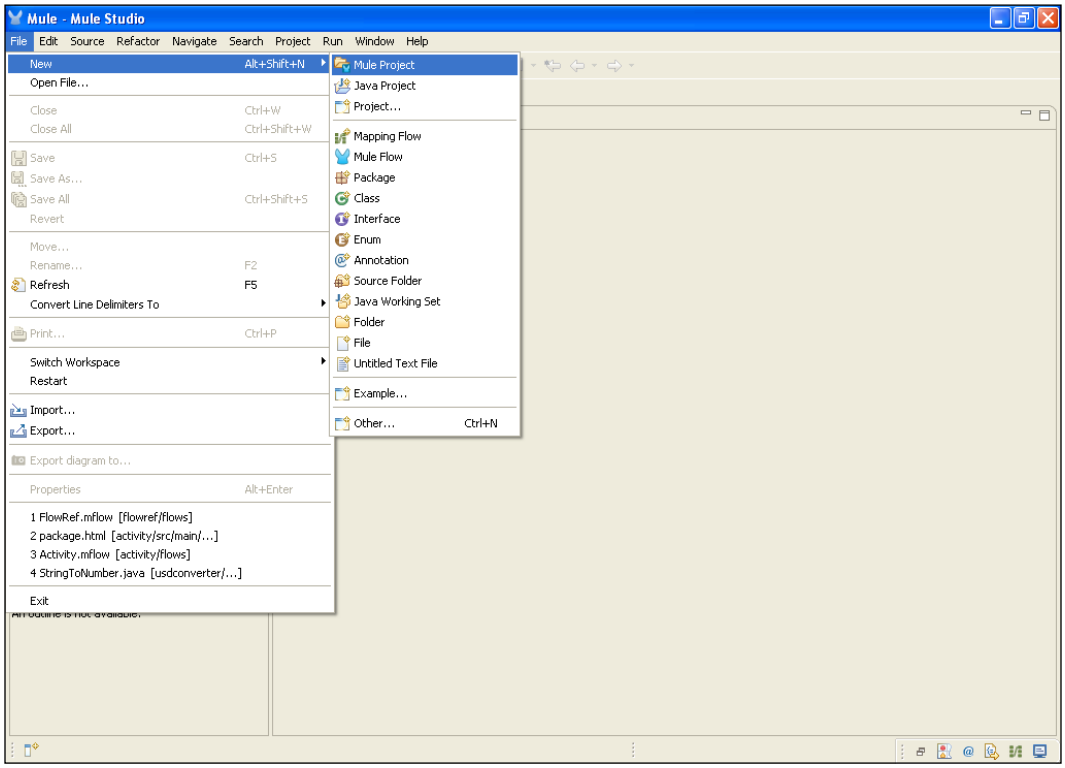
Getting ready

In this example, we will use the HTTP Endpoint, the Dropbox Connector, and the Choice Router. Create a new project by performing the following steps:

1. Open Mule Studio and enter the workspace name as shown in the following screenshot:



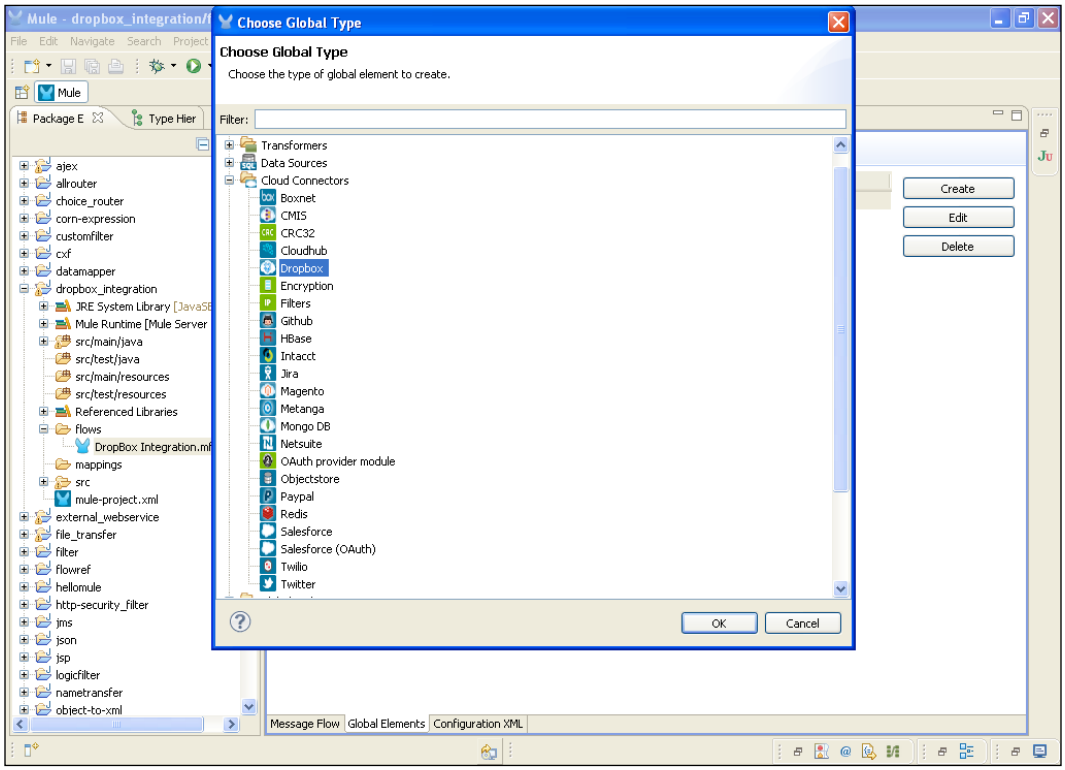
2. To create a new project, go to **File** | **New** | **Mule Project**. Enter the project name, `Dropbox_Integration`, click on **Next** and then on **Finish**. Your new project is created, and you now have to start the implementation.



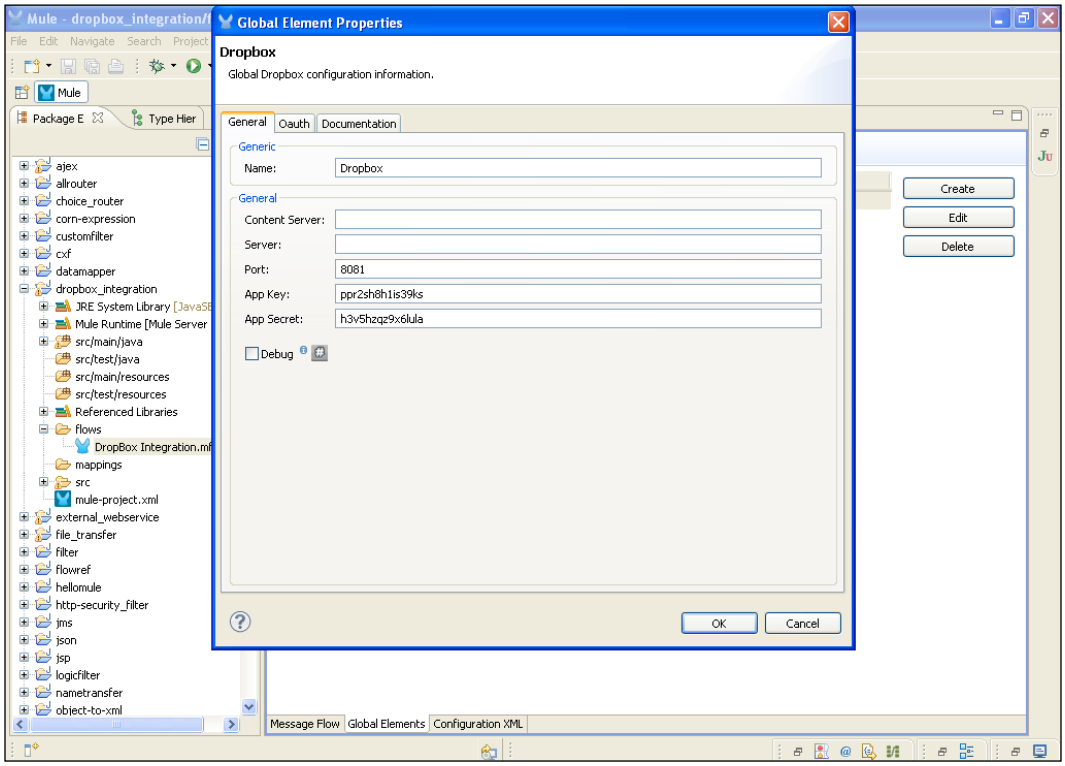
How to do it...

In this section, you will see how to configure the Dropbox Connector and how to use it in a flow.

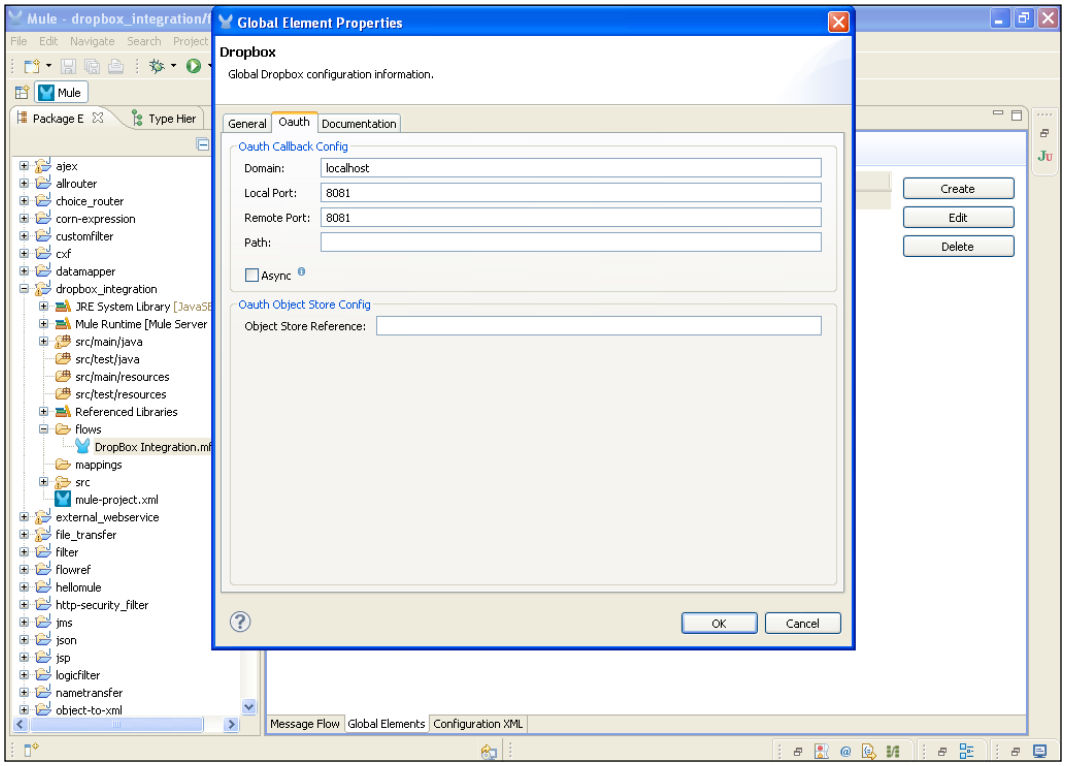
1. Go to the `Dropbox_Integration.mflow` file. Click on the **Global Elements** tab and go to **Cloud Connectors** | **Dropbox**.



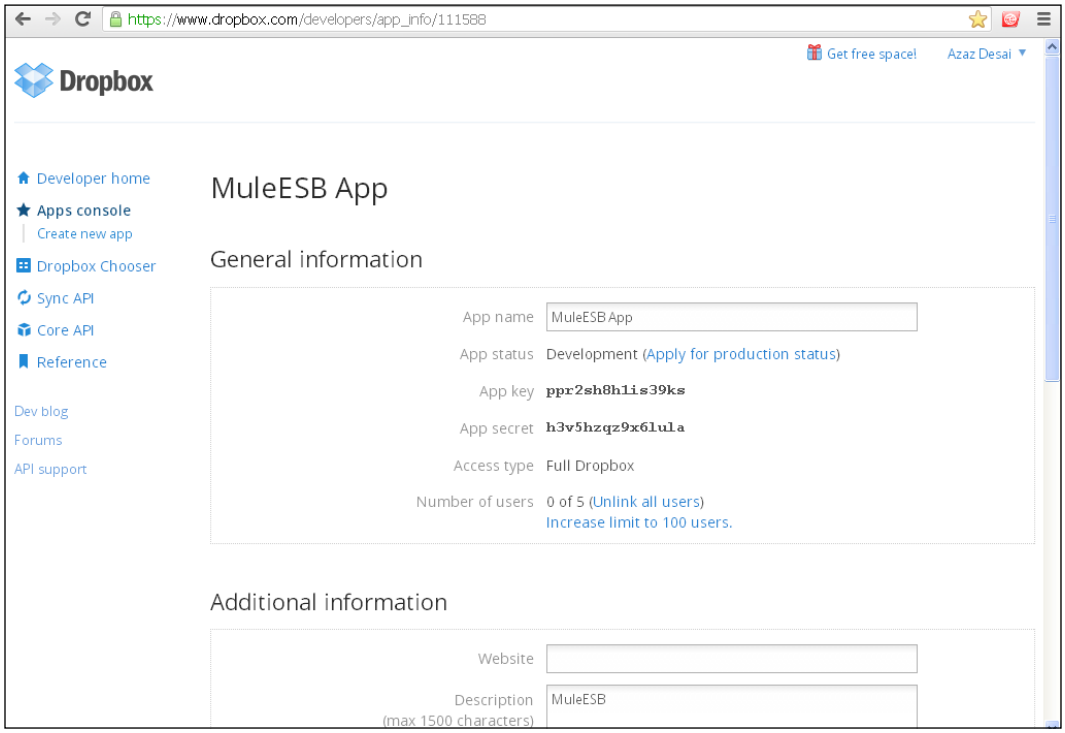
2. Configure the **Dropbox** Connector; here, you add the values for **Port:**, **App Key:**, and **App Secret:** fields.



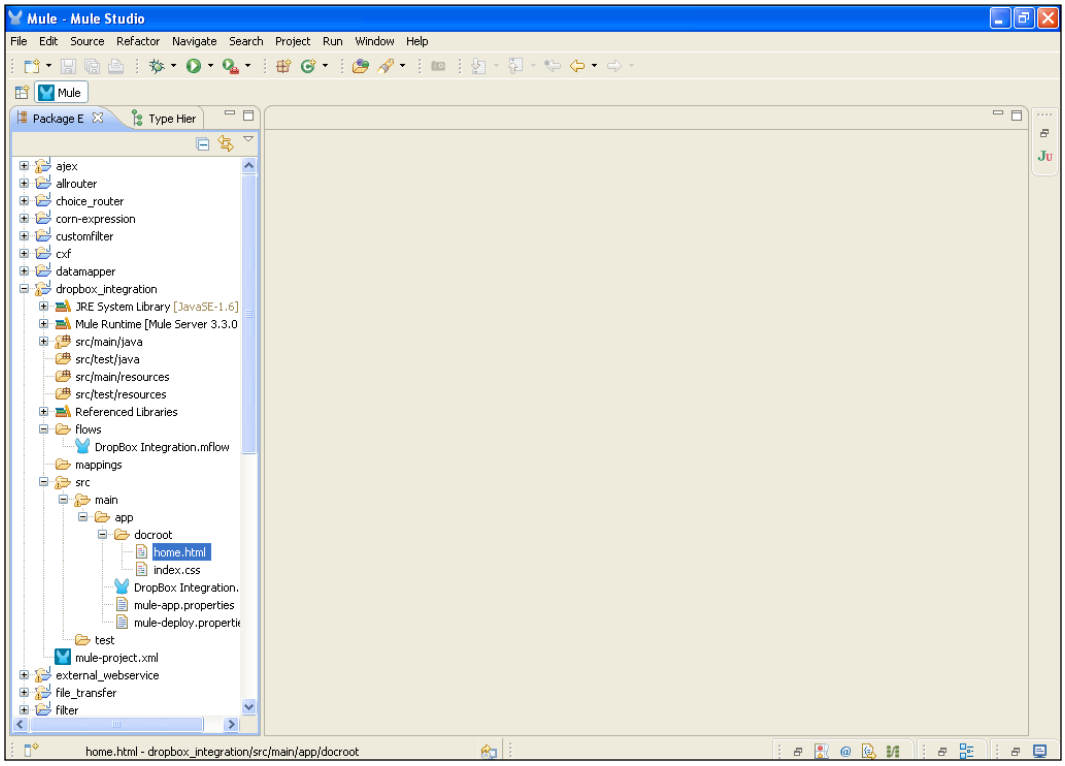
3. Click on the **Oauth** tab. Here, you can fill in the **Domain:**, **Local Port:**, and **Remote Port:** fields.



4. To generate the values of the **App Key:** and **App Secret:** fields, go to the URL `https://www.dropbox.com/developers`. Click on **Create new app**. Once you have created an app, you will see the app key and the app secret.



5. Go to `src/main/app` and create a `docroot` folder. Inside this folder create two files: `home.html` and `index.css`.



6. In the `docroot` folder, click on the `home.html` file. The following code snippet is present inside the `home.html` file:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<link rel="stylesheet" type="text/css" href="index.css" />
<meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
<title>Insert title here</title>
</head>
<body>

  <script type="text/javascript">

  function processElements ( elements, style){
```

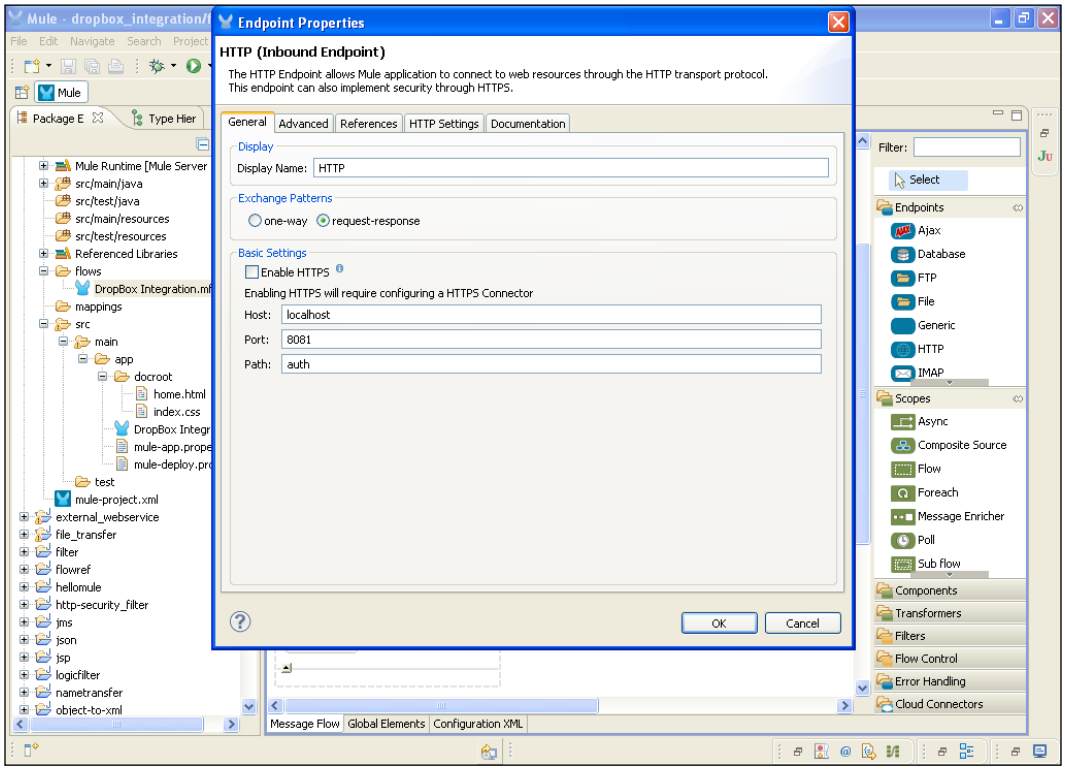
```
    for ( var i = 0; i < elements.length; i++){
      elements[i].style.display=style;
    }
  }

  function updateOptions(value){

  var hideElements = document.getElementsByClassName('hidden');
  var showElements = document.getElementsByClassName(value);

  processElements(hideElements, 'none');
  processElements(showElements, 'block');

  }
  </script>
  <!--onsubmit="this.action=document.getElementById('op').
options[document.getElementById('op').selectedIndex].value;"-->
  <form action=/in method="post">
    Welcome to AttuneInfocom!!! <br /> <br />
    Operation: <select id="op" name="op"
onchange="updateOptions(this.options[this.selectedIndex].value);">
                <option value="selectoption">--Select Option--</
option>
                    <option value="createF">Create Folder</option>
                    <option value="delete">Delete</option>
                </select><br /><br />


    <div class="hidden upFile createF delete downFile
list getLink" id="dropboxPath">Path:<input type="text"
name="dropboxPath" /></div>


    <input type="submit" value="Submit"/>
  </form>


</body>
</html>
```

7. Click on the `index.css` file. The following is the code snippet in the `index.css` file:

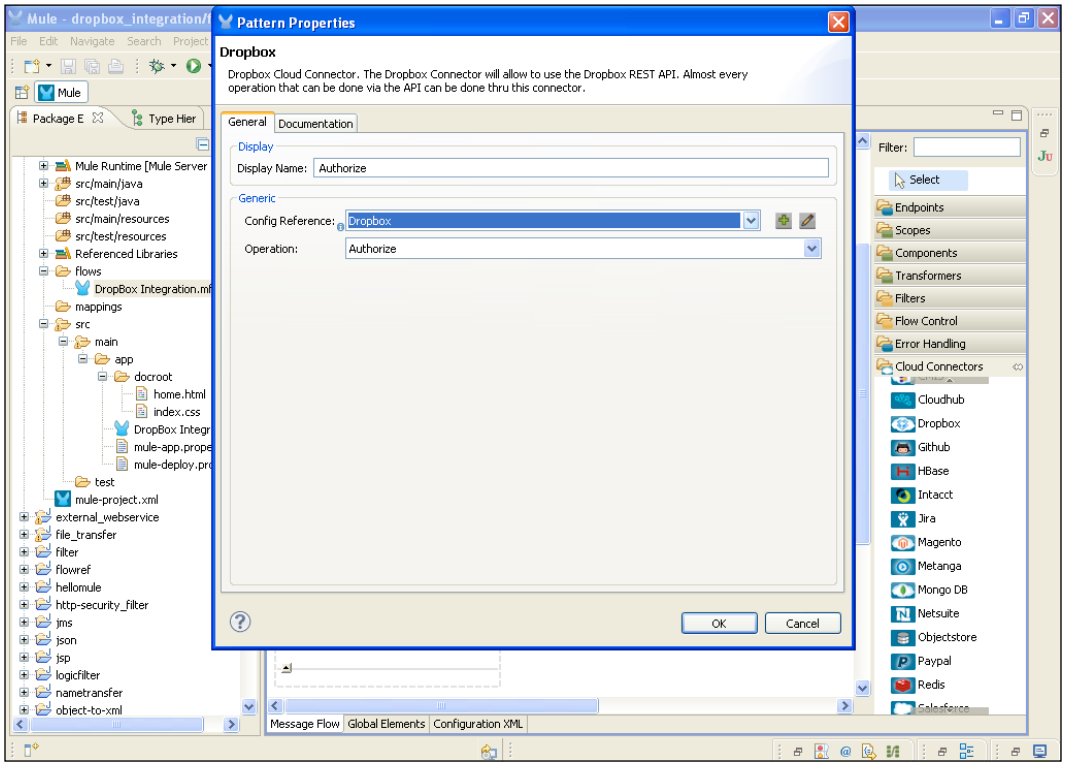
```
form {
background: -webkit-gradient(linear, bottom, left 175px,
from(#CCCCCC), to(#EEEEEE));
background: -moz-linear-gradient(bottom, #CCCCCC, #EEEEEE 175px);
margin:auto;
position:relative;
width:350px;
height:350px;
font-family: Tahoma, Geneva, sans-serif;
font-size: 14px;
font-style: italic;
line-height: 24px;
font-weight: bold;
color: #09C;
text-decoration: none;
-webkit-border-radius: 10px;
-moz-border-radius: 10px;
border-radius: 10px;
padding:10px;
border: 1px solid #999;
border: inset 1px solid #333;
-webkit-box-shadow: 0px 0px 8px rgba(0, 0, 0, 0.3);
-moz-box-shadow: 0px 0px 8px rgba(0, 0, 0, 0.3);
box-shadow: 0px 0px 8px rgba(0, 0, 0, 0.3);
}

textarea#feedback {
width:375px;
height:150px;
}
textarea.message {
display:block;
}
```

```
input.button {
width:100px;
position:absolute;
right:20px;
bottom:20px;
background:#09C;
color:#fff;
font-family: Tahoma, Geneva, sans-serif;
height:30px;
-webkit-border-radius: 15px;
-moz-border-radius: 15px;
border-radius: 15px;
border: 1p solid #999;
}
input.button:hover {
background:#fff;
color:#09C;
}
textarea:focus, input:focus {
border: 1px solid #09C;
}

img,a {
display:none;
}

#obj {
display:none;
}

.hidden {
display:none;
}
```

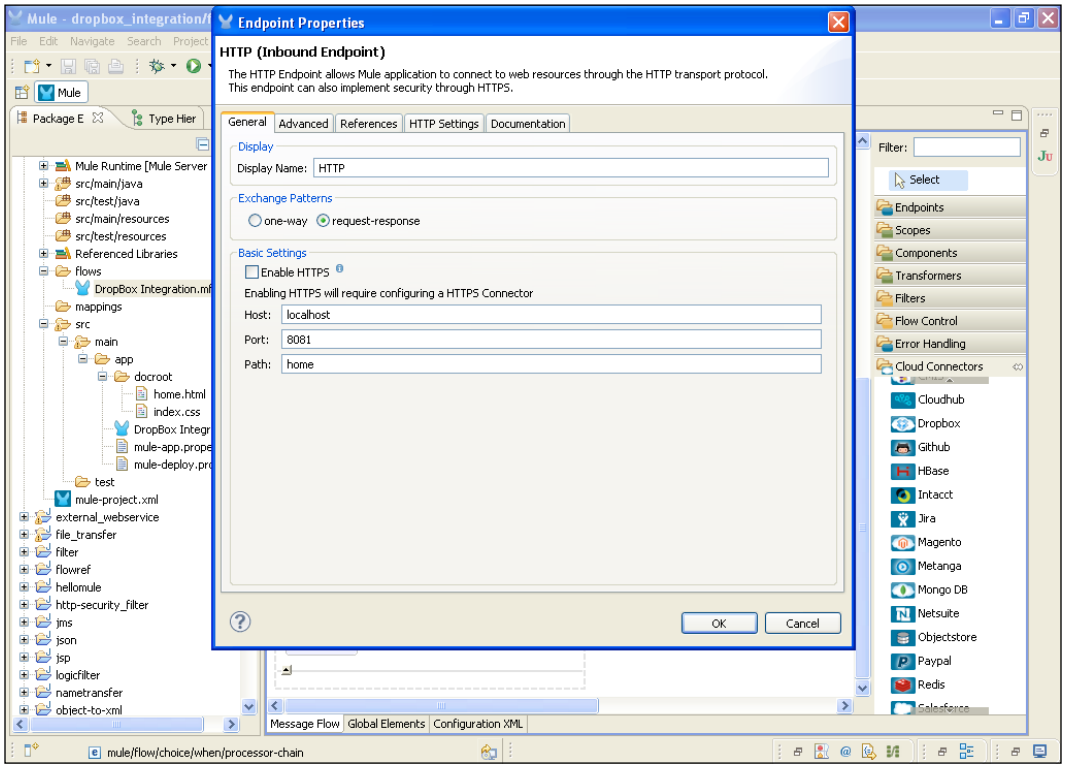
8. Go to the `dropboxIntegration.mflow` file and drag the **HTTP** Endpoint onto the canvas. First, you have to authorize that application. To configure the Endpoint, double-click on it. Enter the port number and the pathname.



9. Drag the **Dropbox** Cloud Connector onto the canvas. To configure the Connector, double-click on it, select the configuration reference **Dropbox**, and select the operation **Authorize**.



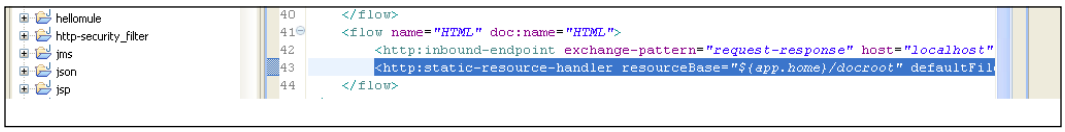
10. To create the second flow, drag the **HTTP** Endpoint onto the canvas. To configure the Endpoint, double-click on it. Enter the port number and the pathname.



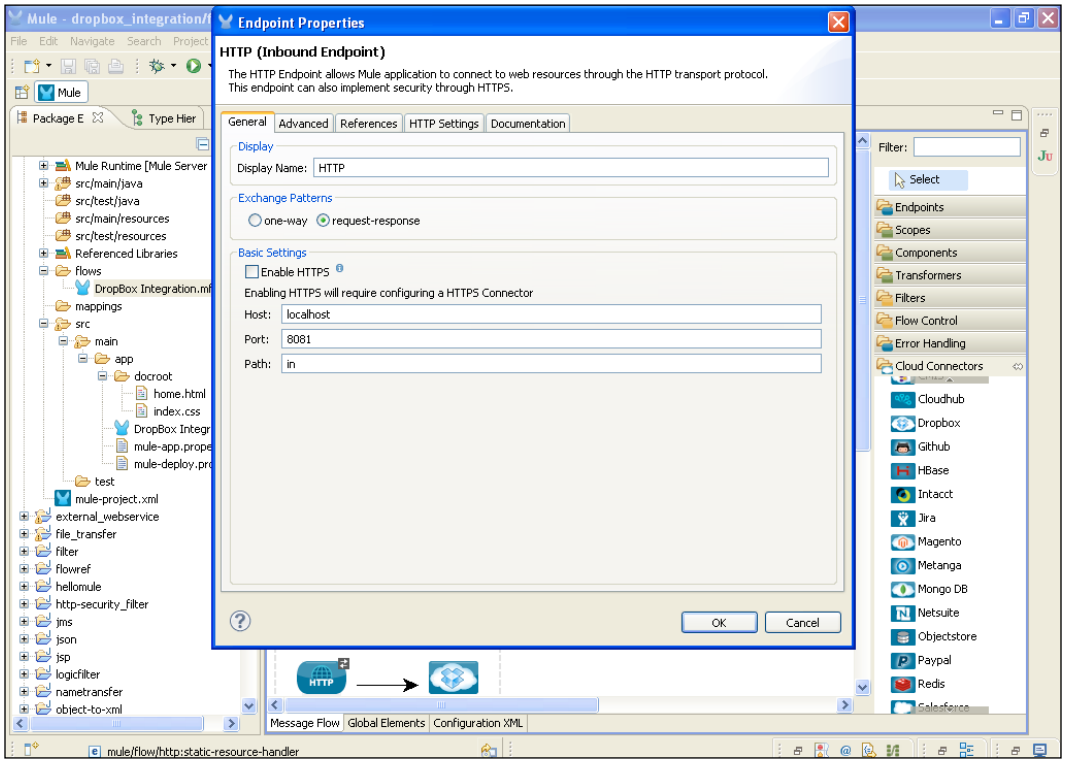
11. After the **HTTP** Endpoint configuration, you have to add the following line of code:

```
<http:static-resource-handler resourceBase="${app.home}/docroot"
defaultFile="home.html"></http:static-resource-handler>
```

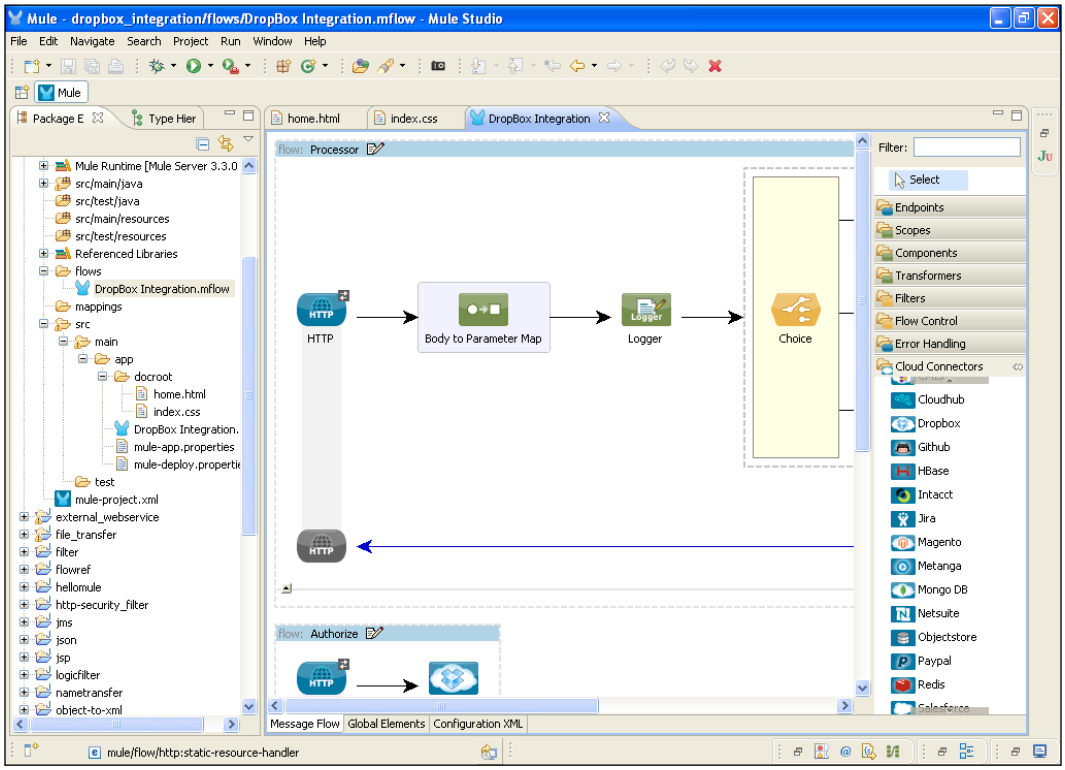
Through this handler, you call the `home.html` page in the `docroot` folder.



12. To create the third flow, drag the **HTTP** Endpoint onto the canvas. To configure it, double-click on the Endpoint. Enter the port number and the pathname.

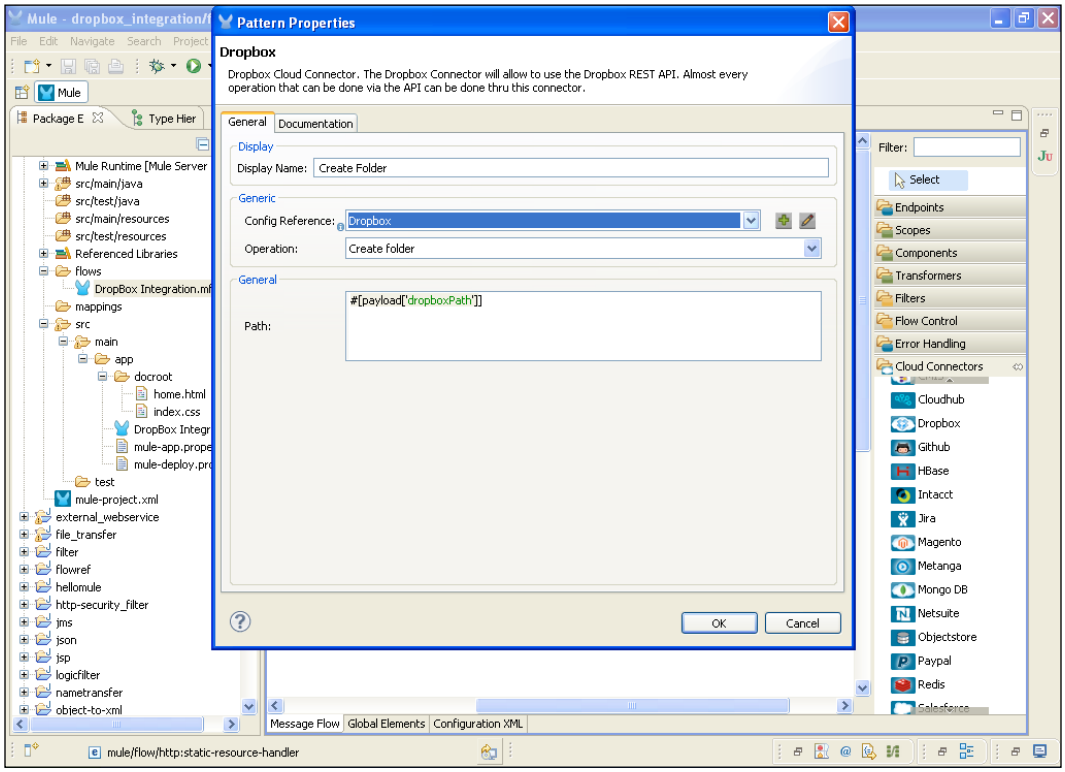


13. Drag the **Body to Parameter Map** transformer onto the canvas. There is no need to configure this Endpoint.



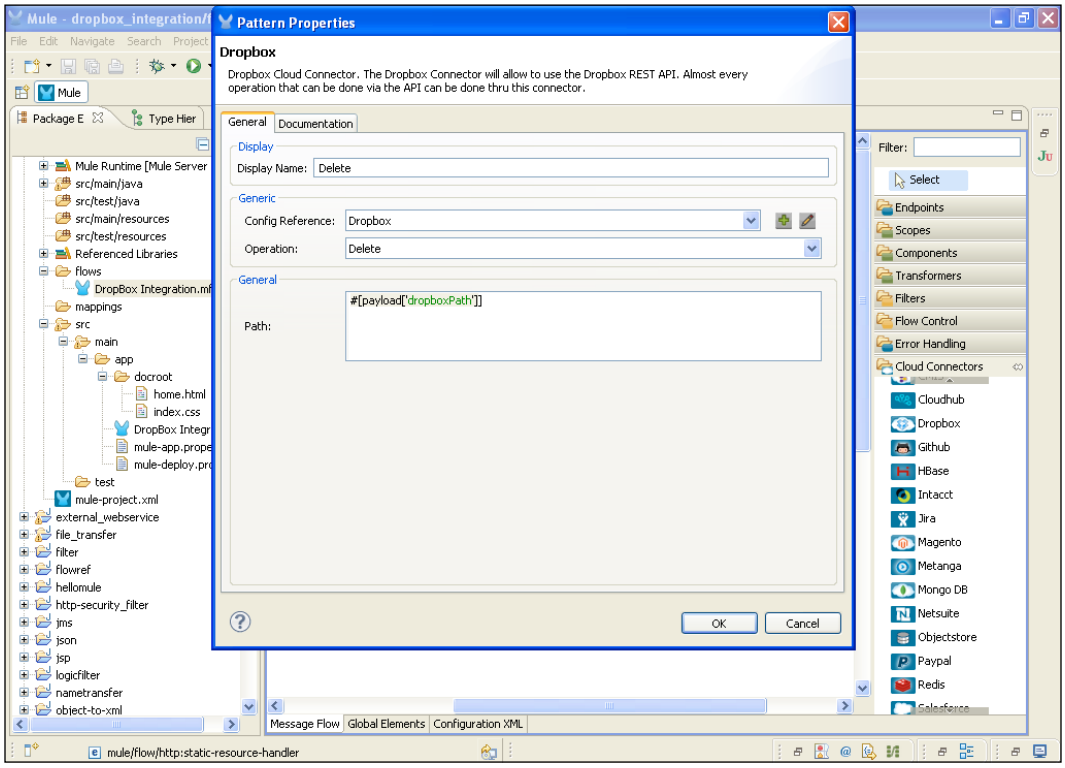
14. Drag the **Logger** component onto the canvas. It is used for displaying the log on the console.
15. Drag the **Choice** Router onto the canvas; you can configure it later.

16. Drag the **Dropbox** Connector onto the canvas. To configure the Connector, double-click on it. Select the configuration reference name, select the operation **Create folder**, and enter the path `#[payload['dropboxPath']]`.



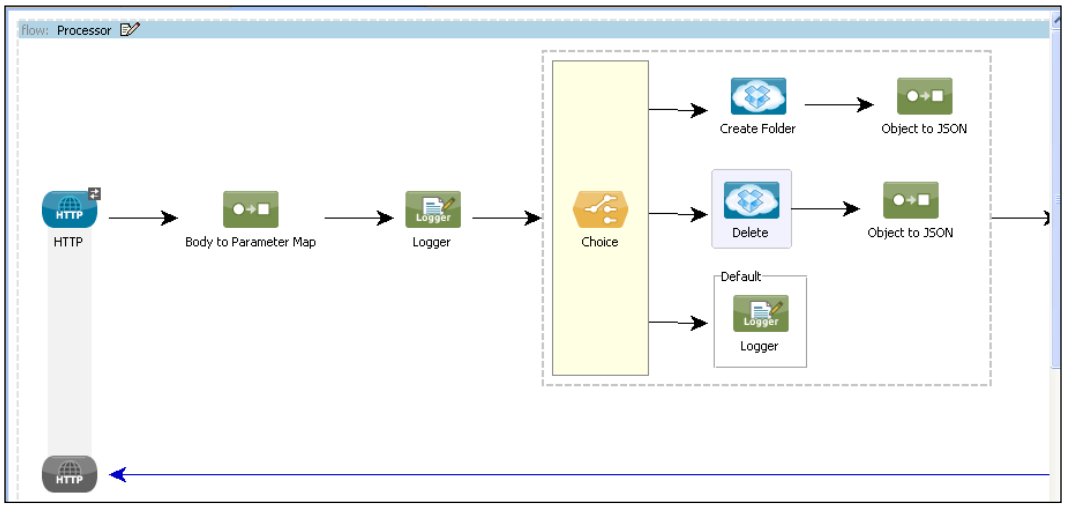
17. Drag the **Object to JSON** transformer onto the canvas; there is no need to configure it. This transformer is used to convert an object to the JSON format.

18. Drag the **Dropbox** Connector onto the canvas. To configure the Connector, double-click on it. Select the configuration reference name, select the operation **Delete**, and enter the path `#[payload['dropboxPath']]`.

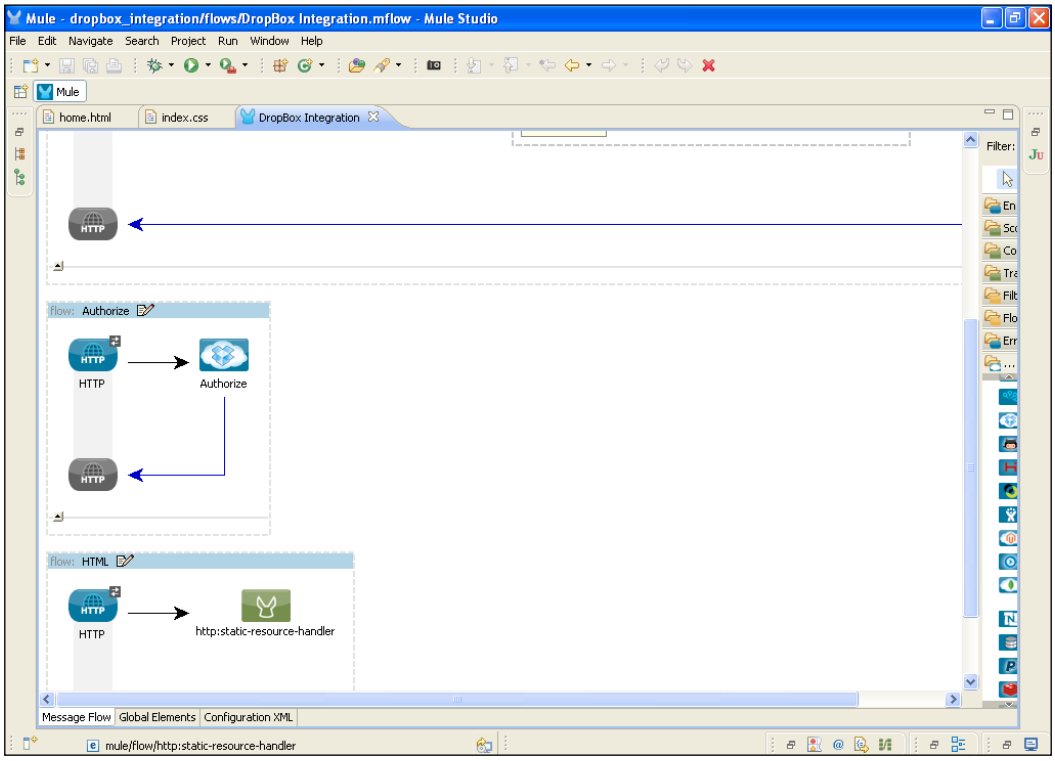


19. Drag another **Object to JSON** transformer onto the canvas; there is no need to configure it. This transformer is used to convert an object to the JSON format.

Your third flow will look like this:



20. This is your authorized flow. You have to first run this flow:

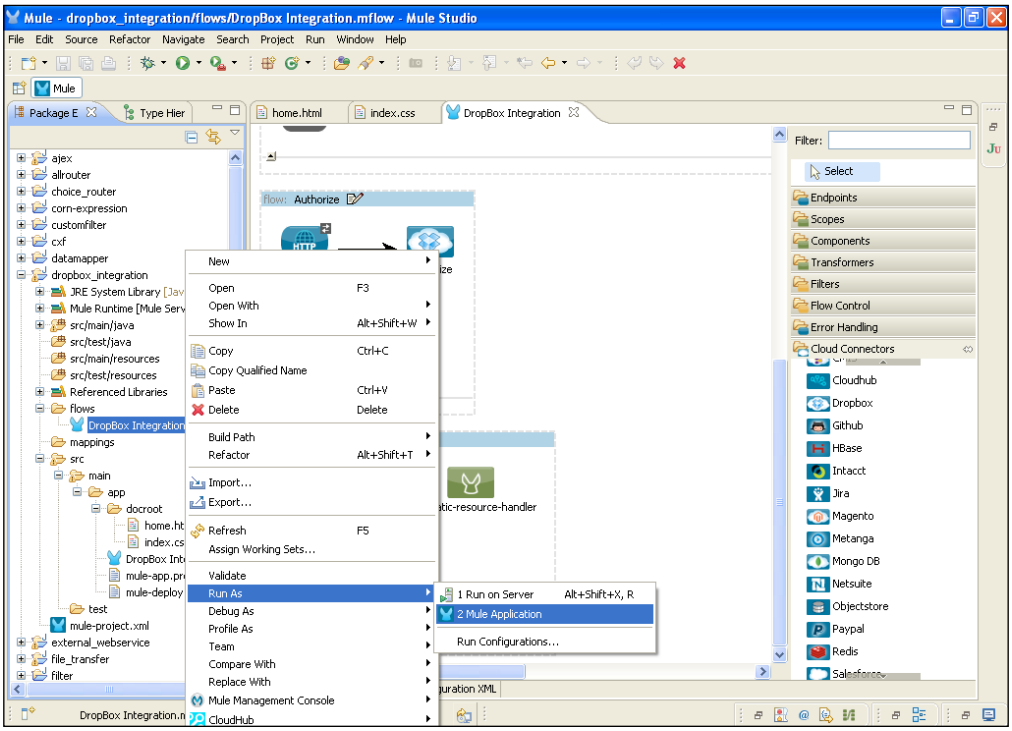


21. The following flow is your second flow. Once you have authorized an app, you need to run this flow. This flow will call the `home.html` page.

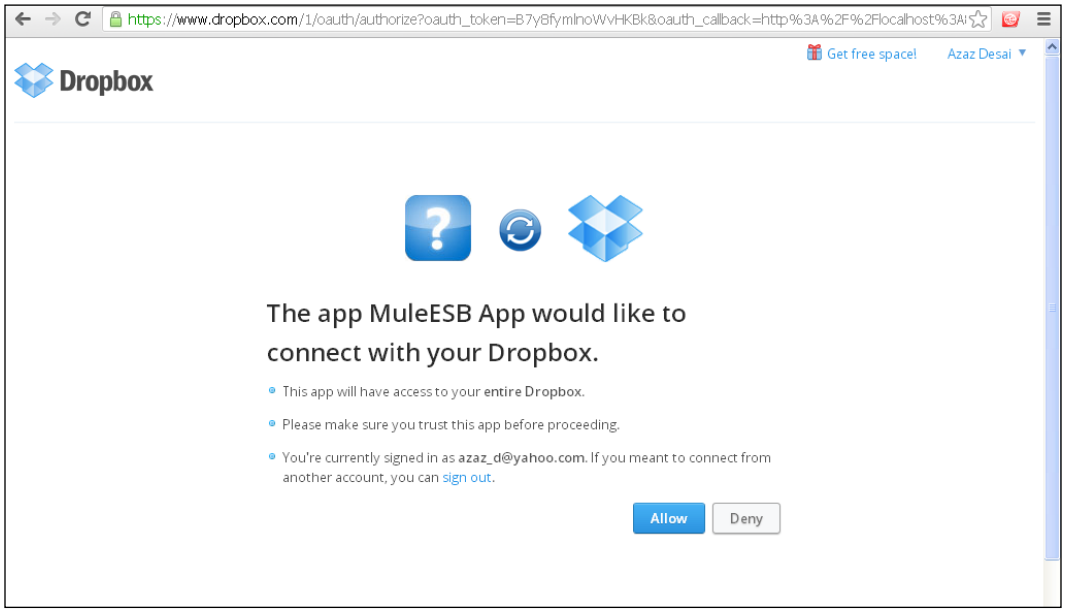
How it works...

In this section, you will see how to deploy the application.

1. To deploy the application code in the Mule server, go to **Run As** | **Mule Application**; the Mule server will deploy your application.

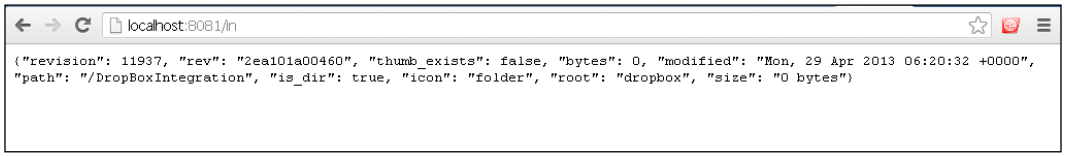


2. Open a browser and paste the following URL in it: `http://localhost:8081/auth`. Once you enter the URL, the page will redirect you to the Dropbox site, and here you have to click on **Allow** to provide access to the app.

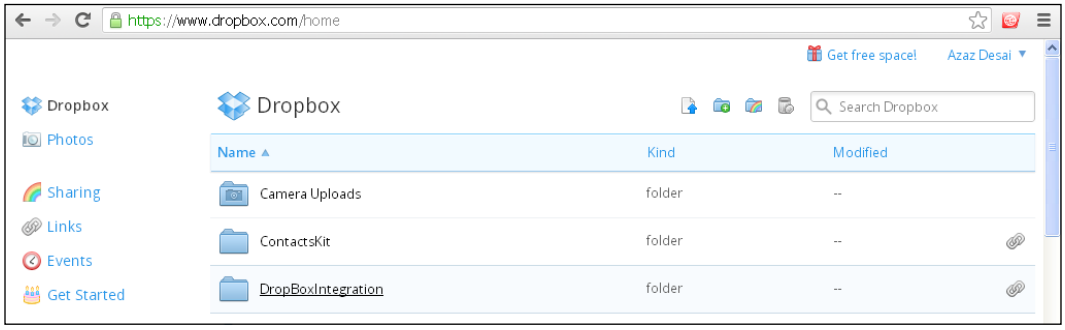


3. Now you can run the second flow, `http://localhost:8081/home/`. Here you need to select the operation. Suppose you have selected the **Create Folder** operation. You then have to enter the folder name, `DropBoxIntegration`, in the **Path:** textbox.

4. Once you click on the **Submit** button, you will see an output, similar to the following screenshot, in your browser:



5. Check your Dropbox account. You will see that a folder has been created on your account named `DropBoxIntegration`.



Index

D

E

F

G

H

I

J

L

M

O

P

R

S

T

V

W

X

Thank you for buying

Mule ESB Cookbook

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

About Packt Open Source

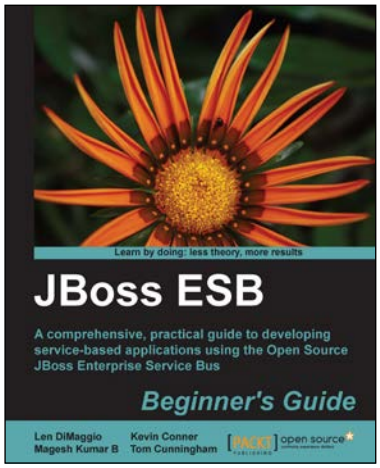
In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT] open source
community experience distilled
PUBLISHING



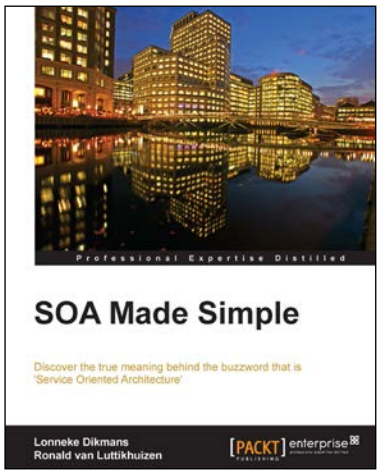
JBoss ESB Beginner's Guide

ISBN: 978-1-84951-658-7 Paperback: 320 pages

A comprehensive, practical guide to developing service-based applications using the Open Source JBoss Enterprise Service Bus

1. Develop your own service-based applications, from simple deployments through to complex legacy integrations
2. Learn how services can communicate with each other and the benefits to be gained from loose coupling
3. Contains clear, practical instructions for service development, highlighted through the use of numerous working examples

---



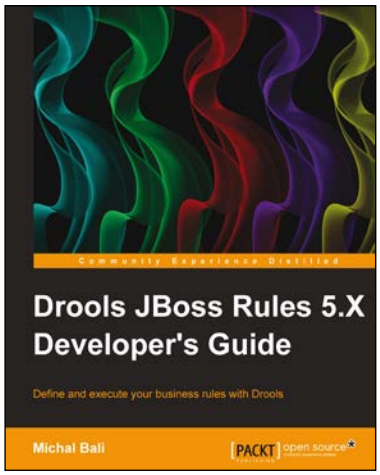
SOA Made Simple

ISBN: 978-1-84968-416-3 Paperback: 292 pages

Discover the true meaning behind the buzzword that is 'Service Oriented Architecture'

1. Get to grips with clear definitions of 'Service' and 'Architecture' to understand the full SOA picture
2. Read about SOA in simple terms from Oracle ACE Directors for SOA and Middleware in this book and e-book
3. A concise, no-nonsense guide to demystifying Service Oriented Architecture

[PACKT] open source
community experience distilled
PUBLISHING



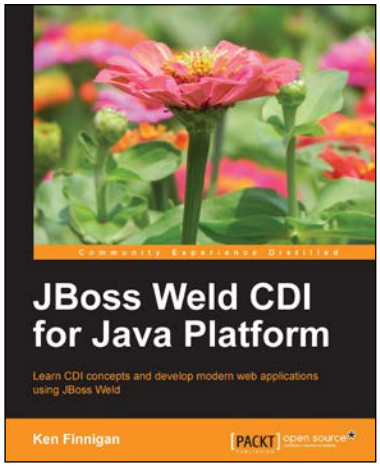
Drools JBoss Rules 5.X Developer's Guide

ISBN: 978-1-78216-126-4 Paperback: 338 pages

Define and execute your business rules with Drools

1. Learn the power of Drools as a platform for writing your business rules
2. Integrate Drools into your Java business application using the Spring framework
3. Through real-world examples and solutions, you will be taken from novice to expert

---



JBoss Weld CDI for Java Platform

ISBN: 978-1-78216-018-2 Paperback: 122 pages

Learn CDI concepts and develop modern web applications using JBoss Weld

1. Learn about dependency injection with CDI
2. Install JBoss Weld in your favorite container
3. Develop your own extension to CDI
4. Decouple code with CDI events
5. Communicate between CDI beans and AngularJS