First Edition (July 2008)

This edition applies to Version 6.1 of WebSphere eXtreme Scale.

This document created or updated on July 29, 2008.

© Copyright International Business Machines Corporation 2008. All rights reserved.
Note to U.S. Government Users Restricted Rights -- Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
## Contents

### Notices
- Notices ................................................................. vii
- Trademarks ............................................................ vii

### Preface
- The team that wrote this paper ..................................... ix
- Become a published author .......................................... xi
- Comments welcome .................................................... xii

### Chapter 1. Introduction to WebSphere eXtreme Scale
- 1.1 Introduction to WebSphere eXtreme Scale ..................... 2
- 1.2 WebSphere eXtreme Scale overview .............................. 4
  - 1.2.1 Architectural concepts ...................................... 4
  - 1.2.2 Container services ........................................... 4
  - 1.2.3 Partitions ..................................................... 5
  - 1.2.4 Catalog services ............................................. 8
  - 1.2.5 Maps ........................................................ 12
  - 1.2.6 MapSet ...................................................... 16

### Chapter 2. eXtreme Scale programming model
- 2.1 eXtreme Scale programming model .............................. 18
- 2.2 eXtreme Scale application programming interfaces .......... 21
  - 2.2.1 Package com.ibm.websphere.objectgrid interface summary . . 24
  - 2.2.2 Package com.ibm.websphere.objectgrid class summary .......... 25
  - 2.2.3 eXtreme Scale core programming API ........................ 25
  - 2.2.4 ObjectMap API .............................................. 32
  - 2.2.5 EntityManager API .......................................... 38
  - 2.2.6 Query APIs .................................................. 45
  - 2.2.7 DataGrid API ................................................. 57
  - 2.2.8 Package com.ibm.websphere.objectgrid.datagrid interface summary .............................................. 58
  - 2.2.9 eXtreme Scale system programming API ...................... 65
# Table of Contents

**Chapter 3. The WebSphere eXtreme Scale mediator pattern**

- 3.1 The pattern .................................................. 68
- 3.2 Example ....................................................... 68
- 3.3 Context ........................................................ 69
- 3.4 Problem ....................................................... 70
- 3.5 The solution ................................................... 70
- 3.6 Structure ....................................................... 72
- 3.7 The dynamics .................................................. 73

**Chapter 4. The base application** .................................... 75

- 4.1 The base application implementation .......................... 76
  - 4.1.1 ITSOFinance application ................................ 76
  - 4.1.2 ITSOFinanceData application ............................ 83
- 4.2 Setting up the base application environment .................. 89
  - 4.2.1 Configure the application servers ....................... 90
  - 4.2.2 Configure JMS messaging service ....................... 94
  - 4.2.3 Install ITSOFinance and ITSOFinanceData applications 113
- 4.3 Testing the base application .................................. 116

**Chapter 5. The eXtreme Scale mediator application** ............ 121

- 5.1 The mediator application implementation ..................... 122
  - 5.1.1 ITSONMediator application ............................. 122
  - 5.1.2 MediatorMDBBean ....................................... 123
  - 5.1.3 MediatorLoader ......................................... 139
- 5.2 Setting up the mediator environment ........................ 142
  - 5.2.1 ITSOFinanceGrid configuration ........................ 143
  - 5.2.2 Install the mediator application ....................... 148
  - 5.2.3 Create the mediator dynamic cluster ................... 150
  - 5.2.4 Update messaging services ............................. 155
  - 5.2.5 Install the ITSONMediator application ................. 173
- 5.3 Testing the mediation application ............................ 174

**Appendix A. Additional material** ................................ 177

Locating the Web material ........................................... 177
Using the Web material ............................................... 178
  - System requirements for downloading the Web material ....... 178
  - How to use the Web material .................................. 178
Abbreviations and acronyms ........................................... 179

Related publications .................................................. 181
Online resources ......................................................... 181
How to get Redbooks .................................................. 181
Help from IBM .......................................................... 181

Index ................................................................. 183
Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.
Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at http://www.ibm.com/legal/copytrade.shtml

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:


The following terms are trademarks of other companies:

EJB, Java, JavaBeans, JDK, JSP, JVM, and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Windows, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.
Preface

The IBM® WebSphere® Extended Deployment provides a dynamic and high performance computing infrastructure across a range of application servers and execution environments. In addition, it enhances the utilization and operational management of the infrastructure. In version 6.1, WebSphere Extended Deployment delivers additional capabilities in three functionally independent and separately available components. WebSphere Virtual Enterprise provides application server virtualization, resource management, and operational facilities, such as performance visualization, health monitoring, and application versioning. WebSphere eXtreme Scale provides high-end caching and transaction partitioning capabilities. WebSphere eXtreme Scale delivers performance improvements across a wide range of application scenarios. Compute Grid provides dynamic workload management which offers support for mixed application types and ensures that service levels are met for priority requests. It also supports new batch-like jobs referred to as native execution.

In this IBM Redpaper publication, we describe the concepts, programming model, and implementation of the mediator application pattern using WebSphere eXtreme Scale. The WebSphere eXtreme Scale mediator pattern promotes loose coupling between architectural components while maintaining unified interfaces. Usually large and distributed systems are fairly complex, making it difficult and expensive to change. The WebSphere eXtreme Scale mediator pattern enables the introduction of the eXtreme Scale cache into distributed messaging and service oriented applications, without altering the interfaces and behavior, to provide significant improvement in performance, availability, and scalability.

As an architect, solution designer, or developer, this book is aimed at you. The goal is to enable you to create solutions that take advantage of the features of WebSphere eXtreme Scale.

The team that wrote this paper

This paper was produced by a team of specialists from around the world working at the International Technical Support Organization, Raleigh Center.

Edward Oguejiofor is a project leader at the International Technical Support Organization, Raleigh Center. He has over 20 years experience in distributed and enterprise systems architecture and design. His areas of expertise include Web
Services and Service Oriented Architecture. He also provides technical and thought leadership in emerging technologies and collaborative services. He holds a degree in Computer Science from University of London’s Imperial College of Science and Technology.

**Gopinath Kesavan** is an Architect with Cognizant Technology Solutions, Global Technology Office in Chennai, India. He has worked in the IT industry for 10 years and holds a Master’s degree in Software Engineering. His area of expertise includes Core Java™, WebSphere Application Server, Performance and High Availability.

**Vincent Pedroza** is a Software Engineer with IBM Software Group. He has over 6 years experience working with WebSphere Application Server and WebSphere Extended Deployment. He has worked at IBM for 20 years and holds a Bachelor of Science degree from Rochester Institute of Technology. His areas of expertise include test automation and system verification for WebSphere eXtreme Scale and application integration middleware software.

*Figure 0-1  (left to right) Edward Oguejiofor, Gopinath Kesavan, and Vincent Pedroza.*
Thanks to the following people for their contributions to this project:

IBM Software Group
- Jared Anderson
- Douglas Berg
- Joe Bockhold
- Cheng-chieh (Jerry) Cheng
- Chris Johnson
- Billy Newport
- Alasdair Nottingham
- Michael Schmitt

International Technical Support Organization, Raleigh Center
- Tamikia Barrow
- Margaret Ticknor

International Technical Support Organization, Poughkeepsie Center
- Michael B. Schwartz

**Become a published author**

Join us for a two- to six-week residency program! Help write a book dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You will have the opportunity to team with IBM technical professionals, Business Partners, and Clients.

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you will develop a network of contacts in IBM development labs, and increase your productivity and marketability.

Find out more about the residency program, browse the residency index, and apply online at:

[ibm.com/redbooks/residencies.html](http://ibm.com/redbooks/residencies.html)
Comments welcome

Your comments are important to us!

We want our papers to be as helpful as possible. Send us your comments about this paper or other IBM Redbooks® in one of the following ways:

▶ Use the online Contact us review Redbooks form found at:
  ibm.com/redbooks

▶ Send your comments in an e-mail to:
  redbooks@us.ibm.com

▶ Mail your comments to:
  IBM Corporation, International Technical Support Organization
  Dept. HYTD Mail Station P099
  2455 South Road
  Poughkeepsie, NY 12601-5400
Introduction to WebSphere eXtreme Scale

In this chapter we introduce the WebSphere eXtreme Scale component of WebSphere Extended Deployment version 6.1, and then turn our attention to the in-memory grid enabled cache which is our focus in this book. We discuss the concepts, features, and programming model.

This chapter contains the following sections:

- Introduction to WebSphere eXtreme Scale
- WebSphere eXtreme Scale overview
1.1 Introduction to WebSphere eXtreme Scale

The WebSphere eXtreme Scale component of the WebSphere Extended Deployment provides an environment for efficient and high-performance processing of massive volumes of data. It supports Extreme Transaction Processing (XTP) which is characterized by the processing of large scale and highly-distributed business-critical transactions.

Gartner defines XTP as “an application style aimed at supporting the design, development, deployment, management, and maintenance of distributed TP applications characterized by exceptionally demanding performance, scalability, availability, security, manageability, and dependability requirements.”

The two key parts of WebSphere eXtreme Scale are the Partitioning facility, which provides a partitioning model that enables “asymmetric clustering,” and the ObjectGrid component which is an extensible data caching framework used for developing advanced high-performance Java applications.

- Partitioning facility
  
The partitioning facility is a programming framework and a system management infrastructure. Each uniquely addressable endpoint within a cluster can become a partition. The partitioning facility incorporates the ability to make each endpoint (hence each partition) highly available and manageable using the high availability manager (HA manager) in WebSphere Extended Deployment.

The partitioning facility can be used to address different usage scenarios where transactions are routed to application-specific partitions. The following types of partitioning are supported:

- HTTP partitioning
  
  This capability works in conjunction with the on-demand router (ODR) in WebSphere Extended Deployment to route HTTP requests to appropriate target application servers.

- EJB™ workload partitioning
  
  This capability augments the EJB workload management in the typical Java Platform Enterprise Edition (Java EE) technologies. To use the partitioning facility, you have to develop an additional stateless session bean called a partitioned stateless session bean. (PSSB) The WebSphere application server makes use of the PSSB to query the application at start-up to determine the number of partitions the application requires. In addition to the normal Session Bean interface, the PSSB also implements the PartitionHandlerLocal interface which is used for the processing of partition-related events.
– Database partitioning

This capability enables you to create separate instances of databases which hold subsets of the data. The subset of the data is used exclusively by partitions. Through careful design and implementation, you can easily scale the database across multiple servers, reducing database contention, aggressively cache data in memory, and realize improved performance and scalability.

**Note:** In this paper, we focus only on solution design and development using the ObjectGrid component of the WebSphere eXtreme Scale component. You can find more information about the WebSphere Partitioning Facility from IBM Redbooks publication “Using WebSphere Extended Deployment V6.0 To Build an On Demand Production Environment,” available at: http://www.redbooks.ibm.com/abstracts/sg247153.html

**ObjectGrid component**

The ObjectGrid component of WebSphere eXtreme Scale is designed for extremely fast processing of large volumes of data. This feature partitions data into non-overlapping in-memory chunks that can be stored on several machines. Queries can be run in parallel on all the machines in the cluster.

Because the data in is maintained in memory, it is possible to lose the data when a node in the cluster fails. Fault tolerance is provided by maintaining replica copies of each partition on different nodes. When a machine fails, a replica on a surviving machine is promoted to be the new primary partition and a new replica is reassigned on another surviving machine to ensure that fault tolerance is maintained.

WebSphere eXtreme Scale scales horizontally across thousands of machines to create large grids with complex topologies. It manages the automatic placement of data in the grid and the co-location of business logic with the partitioned data.
1.2 WebSphere eXtreme Scale overview

WebSphere eXtreme Scale is an in-memory transactional caching framework for applications that are written in Java. It provides a flexible infrastructure for realizing high performance, scalable, and data-intensive applications. You can use the WebSphere eXtreme Scale for solutions ranging from simple in-memory database to powerful distributed coherent cache that scales to thousands of servers.

1.2.1 Architectural concepts

In order to provide scalable performance and high availability, WebSphere eXtreme Scale introduces a number of architectural concepts for coherent caching. In this section we provide overview of the key concepts.

1.2.2 Container services

eXtreme Scale runs in a Java Virtual Machine. (JVM™) The application data is hosted by container services in JVMs. A container can be started in a JVM by itself or you can have multiple container services running in a JVM.

![Java Virtual Machine and Container Service](image)

*Figure 1-1  Local eXtreme Scale container service*

eXtreme Scale can be deployed to run locally in a single JVM, or across several JVMs in a distributed setting. In a distributed setting, the application data is broken into parts, called partitions, and hosted across multiple containers. Each container hosts a subset of the complete data.
Both local and distributed environments can be configured using an eXtreme Scale XML configuration file or an equivalent programmatic configuration.

### 1.2.3 Partitions

Partitions are organized across the network of containers running on JVMs. Each partition hosts a subset of the application data. A hash mechanism is used as the basis for dividing the data into partitions.

Partitioning is used as the mechanism for scaling out applications. Partitions are automatically repaired and organized on available containers. By splitting the data across several JVMs, partitioning makes it possible to store more data than would fit into the JVM heap in a grid.
**Shards**

An instance of a partition is called a shard. Each shard hosts the full set of data for the partition. Multiple shards from different partitions can exist in a single container but never more than one shard from the same partition. A shard can either be a primary or a replica shard. A partition can have one or more replica shards.

- **Primary shard**
  
  All insert, update, and delete operations to the partition take place on the primary shard only. The primary shard sends data to the replicas, and manages commits and rollbacks of transactions.

- **Replica shard**
  
  Replica shards are “mirrored” instances of the primary partition. Replicas are never hosted in the same container as the primary and are not normally hosted on the same machine as the primary. A replica shard can be either a synchronous shard or an asynchronous shard.

  Synchronous replicas maintain the same state as the primary. When a primary replicates data to a synchronous replica, the transaction is not committed until it commits on the synchronous replica.

  Asynchronous replicas might or might not be at the same state as the primary. When a primary replicates data to an asynchronous replica, the primary does not wait for the asynchronous replica to commit. An asynchronous replica still maintains the order of the transactions that is sent from the primary.

**Note:** Replicating data from primary to replica shards increases the availability and persistence guarantee of data. However, replication adds cost to the transaction. WebSphere eXtreme Scale allows this cost to be finely controlled through synchronous and asynchronous replication as well as hybrid replication models using both synchronous and asynchronous replication modes.
Although a grid can have many thousands of partitions, you must choose a reasonable number of partitions based on the expected number of JVMs that are likely to be deployed. A grid scales based on the number of partitions times the number of shards per partition. For example, if you have 16 partitions and each partition has one primary and one replica (or two shards), then you can potentially scale to 32 JVMs. Each shard carries some overhead and WebSphere eXtreme Scale is designed to scale out to handle this overhead in line with how many server JVMs are available.
1.2.4 Catalog services

The catalog server consists of a set of services which include the dynamic eXtreme Scale server creation and grouping management, the heartbeat health monitoring of the servers, and the placement of shards in distributed partitions. The catalog server plays a critical role in WebSphere eXtreme Scale installations, so to ensure its availability the catalog server is designed to be clustered. One of the servers in the catalog cluster is always elected as the master catalog server. If the master catalog server fails, another catalog server is elected the new master catalog server. Any bootstrapped clients and already-started eXtreme Scale servers are automatically rerouted to the new master catalog server.

**Note:** You should not specify the use of thousands of partitions if your application runs on a grid consisting of a small number of container JVMs, for example, four container JVMs. Your configuration should have a reasonable number of shards for each container JVM. It is unreasonable to configure 2000 partitions with primary and replica shards to run on four container JVMs. This configuration would result in 4000 shards being placed in four container JVMs, or 1000 shards per container JVM.

A better configuration would be under 10 shards for each expected container JVM. This configuration still gives the possibility of scaling ten times the initial configuration while maintaining a reasonable number of shards per container JVM.
The catalog server provides the following services:

- **Location Service**
  
  Location Service enables clients to find containers that host the data they seek, and containers make use of the location service to find and register hosted application with the Placement Service.

- **Placement Service**
  
  The placement services is responsible for allocating individual shards to their host container. There is always exactly one instance of the service running in the catalog cluster. The state information is replicated across servers hosting the catalog service, so should the instance of the placement service fail, then another process is elected and takes over.
Core group management

Servers in the grid are organized in groups of 20 servers which are then federated to create a hierarchical topology. Servers and containers are grouped based on the sequence of registration with the catalog server. When the eXtreme Scale container starts up it registers with the core group manager to get its core group name. The core group manager is responsible for the dynamic management of the servers and notifying servers that are in the same core group about peer members.

One of the servers in the group of 20 is elected the leader. It acts as the peer manager and communicates the groups health and events with the master catalog server through Internet Inter-ORB Protocol (IIOP).

![Diagram of eXtreme Scale group leader servers](image-url)
Each group of eXtreme Scale servers monitors its member health through the integrated distribution and consistency services (DCS) and high availability manager. The group leader server reports the group members health to the Core group manager in the Catalog server. The core group manager is responsible for collecting the health information from all servers and manages the corresponding containers.

► Administration

The catalog server also provides the central point for the eXtreme Scale system administration.

![Catalog server services](image.png)

*Figure 1-6  Catalog server services*
1.2.5 Maps

Maps provide a structured way for storing data in eXtreme Scale. It associates key objects to data objects, where each key maps to a single data object. eXtreme Scale supports two types of maps, ObjectMaps and entity maps.

- **ObjectMaps**
  
  ObjectMaps are used for caching data objects as key/value pairs. ObjectMaps are like Java maps. The ObjectMap API provides a map-based API so data can be put into and retrieved from an ObjectMap using the usual Map-like methods. All ObjectMap operations are performed within the scope of a transaction.

  ![Figure 1-7 ObjectMap key/value association](image)

  eXtreme Scale allows for the definition of indexes on attributes of the key or value for faster and more efficient execution of queries. When relationships can be identified between objects in the map, you can define a map schema that captures the relationship. Schemas are also used for querying Map objects. A grid can have multiple Maps and schemas defined.

- **Entity maps**

  eXtreme Scale also supports use of entities to store data. Entities allow applications to easily have complex object graphs that span multiple Maps. Each entity can have a set of key attributes and set of non-key attributes. eXtreme Scale automatically discovers the schema for the entity using either an entity descriptor XML file or annotated Java classes. Entities can have one-to-one, one-to-many, many-to-one and many-to-many relationships. Figure 1-8 on page 13 shows a sample schema for entities which consists of customer, accounts, and transactions. Example 1-1 on page 14 shows Java code that declare entities, and Example 1-2 on page 15 shows sample code that make use of the entities to persist objects in the grid.
### Figure 1-8  Customer data entities

<table>
<thead>
<tr>
<th>Key object</th>
<th>Value object</th>
</tr>
</thead>
<tbody>
<tr>
<td>customerID : String</td>
<td>Customer</td>
</tr>
</tbody>
</table>
| firstName : String  
lastName : String  
Address : String  
taxID : String  
eMail : String | |

1..n

<table>
<thead>
<tr>
<th>Key object</th>
<th>Value object</th>
</tr>
</thead>
<tbody>
<tr>
<td>accountID : String</td>
<td>Account</td>
</tr>
</tbody>
</table>
| accountNumber : String  
typeOfAccount : String  
balance : double | |

1

<table>
<thead>
<tr>
<th>Key object</th>
<th>Value object</th>
</tr>
</thead>
<tbody>
<tr>
<td>account : String</td>
<td>Transaction</td>
</tr>
</tbody>
</table>
| Date : java.Util.Date  
amount : double  
txDescription : String | |

0..n
Example 1-1  Sample annotated Java entities

```java
@Entity
public class Customer {
    @Id String customerID;
    String firstName;
    String lastName;
    String address;
    String taxID;
    String eMail;
}

@Entity
public class CustomerQueryResult {
    @Id long rowId;
    String id;
    String firstName;
    String surname;
}

@Entity
public class Account {
    @Id String accountID;
    String accountNumber;
    String typeOfAccount;
    double balance;
    @ManyToOne(cascade=CascadeType.PERSIST) Customer customer;
}

@Entity
public class Transaction {
    @Id @ManyToOne(cascade=CascadeType.PERSIST) Account account;
    java.util.Date date;
    double amount;
    String txDescription;
}
```
Example 1-2  Sample Java code

```java
public class Application {
    static public void main(String [] args)
        throws Exception {

        ObjectGrid og =
                ObjectGridManagerFactory.getObjectGridManager().createObjectGrid();
        og.registerEntities(new Class[] {Account.class});

        Session s = og.getSession();
        EntityManager em = s.getEntityManager();

        em.getTransaction().begin();

        Customer cust = new Customer();
        cust.id = "01";
        cust.firstName = "John";
        cust.surname = "Doe";
        cust.address = "Main Street";
        cust.taxID = "123-456-7890";
        cust.email = "john.doe@email.com";

        Account acct = new Account();
        acct.customer = cust;
        acct.accountID = "1"
        acct.typeOfAccount= "Checking";
        acct.accountNumber= "CHK-123-4567";
        acct.quantity = 100;

        em.persist(acct);
        em.getTransaction().commit();

        em.getTransaction().begin();
        acct = (Account)em.find(Account.class, "1");
        System.out.println("Found account for customer: " + o.customer.firstName + " "+ o.customer.surname);
        em.getTransaction().commit();
    }
}
```
1.2.6 MapSet

A mapSet is used to group a number of maps together. The group of maps share a common partitioning key and are also partitioned and replicated in a similar manner. A mapSet is only used for distributed topologies and is not required for local grids. The data within the maps are replicated based on the policy defined in the ObjectGrid deployment file. Example 1-3 shows two maps called FinanceMap and RequestMap defined in the mapSet FinanceMapSet. The mapSet element also defines the number of partitions for the mapSet, the minimum number of synchronous replicas for each partition, the maximum number of synchronous replicas for each partition, the maximum number of asynchronous replicas for each partition, and the number of eXtreme Scale containers that are required before initial placement occurs for the shards.

Example 1-3 Sample ObjectGridDeployment.xml file

```xml
<?xml version="1.0" encoding="UTF-8"?>
<deploymentPolicy xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://ibm.com/ws/objectgrid/deploymentPolicy
    ../deploymentPolicy.xsd"
    xmlns="http://ibm.com/ws/objectgrid/deploymentPolicy">
    <objectgridDeployment objectgridName="ITSOFinanceGrid">
        <mapSet name="FinanceMapSet" numberOfPartitions="3"
            minSyncReplicas="1" maxSyncReplicas="1" maxAsyncReplicas="0"
            numInitialContainers="3">
            <map ref="FinanceMap" />
            <map ref="RequestMap" />
        </mapSet>
    </objectgridDeployment>
</deploymentPolicy>
```

Note: For more information about the sample ObjectGrid deployment file in Example 1-3 on page 16, see section 5.2.1 on page 143.
In this chapter we introduce the eXtreme Scale programming model and an overview of the eXtreme Scale application programming interfaces.

This chapter contains the following sections:

- eXtreme Scale programming model
- eXtreme Scale application programming interfaces
2.1 eXtreme Scale programming model

The client-server programming model is supported for eXtreme Scale. This includes locally-deployed grids running on a single JVM and distributed grids running on several JVMs. The programming model is based on concepts that enable you to programmatically access and manipulate features of a grid.

- **ObjectGrid**
  The ObjectGrid (or grid) is the container of the Java objects cached. To cache objects using a grid, your application must obtain a reference to an ObjectGrid instance. ObjectGrid instances are created either programmatically or from XML-based configuration. Each instance can define one or more BackingMaps where the key/value pair map objects is stored. See section 2.2.3, on page 25 for more information about eXtreme Scale APIs.

- **BackingMap**
  The BackingMap contains the committed in-memory cached Java objects. An application can create one or more BackingMaps in a grid. The BackingMap can be populated by the application itself or it can be loaded from a backend datasource by implementing a custom Cache Loader. All objects in a given BackingMap must be of the same type.
Each BackingMap has a set of attributes which control the behavior of the BackingMap. Here is the list of the supported attributes:

- **ReadOnly**
  This attribute indicates if the BackingMap is a read-only map or a read and write map. BackingMaps are by default read and write maps.

- **NullValuesSupported**
  This attribute indicates if null values can be stored into the map. BackingMaps do not support null values by default. If set to accept null values, an operation that returns null can mean that either the value is null or the map does not contain the key specified by the operation.

- **LockStrategy**
  This attribute determines if a lock manager is used or not. If a lock manager is used, then the attribute indicates whether optimistic or pessimistic locking approach is used for locking entries in the map. If this attribute is not set, eXtreme Scale uses optimistic LockStrategy for the BackingMap.

- **CopyMode**
  This attribute determines how value objects are copied when read from or put into the BackingMap during the commit cycle of a transaction. The modes allow applications to make trade-off between performance and data integrity. If this attribute is not set, COPY_ON_READ_AND_COMMIT copy mode is used.

- **CopyKey**
  This attribute determines if the BackingMap makes a copy of a key object when an entry is first created in the map. The default action is to not make a copy of key objects.

- **NumberOfBuckets**
  This attribute indicates the number of hash buckets to be used by the BackingMap. The BackingMap uses hash maps to determine the bucket for keys. If the BackingMap has more buckets then the number of keys that
have the same bucket are lower, this results in more concurrency and better performance.

- **NumberOfLockBuckets**

  This attribute indicates the number of lock buckets used by the lock manager for a BackingMap. When the LockStrategy is set to either OPTIMISTIC or PESSIMISTIC, a lock manager is created. The lock manager uses a hash map to keep track of entries locked by one or more transactions. When the LockStrategy attribute is set to NONE, the lock manager is not used and setting numberOfLockBuckets has no effect.

- **LockTimeout**

  This attribute is used when the BackingMap is using a lock manager and the LockStrategy attribute is set to either OPTIMISTIC or PESSIMISTIC. The value of this attribute is in seconds and determines how long the lock manager waits for a lock to be granted. If this attribute is not set, then 15 seconds is used as the LockTimeout value.

- **TtlEvictorType**

  Every BackingMap has built-in time-to-live evictor that uses a time-based algorithm to determine which map entries to evict. By default, the built-in time-to-live evictor is not active. You can activate it by calling the setTtlEvictorType method with one of three values: CREATION_TIME, LAST_ACCESS_TIME, or NONE.

- **TimeToLive**

  This attribute is used to specify the number of seconds that the built-in time-to-live evictor requires to add to the creation or last access time for each entry as described for the TtlEvictorType attribute. If this attribute is not set, then the special value of zero is used to indicate the time to live is infinity. If this attribute is set to infinity, map entries are never evicted by the evictor.

### Sessions

When you create a grid, it does not access the data in BackingMaps directly. Your applications must create Sessions which act like temporary workspaces. All operations in Sessions take place within the context of transactions. When you execute a ‘get’ in your application to data from the BackingMap into the session, the operation must take place between the Session begin() and commit().

When you commit the transaction the working copy in the Session replaces the copy in the BackingMap. You can rollback transactions, discarding the working copy and reverting to the copy in the BackingMap.

An application can have only one active Session per ObjectGrid instance, but it can access one or more ObjectMaps within the Session.
ObjectMap

The ObjectMap contains the uncommitted application data inserted, to be updated or deleted between the Session begin() and commit(). eXtreme Scale provides three modes which determines if and when copies are made of objects in the BackingMap. The modes are:

- COPY_ON_READ_AND_COMMIT
  When you use this mode, the data retrieved by calling “get” method is created in the Session by copying the object in the BackingMap. When the transaction is committed, changes made by the application to the object are copied back to the BackingMap. This mode provides the best data integrity but it has the slowest performance because it makes a deep copy of the object every time a read or commit is performed.

- COPY_ON_READ
  This mode is similar to ‘COPY_ON_READ_AND_COMMIT’ in that data is copied from the BackingMap into the Session. However when the transaction is committed the working copy of the object from the Session is placed directly into the BackingMap, replacing the original. To ensure data integrity, this mode requires that the application guarantee that objects are not reused after a transaction is committed; doing so could corrupt data in the BackingMap. This mode provides better performance when compared to the COPY_ON_READ_AND_COMMIT mode because data is not copied when the commit operation is performed.

- COPY_ON_WRITE
  This mode requires your application to access objects indirectly using a Java dynamic proxy. It provides the best performance while still ensuring data integrity in read-most scenarios, which are very common. The proxy creates a copy of the object only if the application modifies it by calling a “set” method. Otherwise, “read” operations are deferred to the object instance in the BackingMap. When the transaction commits, if a copy was created, it is placed in the BackingMap.

2.2 eXtreme Scale application programming interfaces

eXtreme Scale provides a set of Application Programming Interfaces (APIs) for programmatically accessing eXtreme Scale features using the Java programming language. The APIs are characterized as either transactional, where all operations using the API are performed within transaction contexts, or non-transactional, where the operations using the API do not require a transaction context. As illustrated in Figure 2-2 on page 22, the ObjectMap, EntityManager, Query, and eXtreme Scale APIs are transactional APIs and must
be executed with a session and transaction context. The grid, BackingMap, and plug-in APIs including configuration operations are non-transactional.

Figure 2-2  eXtreme Scale programming interface overview

Note: Plug-ins are components used to customize the functionality of the grid and BackingMap.
The eXtreme Scale APIs are grouped into a set of categories. They include the following:

- **eXtreme Scale core programming API**
  
  This API supports the fundamental interaction with the eXtreme Scale cache. The core API is essential to transactional operations, though it is non-transactional.

- **ObjectMap API**
  
  The ObjectMap API is similar to the java.util.Map interface. It provides a transactional map-based API for CRUD (Create, Read, Update, and Delete) operations.

- **EntityMap API**
  
  The EntityManager API supports the entity programming model for interacting with complex graphs of related objects using Map-based infrastructure.

- **Query APIs**
  
  Query APIs provide two sets of APIs for finding data in the eXtreme Scale cache. With Object Query API, you perform traditional queries against ObjectMap and EntityManager data, and eXtreme Scale Stream Query API, supports continuous queries over streaming data.

- **DataGrid API**
  
  The DataGrid API provides the interface to run business logic in parallel over all or a subset of the data in the grid.

- **eXtreme Scale system programming API**
  
  The eXtreme Scale system programming API provides a programming model for extending the grid through plug-ins. With this API, you can develop plug-ins that provide additional grid functionality.

---

**Note:** For more information about the com.ibm.websphere.objectgrid package which includes the application APIs, see:

2.2.1 Package com.ibm.websphere.objectgrid interface summary

- BackingMap
  This is the public interface to the BackingMap.

- ClientClusterContext
  This interface is a context to represent which cluster/domain the client connected to using one of the ObjectGridManager.connect methods.

- ClientReplicableMap
  This interface represents a replicable client map.

- IObjectGridException
  This interface is used to ensure JDK™ 1.4 Throwable chaining behavior for all exceptions thrown by the grid even when an earlier JDK is used.

- JavaMap
  This interface is a handle to a named Map.

- ObjectGrid
  This object is used for creating sessions to the grid.

- ObjectGridAdministrator
  This interface allows users to call system management functions on a grid cluster from a client process.

- ObjectGridManager
  ObjectGridManager is responsible for creating or getting an ObjectGrid instance and getting the configuration.

- ObjectMap
  This is a handle to a named Map.

- PartitionManager
  This interface will be used for calculating the proper partition for a given input key.

- Session
  This interface represents a session container for ObjectMaps.

- TxID
  This interface is an opaque identifier for a transaction.
2.2.2 Package com.ibm.websphere.objectgrid class summary

- ClientReplicableMap.Mode
  Client Replication mode.
- CopyMode
  This class is used to define the “copy” mode when the setCopyMode method of the BackingMap interface is used.
- HostPortConnectionAttributes
  A HostPortConnectionAttributes object represents a host name and port number pairing.
- LockStrategy
  LockStrategy provides an enumerated type idiom for use on the BackingMap.setLockStrategy(LockStrategy) method.
- ObjectGridManagerFactory
  This factory class is a high level helper class to get ObjectGridManager instances.
- TTLType
  Every BackingMap has a built-in timed based evictor referred to as “time to live” evictor or TTL evictor.

2.2.3 eXtreme Scale core programming API

The eXtreme Scale core programming API supports the fundamental interactions with the eXtreme Scale cache. It includes obtaining ObjectGrid instances and configuring grids and BackingMaps.

Your application must obtain the ObjectGrid instance in order to work with a grid. Having obtained the instance, the application can then call the ObjectGrid.getSession method on a thread to get the Session for the thread.
Obtaining ObjectGrid instance

Example 2-1 shows an example code fragment for obtaining ObjectGrid instance. The instance should be stored in a static variable.

Example 2-1 Example obtaining local ObjectGrid instance

```java
// Obtain a local ObjectGrid reference
// you can create a new ObjectGrid, or get configured ObjectGrid defined in
// the objectGrid XML file

ObjectGridManager objectGridManager =
    ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid ivObjectGrid = objectGridManager.createObjectGrid("objectgridName");
```

The following code fragment shows how to obtain a catalog server-based distributed grid.

Example 2-2 Obtaining a catalog server-based distributed grid

```java
ObjectGridManager ogManager = ObjectGridManagerFactory.getObjectGridManager();

// Obtain a ClientClusterContext by connecting to a catalog server based
// distributed ObjectGrid
// You have to provide a connection end point in the format of
// hostName:endPointPort
// The hostName is the machine where the catalog server reside.
// The endPointPort is the catalog server listening port, the default is
// 2809.

ClientClusterContext ivClientClusterContext = ogManager.connect("localhost:2890",
                  null, null);

// Obtain distributed ObjectGrid via ObjectGridManager by providing a
// ClientClusterContext

ObjectGrid ivObjectGrid = ogManager.getObjectGrid(ivClientClusterContext,
                          "objectgridName");
```

Client ObjectGrid instance

In a distributed grid application, you use one of the ObjectGridManager.connect methods to obtain an ObjectGrid instance. The method you use depends on the environment.

- Use the ObjectGridManager.connect(ClientSecurityConfiguration
  securityProps, URL overRideObjectGridXml)method in the WebSphere
Application Server when the eXtreme Scale server runtime is running in the same cluster.

- Use the `ObjectGridManager.connect(String catalogServerAddresses, ClientSecurityConfiguration securityProps, URL overRideObjectGridXml)` method to connect to a remote collection of eXtreme Scale containers managed by an eXtreme Scale catalog server.

**Note:** The other three `connect` methods on the `ObjectGridManager` interface are used for connecting to static deployment topology eXtreme Scale server runtime.

Each connect method returns a `ClientClusterContext` instance which can be used to get the reference to any remote or collocated grid. You use the method `ObjectGridManager.connect(ClientClusterContext context)` to obtain a reference to the grid. The following examples illustrate these methods:

**Example 2-3  Connecting to a remote grid**

```java
ObjectGridManager ogm = ObjectGridManagerFactory.getObjectGridManager();
ClientClusterContext remoteContext =
    ogm.connect("MyServer.company.com:2809", null, null);
ObjectGrid remoteGrid = ogm.getObjectGrid(remoteContext, "MyObjectGrid");
```

**Example 2-4  Connecting to a co-located grid**

```java
ObjectGridManager ogm = ObjectGridManagerFactory.getObjectGridManager();
ClientClusterContext colocatedContext = ogm.connect(null, null);
ObjectGrid colocatedGrid = ogm.getObjectGrid(colocatedContext, "MyObjectGrid");
```
Server ObjectGrid instance
You make use of two interfaces to obtain an instance of a grid running on an eXtreme Scale server.

- ObjectGridEventListener

ObjectGridEventListeners are custom classes that implement the ObjectGridEventListener interface. The interface is used to receive notifications of life cycle events on the grid. The events include the initialization and destruction of grids and the start and end of transactions. To listen for these events, create a class that implements the ObjectGridEventListener interface and add it to the grid.

**Note:** ObjectGridEventListener is a plug-in. It implements the ObjectGridEventListener interface which provides life cycle events for the grid, shards, and transactions. Plug-ins are used to add integration and customization points for applications and cache providers.

A grid can have multiple ObjectGridEventListener listeners. Use the addEventListener(), setEventListeners(), and removeEventListener() methods on the ObjectGrid interface to add or remove ObjectGridEventListener listeners.

**Example 2-5  Programmatically plug in an ObjectGridEventListener**

```java
ObjectGridManager objectGridManager =
ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid myGrid = objectGridManager.createObjectGrid("myGrid", false);
MyObjectGridEventListener myListener = new MyObjectGridEventListener();
myGrid.addEventListener(myListener);
```

The initialize(Session session) method on the ObjectGridEventListener is invoked when a grid is initialized. A Session instance is passed to the listener to provide access to the grid objects.
Example 2-6  Example code fragment to obtain a grid from a session object

```java
public void initialize(Session session) {
    ObjectGrid myGrid;
    mySession = session;

    myGrid = session.getObjectGrid();
    try {
        ...
    }
    catch (Throwable e) {
        throw new ObjectGridRuntimeException("Cannot initialize", e);
    }
}
```

ObjectGridEventGroup.ShardEvents

ObjectGridEventGroup.ShardEvents is one of three sub-interfaces of ObjectGridEventGroup, a set of single-method interfaces for fine-grained events delivered for grids. The interface declares shardActivate(ObjectGrid grid) and shardDeactivate(ObjectGrid grid) methods. The methods are called when a shard is activated or deactivated as a primary shard. The grid reference is provided with each call, giving your application access to the ObjectGrid instance.

Note: It is recommended that each thread in your application obtain its own Session. You can call one of the getSession() methods on the ObjectGrid instance to obtain the Session.

Using Sessions to access data

When you have obtained an ObjectGrid instance, you have two choices for accessing and manipulating the data stored in the grid. The first is the Map-based API, which can be obtained by the Session.getMap() method. The second is the EntityManager APIs, obtained by using the Session.getEntityManager() method.

The Map API

The Map API has the usual Map operations such as put, get and remove. However, the use of more specific verbs: get, getForUpdate, insert, update, and remove is preferred. These methods convey the intent more precisely that the traditional Map APIs.

Example 2-7 on page 30 shows an example of inserting data in a grid using the MAP API.
Example 2-7  Example code fragment using the Map API to insert data in a grid

```java
ObjectGridManager objectGridManager =
ObjectGridManagerFactory.getObjectGridManager();
    ObjectGrid myGrid = objectGridManager.createObjectGrid("myGrid", false);
    Session mySession = myGrid.getSession();
    ObjectMap customerMap = mySession.getMap("CUSTOMER");
    mySession.begin();
    Customer cust = new Customer();
    cust.customerID = "01";
    cust.firstName = "John";
    cust.lastName = "Doe";
    cust.address = "Main Street";
    cust.taxID = "123-456-7890";
    cust.email = "john.doe@email.com";
    customerMap.insert(cust.customerID, cust);

    mySession.commit();
```

- The EntityManager APIs

  The EntityManager APIs are simpler when compared to the Map APIs. When making an EntityManager find() call, the grid finds the Entity and returns a managed object. The grid automatically tracks changes to the managed object and at commit time performs the necessary updates. Example 2-8 shows an example of inserting data in a grid using EntityManager API.

Example 2-8  Example code fragment using the EntityManager API to insert data in a grid

```java
ObjectGridManager objectGridManager =
ObjectGridManagerFactory.getObjectGridManager();
    ObjectGrid myGrid = objectGridManager.createObjectGrid("myGrid", false);
    Session mySession = myGrid.getSession();
    EntityManager myEntityMgr = mySession.getEntityManager();
    mySession.begin
    Customer cust = new Customer();
    cust.id = "01";
    cust.firstName = "John";
    cust.surname = "Doe";
    cust.address = "Main Street";
    cust.taxID = "123-456-7890";
    cust.email = "john.doe@email.com";
    myEntityMgr.persist(cust);

    mySession.commit();
```
Transactions

Operations within Sessions must take place within transactions. A transaction is an atomic unit of replication which allows for a set of changes to be made atomically. Locks are applied to the data and held until the transaction is committed or rolled back.

Transactions are used for the following reasons:

- Protect a thread from concurrent changes
- Apply multiple changes as an atomic unit at commit time
- Implement a life cycle for locks on changes
- Act as the unit of replication

eXtreme Scale lets you customize transactions in Sessions. You can turn off rollback support and locking; your application must then handle the lack of these features. You can also modify how objects changed in a transaction are copied. Specify this using the ObjectMap.setCopyMode() method. The available copy modes are:

- COPY_ON_READ_AND_COMMIT
  
The COPY_ON_READ_AND_COMMIT mode is the default mode. This mode ensures that your application does not contain a reference to the value object in the BackingMap. Instead, your application is always working with a copy of the value in the BackingMap. This mode ensures that your application can never inadvertently corrupt the data cached in the BackingMap. When ObjectMap.get() method is first called in a transaction, a copy of the value is returned. When the transaction is committed, all changes are copied to the BackingMap to ensure that the application does not have a reference to the committed value in the BackingMap.

- COPY_ON_READ mode
  
The COPY_ON_READ mode improves performance over the COPY_ON_READ_AND_COMMIT mode by eliminating the copy that occurs when a transaction is committed. To preserve the integrity of the BackingMap data, your application should ensure that every reference it has for an entry is destroyed after the transaction is committed. With this mode, the ObjectMap.get() method returns a copy of the value instead of a reference to the value to ensure that changes made by the application to the value do not affect the BackingMap value until the transaction is committed. However, when the transaction does commit, a copy of changes is not made. Instead, a reference to the copy returned by the ObjectMap.get() method is stored in the BackingMap. Your application should destroy all map entry references after the transaction is committed or you run the risk of corrupting the data cached in BackingMap. If you are
having problems using this mode switch to
COPY_ON_READ_AND_COMMIT mode to see if the problem is resolved. If so, then you were failing to destroy all of its references after the transaction has committed.

- COPY_ON_WRITE

The COPY_ON_WRITE mode improves performance over the COPY_ON_READ_AND_COMMIT mode by eliminating the copy that occurs when the ObjectMap.get() method is called for the first time in a transaction. The call returns a proxy to the value instead of a direct reference to the value object. The proxy ensures that a copy of the value is not made unless the application calls a set method on the value interface specified by the valueInterfaceClass argument. The proxy provides a copy-on-write implementation. When a transaction commits, the BackingMap examines the proxy to determine if any copy was made as a result of a set method being called. If a copy was made, then the reference to that copy is stored in the BackingMap. The advantage of this mode is that a value is never copied on a read or at a commit when the transaction never calls a set method to change the value.

- Partitions and transactions

eXtreme Scale transactions can only update a single partition. Transactions from a client can read from multiple partitions but can update only one. If an application attempts to update two partitions the transaction will fail and be rolled back. A transaction using an embedded grid (grid logic) has no routing capability and can see only data in that partition. This business logic can always get a second true client session to access other partitions but this would be an independent transaction.

2.2.4 ObjectMap API

ObjectMaps are like Java Maps that allow data to be stored as key/value pairs. ObjectMaps provide a simple and intuitive approach for the application to store data. An ObjectMap is ideal for caching objects that have no relationships involved. The ObjectMap API provides a transactional map-based API that allows typical CRUD (Create, Read, Update, and Delete) operations ObjectMaps. Similar to the java.util.Map interface, it adds cache-specific semantics such as batch access methods, entry eviction timeouts, locking hints, and entry copy mode overrides. (for safety and performance)

Two interfaces, ObjectMap and JavaMap, are used for transactional interaction between applications and BackingMaps.
ObjectMap interface
An ObjectMap instance is obtained from a Session object that corresponds to the current thread. The ObjectMap interface is the main vehicle used to make changes to entries in a BackingMap.

Obtain an ObjectMap instance
An application gets an ObjectMap instance from a Session object using the Session.getMap(String) method. The following code fragment shows how to obtain an ObjectMap instance:

```java
Example 2-9  Example code fragment on obtain an ObjectMap instance
ObjectGridManager objectGridManager =
ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid myGrid = objectGridManager.createObjectGrid("myGrid", false);
BackingMap myBackingMap = myGrid.defineMap("financeMap");
Session mySession = myGrid.getSession();
ObjectMap objectMap = mySession.getMap("financeMap");
```

Each ObjectMap instance corresponds to a particular Session object. Calling the Session.getMap() method multiple times on a particular Session object with the same BackingMap name will return the same ObjectMap instance.

JavaMap interface
A JavaMap instance is obtained from an ObjectMap object. The JavaMap interface has the same method signatures as ObjectMap, but with different exception handling. JavaMap extends the java.util.Map interface, so all exceptions are instances of the java.lang.RuntimeException class. Because JavaMap extends the java.util.Map interface, it is easy to use a grid with an existing application that uses a java.util.Map interface for object caching.
Obtain a JavaMap instance

An application gets a JavaMap instance from an ObjectMap object using the ObjectMap.getJavaMap() method. The following code fragment shows how to obtain a JavaMap instance.

Example 2-10  Example code fragment on obtain an JavaMap instance

```java
ObjectGridManager objectGridManager = 
ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid myGrid = objectGridManager.createObjectGrid("myGrid", false);
BackingMap myBackingMap = myGrid.defineMap("financeMap");
Session mySession = myGrid.getSession();
ObjectMap objectMap = mySession.getMap("financeMap");
java.util.Map myMap = objectMap.getJavaMap();
JavaMap myJavaMap = (JavaMap) myMap;
```

A JavaMap is backed by the ObjectMap from which it was obtained. Calling getJavaMap multiple times using a particular ObjectMap will return the same JavaMap instance.

The JavaMap interface only supports a subset of the methods on the java.util.Map interface. The the java.util.Map interface supports the following methods:

- containKey(java.lang.Object)
- get(java.lang.Object)
- put(java.lang.Object, java.lang.Object)
- putAll(java.util.Map)
- remove(java.lang.Object)

All other methods inherited from the java.util.Map interface result in a java.lang.UnsupportedOperationException exception.
Note: ObjectMap API interface method semantics

Following is an explanation of the semantics behind methods on the ObjectMap and JavaMap interfaces. See the JavaDoc API documentation at:


▶ containsKey

The containsKey method determines if a key has a value in the BackingMap or Loader. If null values are supported by an application, this method can be used to determine if a null reference returned from a get operation refers to a null value or indicates that the BackingMap and Loader do not contain the key.

▶ flush

The semantics of this method are similar to the flush method on the Session interface. The notable difference is that the Session flush applies the current pending changes for all of the maps modified in the current session. With this method, only the changes in this ObjectMap are flushed to the loader.

▶ get

The get method fetches the entry from the BackingMap. If the entry is not found in the BackingMap but a Loader is associated with the BackingMap, it attempts to fetch the entry from the Loader. The getAll method is provided to allow batch fetch processing.

▶ getForUpdate

The getForUpdate method is the same as the get method, but using the getForUpdate method tells the BackingMap and Loader that the intention is to update the entry. A Loader can use this hint to issue a SELECT for UPDATE query to a database backend. If a Pessimistic LockingStrategy is defined for the BackingMap, the lock manager locks the entry. The getAllForUpdate method is provided to allow batch fetch processing.

▶ insert

Inserts an entry into the BackingMap and the Loader. Using this method tells the BackingMap and Loader you want to insert a previously nonexistent entry. When you invoke this method on an existing entry, an exception occurs when the method is invoked or when the current transaction is committed.
The semantics of the invalidate method depend on the value of the isGlobal parameter passed to the method. The invalidateAll method is provided to allow batch invalidate processing.

Local invalidation is specified when the value false is passed as the isGlobal parameter of the invalidate method. Local invalidation discards any changes to the entry in the transaction cache. If the application issues agetmethod, the entry is fetched from the last committed value in the BackingMap. If no entry is present in the BackingMap, the entry is fetched from the last flushed or committed value in the Loader. When a transaction is committed, any entries marked as locally invalidated have no impact on the BackingMap. Any changes flushed to the Loader are still committed even if the entry was invalidated.

Global invalidation is specified when true is passed as the isGlobal parameter of the invalidate method. Global invalidation discards any pending changes to the entry in the transaction cache and bypasses the BackingMap value on subsequent operations performed on the entry. When a transaction is committed, any entries marked as globally invalidated are evicted from the BackingMap.

Consider the following use case for invalidation as an example: The BackingMap is backed by a database table with an auto increment column. Increment columns are useful for assigning unique numbers to records. The application inserts an entry. After the insert, the application must know the sequence number for the inserted row. It knows that its copy of the object is old, so it uses global invalidation to get the value from the Loader. The following code demonstrates this use case:

```java
Session mySession = objectGrid.getSession();
ObjectMap myMap = mySession.getMap("financeMap");
mySession.begin();
myMap.insert("01", new Customer("John", "Doe", "Main Street", "123-456-7890", "john.doe@email.com");
mySession.flush();
myMap.invalidate("01", true);
Customer cust = myMap.get("01");
System.out.println("The version is: " + cust.getVersion());
myMap.commit();
```
This code sample adds an entry for “John Doe.” The version attribute of Customer is set using an auto-increment column in the database. The application first performs an insert command. It then issues a flush, causing the insert to be sent to the Loader and database. The database sets the version column to the next number in the sequence, making the Customer object in the transaction outdated. To update the object, the application performs a global invalidate. The next get method issued gets the entry from the Loader ignoring the transaction's value. The entry is fetched from the database with the updated version value.

- put

The semantics of the put method are dependent on whether a previous get method was invoked in the transaction for the key. If the application issues a get operation that returns an existent entry in the BackingMap or Loader, the put method invocation is interpreted as an update and returns the previous value in the transaction. A put method invocation without a previous get method invocation or a previous get method invocation that did not find an entry is interpreted as an insert. The semantics of the insert and update methods apply when the put operation is committed. The putAll method is provided to enable batch insert and update processing.

- remove

The remove method removes the entry from the BackingMap and the Loader, if one is plugged in. The value of the object removed is returned by this method. If the object does not exist, this method returns a null value. The removeAll method is provided to enable batch deletion processing without the return values.

- setCopyMode

The setCopyMode method specifies a CopyMode for this ObjectMap. With this method, an application can override the CopyMode specified on the BackingMap. The specified CopyMode is in effect until clearCopyMode method is invoked. Both methods are invoked outside of transactional bounds. A CopyMode cannot be changed in the middle of a transaction.

- touch

Updates the last access time for an entry. This method does not retrieve the value from the BackingMap. Use this method in its own transaction. If the provided key does not exist in the BackingMap due to invalidation or removal, an exception occurs during commit processing.
The EntityManager API provides interfaces for interacting with the eXtreme Scale cache through complex graph related of objects. The EntityManager API uses the Map-based infrastructure. It converts entity objects into key tuples and value tuples, which are then stored as key/value pairs.

The EntityManager API eases the use of grids by following the Plain Old Java Object (POJO) style of programming adopted by most frameworks.

An entity instance can be in one of many states. The following are the states:

- **New**
  A newly-created entity instance that does not exist in the eXtreme Scale cache.

- **Managed**
  The entity instance exists in the eXtreme Scale cache and was retrieved or persisted using EntityManager. An entity must be associated with an active transaction to be in the managed state.

- **Detached**
  The entity instance exists in the eXtreme Scale cache, but is no longer associated with an active transaction.

- **Removed**
  The entity instance has been removed or is scheduled to be removed from the eXtreme Scale cache when the transaction is flushed or committed.
Operations using the EntityManager

You typically obtain the grid reference first in your application, and then a Session
from that reference. Using the Session.getEntityManager method, you are able
to obtain the reference to an EntityManager. Example 2-11 shows how to create
a local grid and access an EntityManager.

Example 2-11  Example code to obtain an EntityManager

```java
ObjectGridManager objectGridManager =
ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid myGrid = objectGridManager.createObjectGrid("myGrid", false);
Session mySession = myGrid.getSession();
EntityManager myEntityMgr = mySession.getEntityManager();
```

Persisting an Entity

Persisting an entity saves the state of a new entity in an eXtreme Scale cache.
Persist is a transactional operation, the new entity is stored in the eXtreme Scale
cache after the transaction commits.

Every entity has a corresponding BackingMap, in which the tuples are stored.
The BackingMap has the same name as the entity, and is created when the class
is registered. Example 2-12 demonstrates how to persist a Customer object.

Example 2-12  Example code to persist an Entity

```java
Session mySession = myGrid.getSession();
EntityManager myEntityMgr = mySession.getEntityManager();
myEntityMgr.getTransaction().begin();
    Customer cust = new Customer();
    cust.customerID = "01";
    cust.firstName = "John";
    cust.lastName = "Doe";
    cust.address = "Main Street";
    cust.taxID = "123-456-7890";
    cust.eMail = "john.doe@email.com";
myEntityMgr.persist(cust);
myEntityMgr.getTransaction().commit();
```

Finding an Entity

You can easily locate an entity in the eXtreme Scale cache with the key after it
has been stored by issuing the find method. This method does not require any
transactional boundary, which is useful for read-only semantics. Example 2-13 on
page 40 shows a one line code that locates a Customer entity.
Example 2-13  Example code for finding an entity

```java
Customer foundCust = (Customer)myEntityMgr.find(Customer.class,"01");
```

**Removing an Entity**

The EntityManager remove method, like the persist method, is a transactional operation. Example 2-14 shows a code fragment for removing an entity with a transactional boundary by calling `getTransaction().begin()` and `getTransaction().commit()`.

Example 2-14  Example code for removing an entity

```java
myEntityMgr.getTransaction().begin();
Customer foundCust = (Customer)myEntityMgr.find(Customer.class,"01");
myEntityMgr.remove(foundCust);
myEntityMgr.getTransaction().commit();
```

**Note:** The entity must first be managed before it can be removed, which is accomplished by calling the find method within the transactional boundary.

**Updating an Entity**

The EntityManager update method is also a transactional operation. The entity must be managed before any updates can be applied. Example 2-15 shows an updating of an entity. Note the persist method is not once the entity is updated. The entity is updated in the eXtreme Scale cache when the transaction is committed.

Example 2-15  Example code for updating an entity

```java
myEntityMgr.getTransaction().begin();
Customer foundCust = (Customer)myEntityMgr.find(Customer.class,"01");
foundCust.address = "101 main Street"; // update the customer address
myEntityMgr.getTransaction().commit();
```

**Entity schemas**

An entity schema is a set of entities and the relationships between them. Each grid can have one schema, and in a distributed grid, all entities must be included in a single MapSet. Entities are defined using annotated Java classes, XML, or a combination of both. Defined entities are then registered with a grid and bound to backing maps, indexes, and other plug-ins.
Entity definition and their relationships

Entities are defined by associating various metadata with a Java class. The metadata can be specified using Java SE 5 annotations, an entity metadata descriptor file, or a combination of both. When defining entity classes, consider the following criteria:

- The @Entity annotation must be defined or specified in the entity XML descriptor file.
- The entity must have a Public or Protected no-arg constructor.
- The entity must have a top-level class. (Interfaces and enums are not valid entity classes)
- Entity classes must not have the final modifier specified.
- Entity inheritance is not supported.
- The entity must have a unique name and type per grid.

Entities should have unique names and type. If you are using annotations, the default name of the entity is the name of the class. The name can be overridden using the name attribute of the @Entity annotation.

Attribute types

The following attribute types are supported:

- Java primitive types including wrappers
  - java.lang.String
  - java.math.BigInteger
  - java.math.BigDecimal
  - java.util.Date
  - java.sql.Date
  - java.sql.Time
  - java.sql.Timestamp
  - byte[]
  - java.lang.Byte[]
  - char[]
  - java.lang.Character[]

User-serializable attribute types are supported but have performance and change-detection limitations. Persistent data that cannot be proxied, such as arrays and user-serializable objects, must be reassigned to the entity if altered.
Entity associations

Bi-directional and uni-directional entity associations (relationships) between entities can be defined as one-to-one, many-to-one, one-to-many, and many-to-many. The EntityManager automatically decomposes the entity relationships to the appropriate key references when storing entities in the grid.

Note: The grid does not enforce referential integrity like a database. Although relationships allow cascading persist and remove operations to child entities, it does not detect or enforce broken links to objects. When removing a child object, the reference to that object must be removed from the parent.

A bi-directional association between two entities requires identifying the owner of the relationship. In a collection-valued association (one-to-many or many-to-many), the many-side of the relationship is the owner. If ownership can not be automatically determined, then the mappedBy attribute of the annotation (or XML equivalent) must be specified. The mappedBy attribute identifies the field in the target entity that owns the relationship. It also helps identity the related fields when there are multiple attributes of the same type and cardinality.

Single-valued associations

One-to-one and many-to-one associations are denoted using the @OneToOne and @ManyToOne annotations. (or XML equivalent) The target entity type is determined by the attribute type. In the following example, we have a uni-directional association between Customer and Address. The Customer entity has a reference to one Address entity. In this case, the association could also be many-to-one because there is no inverse relationship.

Example 2-16   Example single valued association

```java
@Entity
public class Customer {
    @Id String id;
    @OneToOne Address homeAddress;
}

@Entity
public class Address{
    @Id String id;
    @Basic String city;
}
```

To specify a bi-directional relationship between Customer and Address, simply add a reference to Customer from Address and add the appropriate annotation to mark the inverse side off the relationship. Because this is a one-to-one
association, we must to choose an owner of the relationship by specifying the mappedBy attribute on the @OneToOne annotation.

Example 2-17  Example bi-directional relationship

```java
@Entity
public class Address{
    @Id String id;
    @Basic String city;
    @OneToOne(mappedBy="homeAddress") Customer customer;
}
```

Collection-valued associations

One-to-many and many-to-many associations are denoted using the @OneToMany and @ManyToMany annotations (or XML equivalent) attributes. All collection-valued relationships are represented using the following types:

- java.util.Collection
- java.util.List
- java.util.Set

The target entity type is determined by the generic type of the Collection (List or Set), or explicitly using the targetEntity attribute on the @OneToMany or @ManyToMany annotation. (or XML equivalent)

In “Single-valued associations” on page 42, it isn't practical to have one address object per customer, as many customers may share an address or may have multiple addresses. This is better solved using a many association:

Example 2-18  Example collection valued associations

```java
@Entity
public class Customer {
    @Id String id;
    @ManyToOne Address homeAddress;
    @ManyToOne Address workAddress;
}

@Entity
public class Address{
    @Id String id;
    @Basic String city;
    @OneToMany(mappedBy="homeAddress") Collection<Customer>homeCustomers;
    @OneToMany(mappedBy="workAddress", targetEntity=Customer.class) Collection workCustomers;
}
This is a bit more complicated than usual, as you have two different relationships (home and work address) between the same entities. A non-generic Collection is used for the workCustomers attribute to demonstrate how to use the targetEntity attribute when generics are not available.

**Primary keys**

All entities must have a primary key which can be a simple (single attribute) or composite (multiple attribute) key. The key attributes are denoted using the `Id` annotation or defined in the entity XML descriptor file.

Key attributes have the following requirements:

- The value of a primary key must not change.
- A primary key attribute should be one of the following types:
  - Java primitive type and wrappers
  - `java.lang.String`, `java.util.Date`
  - `java.sql.Date`.
- A primary key may contain any number of single valued associations. The target entity of the primary key association must not have an inverse association directly or indirectly to the source entity.

Composite primary keys can optionally define a primary key class. An entity is associated with a primary key class using the `IdClass` annotation or the defined in the entity XML descriptor file. An `IdClass` is useful when used in conjunction with the `EntityManager.find` method.

Primary key classes have the following requirements:

- The primary key class must be public and have a public no-arg constructor.
- The access type of the primary key class is determined by the entity that declares the primary key class.
- If property-based access is used, the properties of the primary key class must be public or protected.
- The primary key fields or properties must match the key attribute names and types defined in the referencing entity.
- Primary key classes must implement `equals` and `hashCode`. 
Example 2-19Primary key class

```java
@Entity
@IdClass(CustomerKey.class)
public class Customer {
    @Id @ManyToOne Zone zone;
    @Id int custId;
    String name;
    // ...
}

@Entity
public class Zone{
    @Id String zoneCode;
    String name;
}

public class CustomerKey {
    Zone zone;
    int custId;
    public int hashCode() {...}
    public boolean equals(Object o) {...}
}
```

**Entity schema configuration**

The entity metadata descriptor file is an XML file used to define an entity schema for a grid. All of the entity metadata can be explicitly defined in the XML file, or as annotations on the entity Java class file. The primary use is for entities that cannot use Java annotations.

Use the XML configuration to create entity metadata based on the XML file. When used in conjunction with annotation, some of the attributes defined in the XML configuration override the corresponding annotations.

**Note:** For description of elements in Entity metadata descriptor file see:


### 2.2.6 Query APIs

eXtreme Scale provides a flexible query engine for retrieving entities using the EntityManager API and Java objects using the ObjectQuery API. The query engine allows SELECT type queries over an entity or object-based schema using
the eXtreme Scale query language. With the eXtreme Scale Stream Query API, you can perform continuous queries over streaming data.

**ObjectQuery API**
The ObjectQuery API provides methods for querying data in the grid stored using the ObjectMap API. When a schema is defined in the grid, the ObjectQuery API can be used to create and run queries over the heterogeneous objects stored in the ObjectMaps.

ObjectMap queries allow for retrieval of objects using non-key attributes and simple aggregations such as sum/avg/min/max against all the data that matches a query. You can construct a query using the Session.createObjectQuery API. It returns an ObjectQuery object which can then be interrogated to obtain the query results. The query object also allows the query to be customized before running the query. The query is run automatically when any method returning the result is called.

Figure 2-3 on page 47 illustrates the interaction of the query with the ObjectMaps and how a schema is defined for classes and associated with a map.
Defining an ObjectMap schema

ObjectQuery relies on schema or shape information to perform semantic checking and to evaluate path expressions. This section describes how to define the schema in XML or programmatically.

- Defining the schema

  The ObjectMap schema is defined in the eXtreme Scale deployment descriptor XML or programmatically using the normal grid configuration techniques.

**Note:** For an example on how to create a schema, see http://www.ibm.com/developerworks/wikis/display/objectgridprog/Baslic+ObjectQuery+Tutorial+Step+4
Schema information describes POJOs: which attributes they consist of and what types of attributes there might be, whether the attributes are primary key fields, single-valued or multi-valued relationships, or bidirectional relationships. Schema information directs ObjectQuery to use field access or property access.

- **Queryable attributes**

  When the schema is defined in the grid, the objects in the schema are introspected using reflection to determine which attributes are available for querying. The following attribute types can be queried:
  
  - Java primitive types including wrappers
  - java.lang.String
  - java.math.BigInteger
  - java.math.BigDecimal
  - java.util.Date
  - java.sql.Date
  - java.sql.Time
  - java.sql.Timestamp
  - byte[]
  - java.lang.Byte[]
  - char[]
  - java.lang.Character[]

  Embedded serializable types not previously mentioned can be included in a query result, but cannot be included in the where or from clause of the query. Serializable attributes are not navigatable.

  Attribute types can be excluded from the schema if the type is not serializable, the field or property is static, or the field is transient. Because all map objects must be serializable, the grid only includes persistable attributes from the object. Other objects are ignored.

  - **Field attributes**

    When the schema is configured to access the object using field attributes, all serializable, non-transient fields are automatically incorporated into the schema. To select a field attribute in a query, use the field identifier name as it exists in the class definition. All public, private, protected, and package protected fields are included in the schema.
- Property attributes

When the schema is configured to access the object using property attributes, all serializable methods that follow the JavaBeans™ property naming conventions will automatically be incorporated into the schema. To select a property attribute in a query, use the JavaBeans style property name conventions. All public, private, protected, and package protected properties are included in the schema.

In the following class, the following attributes are added to the schema: name, address, taxID.

*Example 2-20  Example customer class*

```java
public class Customer {
    public String getName() {}  
    private String getAddress() {}  
      private String getTaxID() {}  
    public NonSerializableObject getData() {}  
}
```

- Relationships

Each relationship must be explicitly defined in the schema configuration. The cardinality of the relationship is automatically determined by the type of the attribute. If the attribute implements the java.util.Collection interface, then the relationship is either a one-to-many or many-to-many relationship.

Unlike entity queries, attributes that refer to other cached objects must not store direct references to the object. References to other objects are serialized as part of the containing object’s data. Instead, store the key to the related object.

For example, if there is a many-to-one relationship between a Customer and Account:

*Example 2-21  Incorrect. Storing an object reference*

```java
public class Customer {  
    String customerId;  
    Collection<Account> accountss;
}

public class Account {  
    String accountId;  
    Customer customer;
}
```
Example 2-22  Correct. The key to the related object

```java
public class Customer {
    String customerId;
    Collection<String> accounts;
}

public class Account {
    String accountId;
    String customer;
}
```

When a query is run that joins the two map objects together, the key will automatically be inflated. For example, the following query would return Customer objects:

```sql
SELECT c FROM Account o JOIN Customer c WHERE accountId=5
```

- Using indices

Index plug-ins are used to add indices to maps. The query engine automatically incorporates any indices defined on a schema map element of the type: `com.ibm.websphere.objectgrid.plugins.index.HashIndex` and the `rangeIndex` property is set to true. If the index type is not `HashIndex` and the `rangeIndex` property is not set to true, then the index is ignored by the query.

**Querying objects with the ObjectQuery API**

The ObjectQuery interface allows querying non-entity objects; heterogeneous objects stored directly in the ObjectMaps. ObjectQuery provides an easy way to find ObjectMap objects without using the keyword and index mechanisms directly.

There are two methods for retrieving results from an ObjectQuery: `getResultIterator()` and `getResultMap()`.

**Retrieving query results using getResultIterator**

Query results are basically a list of attributes. Suppose the query was select a,b,c from X where y=z. This query returns a list of rows containing a, b, and c. This list is actually stored in a transaction scoped Map, which means you must associate an artificial key with each row and use an integer that increases with each row. This map is obtained using the `ObjectQuery.getResultMap()` method. You can access the elements of each row using code similar to Example 2-23 on page 51:
Example 2-23   Retrieving query results using getResultIterator

ObjectQuery myQuery = session.createObjectQuery(  
    "select c.id, c.firstName, c.surname from Customer c where  
    c.surname=?1";

    myQuery.setParameter(1, "Doe");

    Iterator iter = myQuery.getResultIterator();
    while(iter.hasNext())
    {
        Object[] row = (Object[])iter.next();
        System.out.println(" Found a Doe with id "  
            + row[0] + ", firstName: "
            + row[1] + ", surname: "
            + row[2]);
    }

Retrieving query results using getResultMap

Query results can also be retrieved using the result map directly. Example 2-24  
on page 52 shows a query retrieving specific parts of the matching Customers  
and demonstrates how to access the resulting rows. Notice that if you use the  
ObjectQuery object to access the data, then the generated long row identifier is  
hidden. The long row is only visible when using the ObjectMap to access the  
result.

When the transaction is completed this map disappears. The map is only visible  
to the session used. That is, to just the thread that created it. The map uses a key  
of type String which represents the row ID. The values stored in the map either  
are of type Object or Object[], where each element matches the type of the  
element in the select clause of query.
Example 2-24  Retrieving query results using getResultMap

```java
EntityManager myEntityManager;
Query myQuery = myEntityManager.createQuery(
    "select c.id, c.firstName, c.surname from Customer c where 
    c.surname=?1"");
myQuery.setParameter(1, "Claus");
ObjectMap myMap = myQuery.getResultMap();
for(long rowId = 0; true; ++rowId)
{
    Object[] row = (Object[]) myMap.get("01");
    if(row == null) break;
    System.out.println(" Found a Claus with id "
        + row[0]
        + ", firstName: " + row[1]
        + ", surname: " + row[2]);
}
```

EntityManager Query API

The EntityManager Query API provides methods for querying data in the grid stored using the EntityManager API. You construct queries using the EntityManager.createQuery() method. It returns a Query object and you can then interrogated the object to obtain the query results. You can customize the query object before running the query. The query is run automatically when any method that returns the result is called. Figure 2-4 on page 53 illustrates the interaction of the query with ObjectMaps and how the entity schema is defined and associated with a grid map.
Retrieving query results using getResultIterator

Query results are a list of attributes. If the query was select a,b,c from X where y=z, then a list of rows containing a, b and c is returned. This list is stored in a transaction scoped Map, which means that you must associated an artificial key with each row and use an integer that increases with each row. The Map is obtained by calling Query.getResultMap() method. The Map has EntityMetaData, which describes each row in the Map associated with it. You can access the elements of each row using code similar to that in Example 2-25 on page 54.
Example 2-25  Example of retrieving query results using getResultIterator

Query myQuery = myEntityManager.createQuery("select cust.id, cust.firstName, cust.surname from Customer cust where cust.surname=?1");

myQuery.setParameter(1, "Doe");

Iterator iter = myQuery.getResultIterator();
while(iter.hasNext())
{
    Object[] row = (Object[])iter.next();
}

 Retrieving query results using getResultMap

The example code fragment in Example 2-26 shows the retrieval of specific parts of the matching Customers and shows how to access the resulting rows. If you use the Query object to access the data, then the generated row identifier is hidden. The row identifier is only visible when using the ObjectMap to access the result. When the transaction is completed, then this Map will be disposed. The Map is only visible in the Session used to run the query, which is the thread that created it. The Map uses a Tuple for the key with a single attribute, a long with the row ID. The value is another tuple with an attribute for each column in the result set.

Example 2-26   Example retrieving query results using getResultMap

Query myQuery = myEntityManager.createQuery("select cust.id, cust.firstName, cust.surname from Customer cust where cust.surname=?1");
myQuery.setParameter(1, "Doe");
ObjectMap myQueryMap = myQuery.getResultMap();
Tuple keyTuple = myQueryMap.getEntityMetadata().getKeyMetadata().createTuple();
for(long i = 0; true; ++i)
{
    keyTuple.setAttribute(0, new Long(i));
    Tuple row = (Tuple)myQueryMap.get(keyTuple);
    if(row == null) break;

    System.out.println(" Found a Doe with id ", row.getAttribute(0) + ", firstName: " + row.getAttribute(1) + ", surname: " + row.getAttribute(2));
}
Retrieving query results using an entity result iterator

The following code shows the query and the loop that retrieves each result row using the normal Map APIs. The key for the Map is a Tuple. Construct one of the correct types using the createTuple method result in keyTuple. Try to retrieve all rows with rowIds from 0 onwards. When null is returned (indicating key not found), then the loop terminates. You set the first attribute of keyTuple to be the number of the row that you want to find. The value returned by get is also a Tuple with an attribute for each column in the query result. You then retrieve each attribute from the value Tuple using getAttribute.

Example 2-27  Example retrieving query results using an entity result iterator

```java
Query myQuery2 = myEntityManager.createQuery("select cust.id, cust.firstName, cust.surname from Customer cust where cust.surname=?1");
myQuery2.setResultEntityName("CustomerQueryResult");
myQuery2.setParameter(1, "Doe");

Iterator iter2 = myQuery2.getResultIterator(CustomerQueryResult.class);
while(iter2.hasNext())
{
    CustomerQueryResult row = (CustomerQueryResult)iter2.next();
    // firstName is the id not the firstName.
    System.out.println("Found a Doe with id " + row.id
    + ", firstName: " + row.firstName
    + ", surname: " + row.surname);
}
myEntityManager.getTransaction().commit();
```

By specifying the ResultEntityName on the query, it tells the query engine that you want to project each row to a specific object, CustomerQueryResult in this case. Example 2-28 illustrates the class:

Example 2-28  Customer query result object

```java
@Entity
public class CustomerQueryResult {
    @Id long rowId;
    String id;
    String firstName;
    String surname;
};
```
Notice that the each query row is returned as a `CustomerQueryResult` object rather than an `Object[]`. The result columns of the query are projected to the `CustomerQueryResult` object. This has a slightly longer runtime but is more readable.

**Note:** Query result entities should not be registered with the grid at startup. If so, then you will create a global Map with the same name, and the query will fail with an error indicating duplicate Map name.

### Stream query

A grid stream query is a continuous query over streaming data. The streaming data are the in-flight data stored in grid maps. At the heart of the stream query is the Stream Processing Technology (SPT) engine. It is designed for real-time high-performance mission-critical operations. Figure 2-5 shows the diagram of how an SPT engine is used in grid environment.

![Stream query runtime architecture](image)

*Figure 2-5 The stream query runtime architecture*

The SPT engine uses the concepts of streams and views. A stream represents a stream of raw incoming data. There can be any number of streams and the streams can be updated at different frequencies. Each stream is associated with a grid stream map. Whenever there is an insertion or update to the stream map, a stream event is generated. Each stream has its own schema to define the data and type. The schema is defined using a special SPTSQL statement, which has similar syntax as SQL. In the following text, SQL and SPTSQL have the same meaning.
The output from the SQL statements are called views. The SQL statements define the processing rules. The rules can be complex. For example, you can:

- Join multiple streams
- Apply aggregation and computations
- Apply window operations
- Group results

The SQL statements are analyzed, and a graph is constructed. (See Figure 2-5 on page 56) Each node has an input and output. The input of the stream nodes are the grid stream maps, and the output are grid view maps. The intermediate nodes connect to each other to apply the processing rules.

Each time a stream is updated, the SPT engine propagates the changes through the nodes, and finally updates the views. The view changes (insertions, updates, and deletions) are then published to grid maps.

Stream queries are similar to database queries in how they analyze the data. The difference is that they are event driven, operating continuously on data as it arrives and updates the results in real-time. For an example of stream query see the “Hello World” stream query example at http://www.ibm.com/developerworks/wikis/display/objectgridprog/Hello+World+stream+query+example.

Note: For the Stream Engine language tutorial see: http://www.ibm.com/developerworks/wikis/display/objectgridprog/Stream+query+engine+language+tutorial

2.2.7 DataGrid API

Increasingly, it is required to search gigabytes or terabytes of data in seconds. Conventional vertical scaling approaches have not proven capable of handling this issue. eXtreme Scale architecture solves the problem by using a radically different approach. By partitioning the data, the eXtreme Scale can support large grids running on thousands of servers. The DataGrid API provides the programming interface to run business logic over all or a subset of the partitions in the grid in parallel.

Application and data co-location

Grids work best when the business logic is co-located with the partitioned data. Clients invoke the business logic using DataGrid API. The business logic is invoked on the appropriate partitions and the results are returned to the client.
This avoids the cost of transferring the data to the application server which in some cases outweigh the cost of processing the data.

**Process entries in parallel with entry specific results**
eXtreme Scale allows a client to ask for an application agent to perform operations on specific entries. The entries are specified using keys or can be specified using a query. The grid then returns a Map holding the keys of entries processed and the result for those entries. The grid receives the request from the client, determines which partitions have the data for the keys, then runs the application logic in parallel to get the results. When all partitions have processed their entries, the results are returned to the client. This pattern works best when the data returned is small enough to be processed by a single client. For example, this pattern works well for finding customers matching a specific query.

**Process groups of entries and aggregate to a single result**
Rather than transferring the results data back to the client, the data is processed as much as possible on partition nodes, taking advantage of the power of the grid. It minimizes the amount of data finally returned to the client. For example, if a client wants to know the total of the salaries for all customers held in the grid, then the sum of the customers for each partition is calculated on the partition, the results are aggregated and returned to the client. This allows the power of the grid to be used to handle the intermediate results also.

**Cross-partition query support**
Typically, a client interacts with a single partition at a time. For example, if you have a grid that stores employees, each employee has an associated department, and you want to partition the grid on an employee, how do you find all the employees in a certain department? This is a cross-partition query and the eXtreme Scale APIs can solve this with a parallel query.

### 2.2.8 Package com.ibm.websphere.objectgrid.datagrid interface summary

- **AgentManager**
  The AgentManager is the primary interface for submitting MapGridAgent or ReduceGridAgent instances to the grid.

- **EntityAgentMixin**
  This interface can be implemented by an agent.
EntryErrorValue

This is returned as the value when the agent throws any exception in the process method.

MapGridAgent

A MapGridAgent is used to process operations against map entries in a remote (server-side) grid.

ReduceGridAgent

A ReduceGridAgent is used to process a set of entries and reduce them to a single result.

Example using the eXtreme Scale API

The eXtreme Scale APIs support two common grid programming patterns. The first is called parallel Map and the second is parallel reduce:

Parallel Map

The parallel map allows the entries for a set of keys to be processed and returns a result for each entry processed. The application makes a list of keys and receives a Map of key/result pairs after invoking a Map operation. The result is the application of a function to the entry of each key. The function is supplied by the application.

Example 2-29  Customer class

```java
import com.ibm.websphere.projector.annotations.Entity;
import com.ibm.websphere.projector.annotations.Id;

@Entity
public class Customer{
    @Id String ssn;
    String firstName;
    String surname;
    int age;
}
```

The application supplied function is written as a class implementing the MapAgentGrid interface. Example 2-30 on page 60 is an example agent showing a function to return the age of a Customer multiplied by two.
Example 2-30  Double customer age agent

```java
public class DoubleCustomerAgeAgent implements MapGridAgent, EntityAgentMixin {
    private static final long serialVersionUID = -2006093916067992974L;

    int lowAge;
    int highAge;

    public Object process(Session sesn, ObjectMap map, Object key) {
        Customer cust = (Customer) key;
        return new Integer(cust.age * 2);
    }

    public Map processAllEntries(Session sesn, ObjectMap map) {
        EntityManager myEntityManager = sesn.getEntityManager();
        Query myQuery = myEntityManager.createQuery("select cust from Customer cust where cust.age > ?1 and cust.age < ?2");
        myQuery.setParameter(1, lowAge);
        myQuery.setParameter(2, highAge);
        Iterator iter = myQuery.getResultIterator();
        Map<Customer, Integer> rc = new HashMap<Customer, Integer>();
        while (iter.hasNext()) {
            Customer cust = (Customer) iter.next();
            rc.put(cust, (Integer) process(sesn, map, cust));
        }
        return rc;
    }

    public Class getClassForEntity() {
        return Customer.class;
    }
}
```

The first process method in `DoubleCustomerAgeAgent()` is supplied with the Customer to work with. It simply returns double the age of that entry. The second process method is called for each partition and finds all Customer objects with an age between `lowAge` and `highAge` and returns their ages doubled.
Example 2-31 is an example showing how to find the doubled ages for a couple of Customer.

**Example 2-31 Example code using DoubleCustomerAgeAgent**

```java
Session sesn = grid.getSession();
ObjectMap map = sesn.getMap("Customer");
AgentManager amgr = map.getAgentManager();

DoubleCustomerAgeAgent agent = new DoubleCustomerAgeAgent();

// make a list of keys
ArrayList<Customer> keyList = new ArrayList<Customer>();
Customer cust = new Customer();
cust.ssn = "123-45-6789";
keyList.add(cust);
cust = new Customer();
cust.ssn = "123-45-6780";
keyList.add(cust);

// get the results for those entries
Map = amgr.callMapAgent(agent, keyList);
```

The client obtains a Session and a reference to the Customer Map. The agent operation is performed against a specific Map. The AgentManager interface is retrieved from that Map. An instance of the agent to invoke is created and any necessary state is added to the object by setting attributes, there are none in this case. A list of keys are then constructed. In this case the keys indicate that we want to work on customers with social security numbers of "123-45-6789" and "123-45-6780".

The agent is then invoked for that set of keys. The agents process method is invoked on each partition with some of the specified keys in the grid in parallel. A Map is returned providing the merged results for the specified key. In this case, a Map with the values holding the age for customer 1 doubled and the same for customer 2 will be returned.

If the key does not exist, the agent will still be invoked. This gives the agent the opportunity to create the map entry. If using an EntityAgentMixin, the key to process will not be the entity, but will be the actual Tuple key value for the entity.

If the keys are unknown then it is possible to ask all partitions to find Customer objects of a certain shape and return their ages doubled. Example 2-32 on page 62 is an example.
Example 2-32  Constructing and initializing agent

```java
Session sesn = grid.getSession();
    ObjectMap map = sesn.getMap("Customer");
    AgentManager amgr = map.getAgentManager();

    DoubleCustomerAgeAgent agent = new DoubleCustomerAgeAgent();
    agent.lowAge = 20;
    agent.highAge = 9999;

    Map myMap = amgr.callMapAgent(agent);
```

It shows the AgentManager being obtained for the Customer Map. It then shows the agent being constructed and initialized with the low and high ages for Customers of interest. The agent is then invoked using the callMapAgent method. Notice no keys are supplied. This causes the grid to invoke the agent on every partition in the grid in parallel and return the merged results to the client. This will find all Customer objects in the grid with an age between low and high and calculate the age of those Customer objects doubled. This shows how the grid APIs can be used to run a query to find entities matching a certain query.

The agent is simply serialized and transported by the grid to the partitions with the required entries. The results are similarly serialized for transport back to the client.

**Note:** Care must be taken with the Map APIs. If the grid was hosting terabytes of objects and running on a lot of servers then potentially this would overwhelm anything but the largest machines running the client.

This should be used to process a small subset. If a large subset requires processing, we recommend using a reduce agent to process out in the grid rather than on a client.

**Parallel reduction or aggregation agents**

This style of programming processes a subset of the entries and calculates a single result for the group of entries. Examples of such a result would be:

- Minimum value
- Maximum value
- Some other business specific function

A reduce agent is coded and invoked in a similar manner to the Map agents. Example 2-33 on page 63 is an example of a reduce agent that simply adds the ages of the matching entries.
public class SumAgeReduceAgent implements ReduceGridAgent, EntityAgentMixin {

    private static final long serialVersionUID = 2521080771723284899L;

    int lowAge;
    int highAge;

    public Object reduce(Session sesn, ObjectMap map, Collection keyList) {
        Iterator<Customer> iter = keyList.iterator();
        int sum = 0;
        while(iter.hasNext()) {
            Customer cust = iter.next();
            sum += cust.age;
        }
        return new Integer(sum);
    } 

    public Object reduce(Session sesn, ObjectMap map) {
        EntityManager myEntityManager = sesn.getEntityManager();
        Query myQuery = myEntityManager.createQuery("select cust from Customer cust 
        where cust.age > ?1 and cust.age < ?2");
        myQuery.setParameter(1, lowAge);
        myQuery.setParameter(2, highAge);
        Iterator<Customer> iter = myQuery.getResultIterator();
        int sum = 0;
        while(iter.hasNext()) {
            sum += iter.next().age;
        }
        return new Integer(sum);
    } 

    public Object reduceResults(Collection results) {
        Iterator<Integer> iter = results.iterator();
        int sum = 0;
        while(iter.hasNext()) {
            sum += iter.next();
        }
        return new Integer(sum);
    }
}
The agent has three important parts. The first allows a specific set of entries to be processed without a query. It simply iterates over the set of entries adding the ages. The sum is returned from the method. The second uses a query to select the entries to be aggregated. It then sums all the matching Customer ages. The third method is used to aggregate the results from each partition to a single result. The grid performs the entry aggregation in parallel across the grid. Each partition produces an intermediate result that must be aggregated with other partition intermediate results. The third method performs that task.

Example 2-34 is an example code fragment showing how to invoke the agent. It shows a set of entries specified using keys being aggregated.

```
Example 2-34  Example code using Reduce agent
Session sesn = grid.getSession();
ObjectMap map = sesn.getMap("Customer");
AgentManager myAgentManager = map.getAgentManager();
SumAgeReduceAgent agent = new SumAgeReduceAgent();
Customer cust = new Customer();
cust.ssn = "123-45-6789";
ArrayList<Customer> list = new ArrayList<Customer>();
list.add(cust);
cust = new Customer();
cust.ssn = "123-45-6780";
list.add(cust);
Integer v = (Integer)myAgentManager.callReduceAgent(agent,list);
```
If the list is large then a query based agent is preferable. Example 2-35 is an example of invoking the query based version.

**Example 2-35  Example query based agent**

```java
Session sesn = grid.getSession();
    ObjectMap myObjectMap = sesn.getMap("Customer");
    AgentManager myAgentManager = myObjectMap.getAgentManager();
    SumAgeReduceAgent myAgent = new SumAgeReduceAgent();
    myAgent.lowAge = 10;
    myAgent.HighAge = 20;
    Integer v = (Integer)myAgentManager.callReduceAgent(myAgent);
```

### 2.2.9 eXtreme Scale system programming API

The grid provides several plug points to allow applications and cache providers to integrate with various data stores and alternative client APIs and to improve overall performance of the cache. The grid ships with several default, pre-built plug-ins, but you can also build custom plug-ins with the application.

eXtreme Scale plug-ins are concrete classes that implement one or more eXtreme Scale plug-in interfaces. These classes are then instantiated and invoked by the grid at appropriate times.

The eXtreme Scale system programming API provides the system programming model that you can use to develop your own plug-ins and plug them into eXtreme Scale cache to achieve additional functions.

eXtreme Scale instantiates plug-in instances and initializes them using JavaBeans conventions. All plug-in implementations have the following requirements:

- The plug-in class must be a top-level public class
- The plug-in class must provide a public, no-argument constructor
- The plug-in class must be available in the class path for both servers and clients (as appropriate)
- Attributes must be set using the JavaBean style property methods
- Plug-ins, unless specifically noted, are registered before the grid initializes and cannot be changed after the grid is initialized
The following plug-ins are available:

- **eXtreme Scale plug-ins**
  - TransactionCallback
    A TransactionCallback plug-in provides transaction life cycle events.
  - ObjectGridEventListener
    An ObjectGridEventListener provides life cycle events for the grid, shards, and transactions.
  - SubjectSource, MapAuthorization, SubjectValidation
    eXtreme Scale provides several security endpoints to allow for the integration of custom authentication mechanisms.

- **BackingMap plug-ins**
  - Evictor
    With this plug-in you can create a custom evictor for evicting cache entries.
  - Loader
    With this plug-in, a grid map can behave as a memory cache for data typically kept in a persistent store on either the same system or some other system.
  - ObjectTransformer
    With the ObjectTransformer plug-in, you can serialize, deserialize, and copy objects in the cache.
  - OptimisticCallback
    Use the OptimisticCallback plug-in to customize versioning and comparison operations of cache objects when using the optimistic lock strategy.
  - MapEventListener
    A MapEventListener plug-in provides callback notifications and significant cache state changes that occur for a BackingMap.
  - MapIndexPlugin
    Use the indexing feature, which is represented by the MapIndexPlugin plug-in, to build an index or several indices on a BackingMap map to support non-key data access.
The WebSphere eXtreme Scale mediator pattern

In this chapter we describe the WebSphere eXtreme Scale mediator pattern. We provide an overview of the pattern including the context and problems that are appropriate for WebSphere eXtreme Scale mediator pattern. We also discuss the solution and implementation model.

This chapter contains the following sections:

- The pattern
- Example
- Context
- Problem
- The solution
- Structure
- The dynamics
3.1 The pattern

The WebSphere eXtreme Scale mediator pattern is one of the architectural patterns for designing solutions that make use of the eXtreme Scale cache. As an architectural pattern, it promotes loose coupling between architectural components while maintaining unified interfaces. Large and distributed systems are generally fairly complex making it difficult and expensive to change. The WebSphere eXtreme Scale mediator pattern introduces the eXtreme Scale cache into distributed messaging and service-oriented applications without altering the interfaces and behavior to provide significant improvement in performance, availability, and scalability.

**WebSphere eXtreme Scale mediator pattern**

The WebSphere eXtreme Scale mediator pattern enables messaging and service-oriented applications to process massive volumes of data and attain high availability, scalability, and high performance.

3.2 Example

Consider a financial data application, typical of customer relationship management environments where customers have a number of options for contacting a financial services organization’s customer service. Customers can access their accounts directly from the Web and review their personal, account, or credit information. The customers can also contact the financial services organization’s customer support center by telephone.

Imagine then a scenario where a customer retrieved their financial data from the Web and identified some issues that they would rather resolve as quickly as possible. The customer picks up a phone and calls the financial services organization’s customer support center. In order for the customer support agent to attend to this customer, the agent must retrieve the customer’s financial data. If the agent finds it necessary to refer the customer to a supervisor or manager, the same customer data will be retrieved again. This is obviously not a preferred system design since there is cost associated with each access to the data from the database.
Figure 3-1   Example scenario

This application must maintain high performance levels while contending with rapid increases in traffic. Using a traditional clustering approach has not provided the high availability, scalability, and high performance required in this rapid growth scenario. WebSphere eXtreme Scale provides an alternative solution by supporting grid-based application patterns. Using a grid allows large quantities of data to be processed at in-memory speeds with little change in response time as the grid grows.

3.3 Context

A message-driven data processing application that receives and processes multiple service requests asynchronously.
3.4 Problem

The performance of message-driven data processing application in a distributed environment can be improved by processing asynchronous requests for data in parallel. When the data is distributed and cached the application must contend with managing the organization of the data, synchronization of the data, the availability and recovery in case of system failure. To support this computational model, some key requirements must be met. These include:

- Improved scalability: the cost of scaling the system should be predictable as new servers are added to system
- Reduce latency: typically the data is moved from the repository to where it is processed, the time for communicating the request and response should not exceed the time to process the data
- High performance: maximizing performance by servicing the request in parallel
- Transparency: The application code should be shielded from the complexity of the distributed caching environment

3.5 The solution

The ObjectGrid distributed coherent cache is able to scale the capacity and performance linearly as nodes are added to the cluster. The following are some of the key features eXtreme Scale cache:

- Partitioning
  Partitioning is used to scale out the application. It stores transactional data in exactly one partition and supports replication of a configurable number of copies throughout the grid. Instances of partitions are referred to as shards. A single Java Virtual Machine (JVM) might host many partitions, but never more than one shard from the same partition.

- High availability
  Advanced memory-to-memory replication capabilities provide high availability. Replication can be configured to be synchronous, asynchronous, or dynamically combine the two.

- Parallel processing
  Massive parallel processing is supported. Through use of agents, the processing logic is moved closer to the data and thereby avoiding expensive data transmission and slow persistence of storage devices.
- Self-organizing
  Partitions are deployed on the available JVMs in the grid using policies that you specify in the deployment policy file. You simply specify a handful of optional placement rules and ObjectGrid manages the placement of the thousands of partitions that can exist in the grid.

- Self-repairing
  As servers fail or are removed from the grid, ObjectGrid will try to maintain the replica count. It automatically recreates shards for the partitions on failed servers on surviving servers.

The application is split into the following two parts:

- Client and the messaging service: sends requests and gets responses asynchronously
- Data service: responds to asynchronous requests for data.

The mediator pattern encapsulates the interaction between a client application and messaging service, and between messaging service and the data server. The decoupling of the two parts of the application enable the introduction of the ObjectGrid between the client applications and the data server. A key benefit of this solution design is that the mediator pattern requires minimum implementation changes to the existing application.

Figure 3-2  WebSphere eXtreme Scale mediator pattern solution design
3.6 Structure

The WebSphere eXtreme Scale mediator pattern includes the following participants:

- **Client application**
  This is the component that initiates requests for data and retrieves the responses.

- **Client messaging service**
  The client application places requests for data on the Client messaging service and retrieves responses asynchronously from the messaging service. The Java Messaging Service (JMS) can be used to realize the client messaging service. The client application is unaware of the introduction of the grid and continues to run as though it is communicating with the data service application directly.

- **MediatorMDBBean**
  The MediatorMDBBean listens for requests from client application for data on the client messaging service. Incoming requests are inserted in the RequestMap map for processing.

- **MediatorAgent**
  MediatorAgent performs operations on specific entries in partitions in the grid. It is an ObjectGridEventListener, listening to the events in the grid.

- **MediatorLoader**
  A MediatorLoader is a plug-in for a grid BackingMap. Only one Loader can ever be associated with a given BackingMap. Each BackingMap has its own Loader instance.

- **Data messaging service**
  The data service application retrieves requests for data on the data messaging service and places responses asynchronously on the messaging service. The JMS can be used to realize the data messaging service. The data server application is unaware of the introduction of the ObjectGrid and continues to run as though it is communicating with the client application directly.

- **Data service application**
  This application serves as the repository for data.
3.7 The dynamics

The following actions occur in the WebSphere eXtreme Scale mediator pattern:
1. Client application places data requests in the client messaging service queue.
2. The MediatorMDBBean picks up the request message.
3. The MediatorMDBBean places the request in the RequestMap map.
4. The MediatorAgent is invoked to process the request.
5. The MediatorAgent processes the request.
6. The data is looked up in the BackingMap.
7. If the data is available in the BackingMap, a copy is made.
8. The MediatorAgent places a request for retrieval of the data from the data service application on the data message service queue if the data is not available in the BackingMap.
9. The data service application picks up the request.
10. The data service application places the data in the data messaging service queue.
11. The MediatorAgent retrieves the data and updates the BackingMap.
12. The data is placed in the client messaging service queue.
13. The client application retrieves the data from the client messaging service queue.
Figure 3-4  WebSphere eXtreme Scale mediator pattern dynamics
The base application

In this chapter, we describe the base sample financial data application scenario that demonstrates retrieval of customer financial data given a customer ID.

This chapter contains the following sections:

- The base application implementation
- Setting up the base application environment
- Testing the base application
4.1 The base application implementation

The user interface is hosted on an application server called FinanceServer. The application consists of a Servlet and Java Server Pages. The application server is a standalone server. The requests for customer data are processed by the Servlet, which places the data in a messaging queue. Listening on the queue is FinanceDataServer, the processing application which is hosted on another standalone application server. The server application consists of a Message Driven Bean, which listens for and processes requests for customer financial data. The requests are processed and passed back to the Servlet, which has a timeout set for each request.

The user interface application is packaged in ITSOFinance.ear which is hosted in standalone application server node ITSOFinanceServer. The Application processing data is part of the ITSOFinanceData.ear; it is deployed on ITSOFinanceDataServer.

4.1.1 ITSOFinance application

The ITSOFinance Application.ear hosts the user interface. The ear file consists of:

- FinanceInfo.jsp
- FinanceServlet.java
FinanceInfo JSP

Customer data is requested using the user interface application. The key used for querying customer data is customerID. The user interface consists of FinanceInfo.jsp which provides the interface for querying. Two buttons on the FinanceInfo page provides the following functionalities:

- **Get Info**
  
  Processes and fetches the customer data.

- **Help**
  
  Provides information about the FinanceInfo application.
Figure 4-2  FinanceInfo JSP™ query page

After entering the ID in the Customer ID field, click the GetInfo button to fetch the customer data. The retrieved customer data is displayed in the different sections as Personal Information, Credit Information, and Bank Information.
// Get attributes returned from Finance Servlet
String appVersion = (java.lang.String)request.getAttribute("appVersion");
if (appVersion == null) {
    appVersion = "unknown";
}

// Customer info
String cID = (java.lang.String)request.getAttribute("cID");
if (cID == null) {
    cID = "";
}
String cName = (java.lang.String)request.getAttribute("cName");
if (cName == null) {
    cName = "";
}
String message = (java.lang.String)request.getAttribute("message");

// Personal info
String pTaxID = (java.lang.String)request.getAttribute("pTaxID");
if (pTaxID == null) {
    pTaxID = "";
}
String pEMail = (java.lang.String)request.getAttribute("pEMail");
if (pEmail == null) {
    pEMail = "";
}

// Bank info
String bAccount = (java.lang.String)request.getAttribute("bAccount");
if (bAccount == null) {
    bAccount = "";
}
String bBalance = (java.lang.String)request.getAttribute("bBalance");
if (bBalance == null) {
    bBalance = "";
}

// Credit info
String cScore = (java.lang.String)request.getAttribute("cScore");
if (cScore == null) {
    cScore = "";
<FORM NAME="bankForm" action="/Finance/FinanceServlet" method="post">
<INPUT TYPE="HIDDEN" NAME="operation" VALUE="None"/>

<H2>Customer Name</H2>
<B>Customer ID</B> <INPUT TYPE="TEXT" NAME="cID" Size="30" MAXLENGTH="30" VALUE="<%=cID%>">
<INPUT TYPE=submit NAME="submitGetCustomerInfo" value="Get Info" id="oper"
onClick='bankForm.operation.value="getCustomerInfo"'/>
<INPUT TYPE=submit NAME="Help" value="Help" id="oper"
onClick='bankForm.operation.value="help"'/>

<BR><BR>
<B>Customer Name : </B> <%= cName %>
<% if (message != null) { %>
<P><B> <%= message %> </B></P>
<% } %>
<HR>

<H2>Personal Information</H2>
<B>Tax ID : </B> <%= pTaxID %>
<BR><BR>
<B>eMail : </B> <%= pEMail %>
<BR><BR>
<HR>

<H2>Bank Information</H2>
<B>Account : </B> <%= bAccount %>
<BR><BR>
<B>Balance : </B> <%= bBalance %>
<BR><BR>
<HR>

<H2>Credit Information</H2>
<B>Credit Score : </B> <%= cScore %>
<BR><BR>
FinanceServlet
FinanceServlet processes the request for customer data. The FinanceServlet init method initiates the Java Messaging Service (JMS) resources needed for the Servlet. This includes:

- JMS connection factory
- JMS destination queue
- JMS connection

*Example 4-3  FinanceServlet init*

```java
public void init() throws ServletException {
    System.out.println("FinanceServlet.init()");
    super.init();
    // Setup connection for Point to Point Queue
    try {
        initialContext = new InitialContext();
        conFactory = (ConnectionFactory) initialContext.lookup(jndiConnectionFactory);
        dest = (Destination) initialContext.lookup(jndiDestinationQueue);
        con = conFactory.createConnection();
    } catch (Exception e) {
        System.out.println("FinanceServlet.init() - Exception - " + e.getMessage());
        e.printStackTrace();
    }
}
```

Two types of requests are passed to FinanceServlet for processing. The type of request is determined by the “operation” parameter. The two types are:

- help
  Processes the information to be displayed on how to use the application.

- getCustomerInfo
  Processes the request for customer data by retrieving the customer ID for which data is requested and then retrieving the detail customer data. The following steps process the results:
  a. Get CustomerID.
  b. Create JMS session.
  c. Create temporary queue.
d. Create a message consumer for the temporary queue which will be listening for response on this temporary queue.

e. Create a message producer to send the request as JMS message.

f. Create a MapMessage and place the request.

\textbf{Note:} We use MapMessage to represent data as key value pair.

g. Set the JMSType as getCustomerInfo and specify the customer ID for querying.

h. Set the JMSReplyTo temporary queue.

i. Send the message.

j. Wait for the response message for about 15000 milliseconds.

\textbf{Note:} If the response message is received the customer data is sent back. If the response is not received in the time frame, a server response message is sent back.

\begin{quote}
\textbf{Example 4-4} \hspace{1em} FinanceServlet: getCustomerInfo
\end{quote}

```
try {
    // create connection
    session = con.createSession(false, Session.AUTO_ACKNOWLEDGE);
    Queue tempReplyQueue = session.createTemporaryQueue();
    msgConsumer = session.createConsumer(tempReplyQueue);
    con.start();
    msgProducer = session.createProducer(dest);

    // create outbound message
    MapMessage outMessage = session.createMapMessage();
    outMessage.setString("cID", cID);
    outMessage.setJMSType("getCustomerInfo");
    outMessage.setJMSReplyTo(tempReplyQueue);
    msgProducer.setDeliveryMode(DeliveryMode.NON_PERSISTENT);

    // send message
    // System.out.println("FinanceServlet.getCustomerInfo() -
```
// Sending outMessage - " + outMessage.toString());
message = "<BR>" + timeStamp(request) + "Request message sent."
msgProducer.send(outMessage);

// wait for response and process it
Message inMessage = msgConsumer.receive(consumerWait);
if (inMessage instanceof MapMessage) {
    message = message + "<BR>" + timeStamp(request) + "Response message received."
    // System.out.println("FinanceServlet.getCustomerInfo() -
    // Receiving inMessage - " + ((MapMessage)
    // inMessage).getString("cName");
    request.setAttribute("cName", ((MapMessage)inMessage).getString("cName"));
    request.setAttribute("pTaxID", ((MapMessage) inMessage).getString("pTaxID"));
    request.setAttribute("pEMail", ((MapMessage) inMessage).getString("pEMail"));
    request.setAttribute("bAccount", ((MapMessage) inMessage).getString("bAccount"));
    request.setAttribute("bBalance", ((MapMessage) inMessage).getString("bBalance"));
    request.setAttribute("cScore", ((MapMessage) inMessage).getString("cScore"));
    String source = ((MapMessage) inMessage).getString("source");
    message = (source != null) ? message + "<BR>" + source :
    message;
} else {
    message = message + "<BR>" + timeStamp(request) + " - No response message received.";
    System.out.println("FinanceServlet.getCustomerInfo() - No response after " +
    consumerWait + " ms");
    message = "<FONT COLOR=RED> No response getting Customer Information for Customer ID " + cID + " after " + consumerWait + "ms.</FONT>";
}
}
catch(Exception e){
}

4.1.2 ITSOFinanceData application

ITSOFinanceData.ear contains the finance data application which processes the request and sends back the response. The package contains the FinanceData Message Driven Bean (MDB) which listens on the messaging queue for requests to process.

FinanceData message driven bean
The FinanceData MDB is a message listener. It is configured to listen on the messaging queue specified in the configuration for the MDB. It implements the
onMessage method which is invoked based on messages posted on the queue. The JMS configuration is initialized in the MDB init Method.

Example 4-5  JMS configuration in init()

```java
try{
    initialContext = new InitialContext();
    conFactory = (ConnectionFactory) initialContext.lookup(jndiConnectionFactory);
    dest = (Destination) initialContext.lookup(jndiDestinationQueue);
    con = conFactory.createConnection();
}
```

Process getCustomerInfo

The onMessage method checks the JMSType of the incoming message. If the request is getCustomerInfo, then the following steps are followed:

1. Obtain the customer ID from the incoming request.
2. Lookup the requested customer ID in the hash map.

   **Note:** The test data is maintained in a hash map.

3. Retrieve the data and create a Map Message that includes the data.
4. Send the message.

Example 4-6  Finance Data MDB process Get Message

```java
try {
    // get connection
    replyDest = inMessage.getJMSReplyTo();
    session = con.createSession(false, Session.AUTO_ACKNOWLEDGE);
    producer = session.createProducer(replyDest);
    // build reply message
    replyMsg = session.createMapMessage();
    String key = ((MapMessage) inMessage).getString("cID");
    if (backEndHashMap.containsKey(key)) {
        String [] data = backEndHashMap.get(key);
        replyMsg.setString("cID", key);
        replyMsg.setString("cName", data[0]);
        replyMsg.setString("pTaxID", data[1]);
        replyMsg.setString("pEMail", data[2]);
        replyMsg.setString("bAccount", data[3]);
        replyMsg.setString("bBalance", data[4]);
        replyMsg.setString("cScore", data[5]);
        replyMsg.setString("versionID", data[6]);
```
replyMsg.setJMSType("getCustomerInfo");
}
else {
    System.out.println("FinanceDataMDBBean.processGetMessage() - Data " + key + " not in map");
    return;
}
    producer.send(replyMsg);
}
catch(Exception e){
    }

---

**Process getVersionInfo**

If the request is for getVersionInfo, the processVersionMessage method is invoked, and the following steps are performed:

1. Retrieve the versionID and customerID in the request.
2. Check that the customer ID is in the hash map.
3. Verify the version of the data, if it is the same then set the status of the property isInSync to Y otherwise set it to N.
4. Place the data in the MapMessage.
5. Send the message.

**Example 4-7 FinanceData MDB: getVersionInfo method**

```java
try {
    // get connection
    replyDest = inMessage.getJMSReplyTo();
    session = con.createSession(false, javax.jms.Session.AUTO_ACKNOWLEDGE);
    producer = session.createProducer(replyDest);

    // build reply message
    replyMsg = session.createMapMessage();
    String key = ((MapMessage) inMessage).getString("cID");
    String messageVID = ((MapMessage) inMessage).getString("versionID");
    if (backEndHashMap.containsKey(key)) {
        String [] data = backEndHashMap.get(key);

        // Check if customerID in 30s and randomize version
        if (Integer.parseInt(key) > 29 ) {data[6] = String.valueOf(1 + (int)(Math.random() * 4));}
        backEndHashMap.put(key, data);
    }
```
// check if data versions match
String isInSync = (messageVID.equals(data[6])) ? "Y" : "N";
replyMsg.setString("isInSync", isInSync);
replyMsg.setString("versionID", data[6]);
}
else {
    System.out.println("FinanceDataMDBBean.processVersionMessage() - Data " + key + " not in map");
    return;
}
producer.send(replyMsg);
}

Note: If the customer ID is greater than or equal to 30, then a random version change code has been added. This is used to simulate the functionality of another application updating the customer data.

FinanceData MDB configuration
The MDB gets the queue information configured from the JMS activation specification defined for the messaging queue. This is specified in the Integrated Development Environment (IDE) when the EAR is generated. So after creating the enterprise bean details, the FinanceData MDB is linked to the activation specification.

Note: These changes are pre-configured in the example EAR files.

Configure activation specification
The activation specification can be linked with the following procedure:

1. Click FinanceDataMDBBean in the deployment descriptor editor to open the deployment descriptor.
2. Click JCA Adapter on the MDB Bean window under WebSphere Bindings.
3. Enter jms/ITSOActSpec in the ActivationSpec JNDI name field.
Figure 4-3  ITSOFinanceDataMDB Activation Specification configuration

Note: You can also configure or verify the information from the binding file. In META-INF\ibm-ejb-jar-bnd.xmi, the <ejbbindings> tag has the property for activationSpecJndiName set to jms/ITSOActSpec.

Example 4-8  FinanceData MDB bindings file: ibm-ejb-jar-bnd.xmi

```xml
<?xml version="1.0" encoding="UTF-8"?>
<ejb:bean xmi:id="EJBJarBinding_1182979368687"
 xmlns:xmi="http://www.omg.org/XMI" xmlns:ejb="ejb.xmi"
 xmlns:ejbbind="ejbbind.xmi" xmlns:id="EJBJarBinding_1182979368687"
>
<ejbJar href="META-INF/ejb-jar.xml#ejb-jar_ID"/>
<ejbBindings xmi:type="ejbbind:MessageDrivenBeanBinding"
 xmi:id="MessageDrivenBeanBinding_1182979368687"
 activationSpecJndiName="jms/ITSOActSpec">
<enterpriseBean xmi:type="ejb:MessageDriven"
 href="META-INF/ejb-jar.xml#MessageDriven_1182978732515"/>
</ejbBindings>
<ejb:bean>
**Disable the MDB transaction**

The transaction for the Finance Data MDB should be set to **not supported**. This is to ensure that the messages are sent and received without the consumer timing out.

*Example 4-9  Deployment Descriptor for FinanceData Source- assembly descriptor section*

```xml
<assembly-descriptor>
  <container-transaction>
    <method>
      <ejb-name>FinanceDataMDBBean</ejb-name>
      <method-name>*</method-name>
    </method>
    <method>
      <ejb-name>FinanceDataMDBBean</ejb-name>
      <method-name>onMessage</method-name>
      <method-params>
        <method-param>javax.jms.Message</method-param>
      </method-params>
    </method>
    <method>
      <ejb-name>FinanceDataMDBBean</ejb-name>
      <method-name>onMessage</method-name>
      <method-params>
        <method-param>javax.jms.Message</method-param>
      </method-params>
    </method>
    <trans-attribute>NotSupported</trans-attribute>
  </container-transaction>
</assembly-descriptor>
```
4.2 Setting up the base application environment

The sample scenario simulates a financial application where customer financial data is retrieved based on the customer identification. (ID)

The application scenario is installed in two phases. In this first phase, the base application environment is installed. It installs the following parts:

- Web application ITSOFinance
- Data application ITSOFinanceData
- JMS resources necessary to enable the two applications to communicate

This setup simulates a client application communicating directly with the database.

![Diagram showing the base application environment]

Figure 4-4  Base application environment

The base application environment set-up starts with the configuration of application servers. This is followed by the messaging services set-up and the installation and running of ITSOFinance and ITSOFinanceData applications.
4.2.1 Configure the application servers

For the finance data sample application scenario, we configure two application servers, one server is for the ITSOFinance application, and the second server is for ITSOFinanceData, the data manager.

**Note**: Application servers should be created with the Extended Deployment template. If any servers with a template other than this are created during installation, delete them and create new servers.
From the administrative console, perform the following steps:

1. Click **Servers → Application servers** to get to the “Application servers” page shown in Figure 4-6.
2. Click the **New** button to create a new application server. See Figure 4-7.

![Create New Application Server page](image)

**Figure 4-7** Create New Application Server page

3. On the “Create New Application Server” page:
   a. Select **appnode1node01** as the node where the application server will run.
   b. Enter **ITSOFinanceServer** in the Server name field for the name of the server.
   c. Click **Next** to go to the “Select a server template” page.
4. On the “Select a server template” page, shown in Figure 4-8:
   a. Select the defaultXD template.
   
   **Note:** The defaultXD template is the default template for WebSphere Extended Deployment. Use this template when you are creating a dynamic cluster on distributed platforms.

   b. Click Next to access the “Specify server specific properties” page.

5. Accept the default settings and click Next.

6. Click Finish to confirm the new server set-up.

**Setting up ITSOFinanceDataServer:** Repeat steps 1 to 6 to create the ITSOFinanceDataServer. In step 3, select appnode4node01 as the node to run ITSOFinanceDataServer, and enter ITSOFinanceDataServer as the server name.

7. Save the changes so they will be reflected on the master configuration and synchronize nodes.
4.2.2 Configure JMS messaging service

The next step in the set-up process is the configuration of the JMS resources needed to allow ITSOFinance and ITSOFinanceData applications to communicate.

The following JMS resources need to be configured:

- Service integration bus
- Connection factory
- Messaging queue
- Activation specification

The procedure detailed in “Configure the service integration bus” on page 95 configures each of these JMS resources.
Configure the service integration bus
From the administrative consooe, the following steps will configure the service integration bus:

1. Click **Service integration → Buses** to access the Buses page.

![Buses page](image)

**Figure 4-10 Buses page**

2. Click **New** to create a new Service Integration Bus.

3. Create a new messaging bus. On the “Create a new messaging bus” page, shown in Figure 4-11 on page 96, perform the following steps:
   a. Enter **ITSOBUS** for the bus name.
   b. Accept the unchecked default setting of Bus security.
   c. Click **Next**.
4. Click **Finish** on the confirmation window to save the new bus configuration.

5. Add a new bus member.
   a. Click **ITOSBUS** link.
   b. Click **Configuration → Topology → Bus members**, as shown in Figure 4-12 on page 97.
c. Click **Add** on the “New bus members” page to create a new bus member. See Figure 4-13 on page 98.
d. Select **ITSOFinanceServer** as the server on the “Add a new bus member” page as shown in Figure 4-14 on page 99.

e. Click **Next**.
Figure 4-14  Add a server as a new member of the bus

6. Click **Next** to accept default the settings on subsequent pages and proceed to the next page.

7. Click **Finish** to complete creation of the new bus member.

**Set-up ITSOFinanceDataServer as a bus member:** Repeat steps 5 to 7 to add ITSOFinanceDataServer as a bus member.

8. Save the changes to the master configuration.

9. Create the destinations on ITSOBUS.
   a. Click **Buses** followed by **ITSOBUS** from the Buses page.
   b. Click **Destinations** under Destinations resources as shown in Figure 4-15 on page 100.
c. Click **New** on the Destinations page, as shown in Figure 4-16 on page 101, to create a new Destination.
Figure 4-16 Destinations page

d. Select **Queue** as the destination type on the “Create new destination” page, as shown in Figure 4-17 on page 102.

e. Click **Next**.
f. Enter **ITSOQDest** as the name of the queue to set queue attributes.

g. Click **Next**.

h. Select **ITSOFinanceDataServer** as in Figure 4-18 on page 103 to assign the queue to a bus member.

i. Click **Next**.
Configure the connection factory

JMS connection factory is used to create connections to the associated provider of JMS destinations. This step creates a connection factory for the default messaging provider.

From the administrative console, perform the following procedure:

1. Click Resources ➔ JMS ➔ JMS providers.
2. Verify that the scope is set to cell level. In this base application example it is cell=dmgrnodeCell01. (See Figure 4-19 on page 104.)

   Note: Resource creation is performed at cell level to enable visibility for all nodes.

3. Click Default messaging provider, as shown in Figure 4-19 on page 104.
4. Click **Additional Properties → Connection Factories.**
5. Click **New** to create a new Connection factory on the Default “messaging provider Connection factories” page.

6. Configure the Connection factory as shown in Figure 4-21 on page 106.
   a. Enter **ITSOConFactory** in the Name field.
   b. Enter **jms/ITSOConFactory** in the JNDI name field.
   c. Select **ITSOBUS** as the Bus Name.

7. Click **Apply** to apply the configuration.

8. Click **Save** to save the configuration.
Configure the queue

A messaging queue is required for sending and receiving messages using ITSOBUS, the Service Integration Bus that was created in “Configure the service integration bus” on page 95. In this step we create and configure the messaging queue.
From the administrative console, perform the following procedure:

1. Click Resources → JMS → JMS providers.

2. Verify that the scope is set to cell level, in this base application example it is cell=dmgrnodeCell01. (See Figure 4-22.)

3. Select Default messaging provider, as in Figure 4-22.

5. Create a new messaging queue by clicking **New** on the new Queue page.

6. Configure the Queue as shown in Figure 4-24 on page 109.
   a. Enter **ITSOFinanceQ** for the Queue name.
   b. Enter **jms/ITSOFinanceQ** for the queue JNDI name.
   c. Enter the Connection information.
      i. Select **ITSOBUS** as the Bus name.
      ii. Select **ITSOQDest** as the queue name for bus destination.

7. Click **Apply** and **Save** to save the queue configuration.
Configure the activation specification

JMS activation specification configures resources that enable MDBs to communicate with the default messaging provider.

From the administrative console, perform the following steps to configure a JMS activation specification:

1. Click **Resources → JMS → JMS providers**.

2. Verify that the scope is set to cell level. In this base application example, it is **cell=dmgrnodeCell01**. (See Figure 4-25 on page 110.)

3. Click **Default messaging provider** as in Figure 4-25 on page 110.
4. Click **Additional Properties → Activation specifications** on the Default “messaging provider” page, as shown in Figure 4-26 on page 111, to create a new Activation specification.
5. Click New to create a new activation specification.

6. Configure the Activation specification as shown in Figure 4-27 on page 112.
   a. Enter ITSOActSpec for the name.
   b. Enter jms/ITSOActSpec for the JNDI name.
   c. Enter the destination information.
      i. Select Queue as the Destination type.
      ii. Enter jms/ITSOFinanceQ as the Destination JNDI name.
      iii. Select ITSOBUS as the Bus name.

7. Click Apply and Save to save the activation specification.
Figure 4-27  Configure activation specification
4.2.3 Install ITSOFinance and ITSOFinanceData applications

The ITSOFinanceServer and ITSOFinanceDataServer are packaged as ITSOFinance.ear and ITSOFinanceData.ear respectively. Here are the steps for installing both applications.

**Note:** Before you start the installation process, place the ear files in a folder on the local drive, for example in c:\apps.

From the Integrated Services Console perform the following procedure:

1. Select **Applications → Enterprise Applications → Install.**
2. Specify the path to the local folder containing the ear file and click **Next.**
3. Select installation options.
   a. Accept the default settings on the “Select installation options” page.
   b. Click **Next.**
4. Map modules to servers. See Figure 4-28 on page 114.
   a. Click the **Select** check box to select Finance module.
   b. Click the **Clusters and Servers** dropdown list and select **ITSOFinanceServer** as the server to host the application.
   c. Click **Apply** next to the Clusters and Servers list.
   d. Verify that the server name associated with the Finance module is ITSOFinanceServer. The check box will become unchecked.
   e. Click **Next.**
5. On the Summary page:
   a. Review the summary.
   b. Click **Finish** after reviewing and making any changes.
   c. Click **Save** to save the installation options.

   **Note:** Repeat steps 1 to 5 in order to specify the installation options for ITSOFinanceData.ear and map it to ITSOFinanceDataServer. See Figure 4-29 on page 115.
6. Verify the application installation.
   a. Select **Applications** → **All applications** from the Integrated Services Console.
   b. Verify that the ITSOFinance and ITSOFinanceData applications are listed in all applications supported and managed by WebSphere Extended Deployment.
4.3 Testing the base application

Follow these steps to run the installed ITSOFinance and ITSOFinanceData applications:

1. Start the application servers using the following steps:
   a. Select Servers → Application servers from the Integrated Services Console.
   b. Check the boxes to the left of ITSOFinanceServer and ITSOFinanceDataServer.
   c. Click Start.

The status should change to green to indicate that the server has started, as shown in Figure 4-31 on page 117.
2. Test the application using the following steps.

   a. Enter the URL for the application. For example:

      http://<Your ODR node machine name>/Finance/FinanceServlet.

      You will get a Finance Application query page as in Figure 4-32 on page 118.
b. Enter numbers from 0 to 39 for the Customer ID.

c. Click **Get Info**.

This will fetch the personal and banking information for the Customer ID entered. The retrieved data is displayed on the page as in Figure 4-33 on page 119.

**Note:** This verifies that the ITSOFinance and ITSOFinanceData applications are communicating successfully using JMS messaging resources.
Finance Application - Version 1.0.0

Customer Name

Customer ID: 2
Customer Name: Billy

July 10, 2007 4:56:54 PM EDT -- Request message sent.
July 10, 2007 4:56:55 PM EDT -- Response message received.

Personal Information

Fax ID: 2-2-1002
Email: Billy@Billy2.com

Bank Information

Figure 4-33 The customer’s retrieved personal and banking data
The eXtreme Scale mediator application

In this chapter, we describe the introduction of WebSphere eXtreme Scale to the base sample financial data application. It demonstrates an implementation of the WebSphere eXtreme Scale mediator pattern.

This chapter contains the following sections:

- The mediator application implementation
- Setting up the mediator environment
- Testing the mediation application
5.1 The mediator application implementation

Initially we implemented and installed the base application which included the ITSOFinance user interface application and ITSOFinanceData application which included the MDB to process the messages. In this phase of the sample application implementation, we implement the mediator pattern which introduces the grid between ITSOFinance and ITSOFinanceData applications.

The Mediator application is hosted on the ITSOMediatorServer and is packaged as ITSOMediatorApp.ear. In this section, we explain the implementation of the ITSOMediator application.

5.1.1 ITSOMediator application

The ITSOMediatorApp.ear package includes the following:

- MediatorMDBBean
  MediatorMDBBean replaces the FinanceData MDB as the listener on the messaging queue for customer data requests.

- MediatorLoader
  MediatorLoader implements the grid loader interface. It preloads data in ObjectGrid during grid initialization.

- MediatorAgent
  A listener agent attached to the grid and listening for events.

- Configuration files
  The files ObjectGrid.xml and ObjectGridDeployment.xml are used for configuring ObjectGrid.
5.1.2 MediatorMDBBean

The MediatorMDBBean listens for requests from FinanceServlet for customer financial data on the messaging queue. The requests are inserted in the RequestMap map in FinanceGrid for processing.

The MediatorMDBBean ejbCreate method initializes ITSOFinanceGrid and performs the following to connect to the grid:

- Get the grid manager from ObjectGridConnectionFactory.
- Get cluster hosting the grid.

Note: MediatorMDBBean and the grid are co-located. The grid is running locally so nulls are passed as the arguments to connect to the grid.

- Connect to the grid by passing cluster information and the grid name ITSOFinanceGrid.
- Initiate a session.
- Get the ObjectMap for the requestMap from the session.
Example 5-1  MediatorMDBBean ejbCreate method

```java
public void ejbCreate() {
    System.out.println("MediatorMDBBean.ejbCreate()");
    try {
        initializeWithColocatedOG();
        gridSession = og.getSession();
        _requestMap = gridSession.getMap(requestMapName);
    } catch (ConnectException e) {
        System.out.println("MediatorMDBBean.ejbCreate() - Connect exception - " +
                          e.getMessage() );
        e.printStackTrace();
    } catch (TransactionCallbackException e) {
        System.out.println("MediatorMDBBean.ejbCreate() - TransactionCallback exception - " + e.getMessage() );
        e.printStackTrace();
    } catch (ObjectGridException e) {
        System.out.println("MediatorMDBBean.ejbCreate() - ObjectGrid exception - " + e.getMessage() );
        e.printStackTrace();
    } catch (Throwable t) {
        System.out.println("MediatorMDBBean.ejbCreate() - Throwable - " + t.getMessage() );
        t.printStackTrace();
    }
}
```
Example 5-2  MediatorMDBBean initializeWithColocatedOG method

```java
private static synchronized void initializeWithColocatedOG() throws Throwable {
    System.out.println("MediatorMDBBean.initializeWithColocatedOG() - isInitialized = " + isInitialized);
    if (!isInitialized) {
        try {
            ogm = ObjectGridManagerFactory.getObjectGridManager();
            cluster1 = ogm.connect(null, null);
            og = ogm.getObjectGrid(cluster1, gridName);
            isInitialized = true;
        }
        catch (Throwable t) {
            t.printStackTrace();
            throw t;
        }
    }
    catch (Throwable t) {
        t.printStackTrace();
        throw t;
    }
}
```

Incoming requests on the messaging queue trigger messaging events. The MediatorMDBBean onMessage method is invoked to handle messaging events. Each event is processed and the request is placed in RequestMap.

- Check for the incoming request as MapMessage type.
- Get the customerID from the request.
- Begin a transaction and insert the request in RequestMap and commit the transaction.

Example 5-3  MediatorMDBBean onMessage method

```java
try {
    if (msg instanceof MapMessage) {
        MapMessage m1 = (MapMessage)msg;
        String customerID = m1.getString("cID");
        RequestKey requestKey = new RequestKey(customerID, processId);
        gridSession.begin();
        _requestMap.insert(requestKey, m1);
        gridSession.commit();
    }
    else {
        System.out.println("MediatorMDBBean.onMessage() - Message format not recognized Use Map Message");
    }
}
```
MediatorMDBBean configuration
The MediatorMDBBean is configured to listen to the messaging queue for requests from FinanceServlet using the activation specification configuration.

Note: The example EAR’s are built-in with the configurations and you do not have to make these changes. If the applications are developed and configured by yourself, ensure these configurations are present in the EAR deployed to the server.

Configure the activation specification
The activation specification can be linked in Rational® Application Developer.
1. Open deployment descriptor with deployment descriptor editor.
2. Click MediatorMDB.

Figure 5-2   Set ActivationSpec details for MediatorMDB
3. On the MDB Bean window under WebSphere Bindings.
   a. Click **JCA Adapter**.
   b. Enter **jms/ITSMediatorActSpec** in the ActivationSpec JNDI name field.
   c. Save the changes.

*Figure 5-3  Configure activation specification for MediatorMDB*
Disable the MDB transaction

Transactions should be disabled for MediatorMDB. This is to ensure that the messages are sent and received without having it time-out. To do this in Rational Application Developer follow these steps:

1. Click the deployment descriptor for the MediatorMDB to load the descriptor.
2. Click the Assembly tab to select it.
3. Click Add under Container Transactions (see Figure 5-4 on page 129) on the Assembly Descriptor page.

Note: You can also configure or verify the information from the binding file. In META-INF\ibm-ejb-jar-bnd.xmi, the <ejbbindings> tag has a property to set the activationSpecJndiName set to jms/ITSOMediatorActSpec.

Example 5-4  ibm-ejb-jar-bnd.xmi: MediatorMDB bindings file

```xml
<?xml version="1.0" encoding="UTF-8"?>
<ejbbnd:EJBJarBinding xmi:version="2.0"
xmns:xmi="http://www.omg.org/XMI" xmlns:ejb="ejb.xmi"
xmns ejbbnd="ejbbnd.xmi" xmi:id="EJBJarBinding_1182873054687">
  <ejbJar href="META-INF/ejb-jar.xml#ejb-jar_ID"/>
  <ejbbindings xmi:type="ejbbnd:MessageDrivenBeanBinding"
xmi:id="MessageDrivenBeanBinding_1182873054687"
activationSpecJndiName="jms/ITSOMediatorActSpec">
    <enterpriseBean xmi:type="ejb:MessageDriven"
href="META-INF/ejb-jar.xml#MediatorMDB"/>
  </ejbbindings>
</ejbbnd:EJBJarBinding>
```
4. Select the **MediatorMDB** check box. (See Figure 5-5 on page 130.)

5. Click **Next**.
Figure 5-5   Disable transaction for MediatorMDB - Container transaction
6. On the Container Transaction Type and Method Elements page. (See Figure 5-6.)
   a. Select **NotSupported** for the Container transaction type field.
   b. Click the checkboxes to select all methods in MediatorMDB.
   c. Click **Finish**.

![Figure 5-6](image)

You can verify the changes from the assembly descriptor section of the deployment descriptor for MediatorMDBBean by clicking the source. Alternatively it can be added to the deployment descriptor XML file as in Example 5-5 on page 132.
Example 5-5  ejb-jar.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar id="ejb-jar_ID" version="2.1"
xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd">
  <display-name>
    ITSOMediator</display-name>
  <enterprise-beans>
    <message-driven id="MediatorMDB">
      <ejb-name>MediatorMDB</ejb-name>
      <ejb-class>com.ibm.itso.finance.mediator.MediatorMDBBean</ejb-class>
      <messaging-type>javax.jms.MessageListener</messaging-type>
      <transaction-type>Container</transaction-type>
    </message-driven>
  </enterprise-beans>
  <assembly-descriptor>
    <container-transaction>
      <method>
        <ejb-name>MediatorMDB</ejb-name>
        <method-name>*</method-name>
      </method>
      <method>
        <ejb-name>MediatorMDB</ejb-name>
        <method-name>onMessage</method-name>
        <method-params>
          <method-param>javax.jms.Message</method-param>
        </method-params>
      </method>
      <method>
        <ejb-name>MediatorMDB</ejb-name>
        <method-name>onMessage</method-name>
        <method-params>
          <method-param>javax.jms.Message</method-param>
        </method-params>
      </method>
    </container-transaction>
  </assembly-descriptor>
</ejb-jar>
```
MediatorAgent

MediatorAgent is the ObjectGridEventListener, listening to the events in the grid. It also implements the following interfaces:

- **com.ibm.websphere.asynchbeans.Work**
  This is the asynchronous beans work interface. The WorkManager in WebSphere will control the work threads running inside the application server.

- **com.ibm.websphere.objectgrid.plugins.ObjectGridEventListener**
  This is the grid event listener interface. Significant events in the grid are communicated to the listener.

  This is a listener attached specifically to the shard that is activated, when a shard becomes a primary. It implements shardActivated and shardDeActivated methods. The catalog service is responsible for placing shards. Each ObjectGrid has a number of partitions. Each partition has a primary shard and an optional set of replica shards.

Example 5-6  ShardEvents

```java
public void shardActivated(ObjectGrid objectgrid);
public void shardDeactivate(ObjectGrid arg0);
```

Shard activated

MediatorAgent is activated when the shard becomes the primary shard. This results in shardActivated method being called with the grid as the argument. The MediatorAgent and its resources are initialized in shardActivated method.

MediatorAgent starts an asynchronous work thread. The thread checks the request map at regular intervals. This request map is part of the shard to which the listener is attached.

**Note:** Based on the number of partitions that are started, shards are created. Each shard will have a shard listener listening for its events. So there are as many MediatorAgents to process the requests as there are partitions. This enhances the scalability of the design.

MediatorAgent is activated when the shardActivated event is received. It executes the following steps to initializes required resources:

1. Get ITSOFinanceGrid session object.
2. Get RequestMap and FinanceMap ObjectMaps.
3. Initialize client and server JMS resources.
4. Initialize the worker manager to start controlling the worker thread.

5. Get the partitionID for the partition the listener is attached.
   
The partitionID is passed back in the response to FinanceServlet indicating
   the partition that processed the request for customer data.

Example 5-7  MediatorAgent shardActivated Event - initialize method

```java
this.itsoFinanceGrid = objectgrid;
this.mediatorSession = objectgrid.getSession();
this.requestMap = mediatorSession.getMap("RequestMap");
this.financeMap = mediatorSession.getMap("FinanceMap");
initServerJMS();
initClientJMS();
initWorkManager();
this.partitionID = (itsoFinanceGrid.getMap(financeMap.getName()).getPartitionId());
```

6. Start the processing worker thread.

Example 5-8  MediatorAgent shardActivateEvent - start ListenerThread

```java
workManager.startWork(this);
```

**Note:** WorkManager and asynchronous bean work thread was used
because the application is targeted to run in IBM WebSphere application server

7. Start the worker thread. It checks periodically for requests received by
   MediatorMDBBean from FinanceServlet and posted in RequestMap.
   
The thread picks up incoming requests for this partition from RequestMap.

8. The request is removed from RequestMap for processing.

Example 5-9  MediatorAgent: worker thread run method

```java
try {
    while (isListenerRunning()) {
        MapMessage message2Publish = null;
        mediatorSession.begin();

        // start processing request
        Object nextAvailableKey =
            requestMap.getNextKey(Long.parseLong(requestThreadWait));
        if (nextAvailableKey!=null) {
            MapMessage request = (MapMessage)requestMap.remove(nextAvailableKey);
```
// process the request ...
message2Publish = processInRequest(request);
}
mediatorSession.commit();

// Need to publish to grid outside of transaction used to
// process request because
// transactions can not be nested.
if (message2Publish != null) {
    publishToGrid(message2Publish.getString("cID"), message2Publish);
}
}

The shardDeActivated event

The shardDeActivated event is triggered when a shard is no longer the primary
shard. In handling this event, the request processing done by the worker thread
that was initiated in the run() method is stopped, the isListenerRunning flag is set
to false, and the worker thread is stopped.

Request processing

Incoming requests posted to RequestMap are passed to processGetRequest
method which executes the following:

➤ Check the incoming request.
➤ Extract the customerID.
➤ Check for the customer data in the grid.
➤ Fetch the data from the FinanceMap if the customer data is in the grid.

Example 5-10   MediatorAgent - processGetRequest method

try {
    // get customer from grid
    m1 = getFromGrid(customerID);
    if (m1 == null) {
        notInGrid = true;
    }
}
catch (ObjectGridException e) {
    // not in grid
    notInGrid = true;
}
Example 5-11 MediatorAgent - getFromGrid method

MapMessage m1 = (MapMessage) (financeMap.get(customerID));
return m1;

- Check the version of the customer data in the grid with that on the data server.
- If the data is not in the grid or it is not synchronized with the data server, then fetch the customer data from the data server.

Example 5-12 MediatorAgent - processGetRequest method

try {
    String customerID = request.getString("cID");
    MapMessage m1 = null;
    MapMessage m2 = null;
    boolean notInGrid = false;
    boolean notInSync = false;
    boolean isDataFromServer = false;
    try {
        // get customer from grid
        m1 = getFromGrid(customerID);
        if (m1 == null) {
            notInGrid = true;
        }
    }
    catch (ObjectGridException e) {
        // not in grid
        notInGrid = true;
    }
    if (notInGrid) {
        // not in grid, get from Finance Data server
        m1 = getFromServer(customerID, null, "getCustomerInfo");
        isDataFromServer = true;
    }
    else {
        // customer data in grid, check version info to make sure it is current
        String versionID = m1.getString("versionID");
        m2 = getFromServer(customerID, versionID, "getVersionInfo");
        String status = m2.getString("isInSync");
        if (status.equals("N")) {
            // Not in sync, go get latest from Finance Data server
            notInSync = true;
            isDataFromServer = true;
            m1 = getFromServer(customerID, null, "getCustomerInfo");
        }
    }
}
if ((notInGrid)||(notInSync)) {
    // set return message so that it can be published to data to grid
    message2Publish = m1;
}
sendReplyMsg(request,m1,isDataFromServer);

- Publish the customer data to the grid if it was fetched from the data server.
- Create a response message using the customer data.
- Indicate source of the customer data. For example, “Data From Grid # Partition X” or “Data From Server.”
- Post response message to the messaging queue for FinanceServlet.

**MediatorAgent: Get data from server**

To retrieve the customer data from the data server, MediatorAgent sends JMSMessages to the DataServer. Two types of messages are sent by MediatorAgent

- getVersionInfo
  The getVersionInfo message is used to retrieve the version of the customer data. Version information is used to check whether the data in the grid is the latest customer data. The JMSType is set to getVersionInfo, customerID is sent as “cID” and version id as “versionID.”

- getCustomerInfo
  The getCustomerInfo message is sent as the request for retrieving the customer financial data. JMSType is set to getCustomerInfo, and customerID is sent as “cID.”

The JMS resources are initialized in the MediatorAgent shardActivated() method. The resources are used to create temporary queues, messaging producers, and consumers. The producer sends a message, the consumer waits for the message with time-out. The response from the data server is processed by MediatorAgent.

**Example 5-13  MediatorAgent- getFromServer method**

```java
try {
    MapMessage outMessage = serverJMSSession.createMapMessage();
    tempReplyQueue = serverJMSSession.createTemporaryQueue();
    serverQueueReceiver =
        serverJMSSession.createConsumer(tempReplyQueue);
    outMessage.setString("cID", customerID);
    if (versionID != null){
```
outMessage.setString("versionID", versionID);
}
outMessage.setJMSType(msgType);
outMessage.setJMSDestination(serverDestQueue);
outMessage.setJMSReplyTo(tempReplyQueue);
serverQueueSender.send(outMessage);
MapMessage returnMessage =

(MapMessage)serverQueueReceiver.receive(Long.parseLong(serverMessageWait));
return returnMessage;
}

MediatorAgent: Publish to grid

The customer data is published to ObjectGrid using transactions. Transactions are executed within ObjectGrid session contexts using session.begin and session.commit.

Use ObjectMap.insert for new customer data and updates for existing customer data. When new customer data is inserted into maps, always use transactions to ensure that the grid is notified of the changes.

Note: Nested transactions are not supported so ensure that there are no active transactions before starting a new transaction. This will result in the TransactionAlreadyActive Exception being thrown. Also, ensure that sessions are rolled back if there are failures.

Example 5-14 MediatorAgent: publishToGrid method

try {
    mediatorSession.begin();
    financeMap.getForUpdate(customerID);
    financeMap.put(customerID, m1);
    mediatorSession.commit();
}

MediatorAgent: send response

The JMS resources required for sending response message are initialized in the shardActivated method. The destination is taken from the request message from FinanceServlet. The servlet creates a temporary queue destination and places it in the JMSReplyTo attribute of the Message. This destination is used to create a MessageProducer for sending message.
The customer data from the grid which is itself a MapMessage is used to populate the response message. The source of the data is used to differentiate between customer data retrieved from the data server and from grid partition.

The response is posted to the messaging queue.

**Example 5-15  MediatorAgent: sendReplyMsg method**

```java
try {
    Destination replyTo = request.getJMSReplyTo();
    MessageProducer resultProducer =
        clientJMSSession.createProducer(replyTo);
    MapMessage replyMsg = clientJMSSession.createMapMessage();
    Enumeration e = result.getMapNames();

    // put attributes into reply message
    while (e.hasMoreElements()) {
        String ekey = (String) e.nextElement();
        Object evalue = result.getObject(ekey);
        replyMsg.setObject(ekey, evalue);
    }

    // indicate source of data
    if (isDataFromServer) {
        replyMsg.setString("source", "Data from Server");
    } else {
        replyMsg.setString("source", "Data from ObjectGrid - processed in partition \\
          partition #" + partitionID);
    }

    resultProducer.send(replyMsg);
}
```

### 5.1.3 MediatorLoader

The MediatorLoader gets activated during grid initialization. It implements the com.ibm.websphere.objectgrid.plugins.Loader interface, specifically the preLoadMap method. The data for a configurable number of customers is fetched from the data server. ObjectGrid determines the partitions in which to place the customer data.
The following is executed as part of the preload activity:

- Initialize the JMS connection factory, connection and session.
- Get the JMS destination for the data server message call.
- Initialize the message consumer with a temporary queue destination.
- Create a message producer to send messages.

Example 5-16  MediatorLoader: preloadmap method

```java
try{
    // setup commections
    InitialContext initialContext = new InitialContext();
    ConnectionFactory qcf = (ConnectionFactory)
        initialContext.lookup(serverDestFactoryJndiName);
    serverJMSConnection = qcf.createConnection();
    serverJMSConnection.start();
    serverJMSSession = serverJMSConnection.createSession(false,Session.AUTO_ACKNOWLEDGE);
    serverDestQueue = (Destination) initialContext.lookup(serverDestQJndiName);
    tempReplyQueue = serverJMSSession.createTemporaryQueue();
    serverQueueReceiver = serverJMSSession.createConsumer(tempReplyQueue);
    serverQueueSender = serverJMSSession.createProducer(serverDestQueue);
    serverQueueSender.setDeliveryMode(DeliveryMode.NON_PERSISTENT);
    MapMessage outMessage = serverJMSSession.createMapMessage();

    // get partition information
    PartitionManager partitionMgr = financeMap.getPartitionManager();
    System.out.println("MediatorLoader.preloadMap() - Number of partitions is - " +
        partitionMgr.getNumOfPartitions());
    int partitionID = financeMap.getPartitionId();
    System.out.println("MediatorLoader.preloadMap() - PartitionID #" + partitionID);

    // Build list of data to preload for this partition
    Vector<MapMessage> msglist= new Vector<MapMessage>();
    int k = Integer.parseInt(preLoadLimit);
    outMessage.setJMSType("getCustomerInfo");
    outMessage.setJMSDestination(serverDestQueue);
    outMessage.setJMSReplyTo(tempReplyQueue);
    for (int i=0; i< k; i++) {
        System.out.println("MediatorLoader.preloadMap() check on " + i);

        // check if customer key should be preloaded in this partition
        if (partitionMgr.getPartition(String.valueOf(i)) == partitionID){
            // This element belongs to this partition, fetch data and add to list for loading
            System.out.println("MediatorLoader::preloadMap - " + i + " is in " + partitionID);
        }
    }
}
```
outMessage.setString("cID", String.valueOf(i));
serverQueueSender.send(outMessage);
long serverwait = Long.parseLong(serverMessageWait);

try {
    MapMessage returnMessage;
    returnMessage = (MapMessage) serverQueueReceiver.receive(serverwait);
    msglist.addElement(returnMessage);
}

catch (Exception e) {
    System.out.println("MediatorLoader.preloadMap() - exception - " + e);
    e.printStackTrace();
}

System.out.println("MediatorLoader.preloadMap() - Now go load grid with - " +
msglist.size() + " entries");
loadGrid(gridSession, financeMap, msglist);

- For each customerID, the partition is retrieved and checked to ensure that it
  matched with the current partition before the customer data is loaded. If it is
  not matched for the current partition, the customer data is skipped and
  processing proceeds with the next customerID.

  **Note:** If a key does not belong to the particular partition being updated, the
  grid load will result in failure.

*Example 5-17  partition ID verification in the MediatorLoader

```java
// check if customer key should be preloaded in this partition
if (partitionMgr.getPartition(String.valueOf(i)) == partitionID){
    // This element belongs to this partitoin go fetch data
    // and add to list for loading
    System.out.println("MediatorLoser::preloadMap - " + i + " is in " +
    partitionID);
    outMessage.setString("cID", String.valueOf(i));
}
```

- Collate the data received from the server iteratively into a customer list.
- Populate the grid using the list of customer data.
Example 5-18  MediatorLoader: loadGrid method

```java
try {
    ObjectMap map = gridSession.getMap(financeBackMap.getName());
    gridSession.beginNoWriteThrough();
    Iterator<MapMessage> elist = msglist.iterator();
    MapMessage returnMessage=null;
    System.out.println("MediatorLoader.loadGrid() number to load - " +
        msglist.size());
    // loop through list of elements to load into grid
    while(elist.hasNext()){
        returnMessage = elist.next();
        map.put(returnMessage.getString("cID"), returnMessage);
    }
    gridSession.commit();
}
```

The gridSession is started with a beginNoWriteThrough, which ensures that the
changes made by the transaction are only applied to the BackingMap and not
passed back to loader's get method.

**Note:** In this example scenario we use the MapMessages that is being passed
in JMS queues itself to be stored into the grid for simplicity. In real life scenario
use Objects to place into ObjectGrid

### 5.2 Setting up the mediator environment

This is the second part of the two-phase installation of the sample application
scenario. In this phase, we introduce eXtreme Scale cache to the base
application environment. It is an implementation of the eXtreme Scale mediator
application pattern. It shows how a grid can be used to cache data and minimize
the number of requests and amount of data retrieved from the data server.
Setting up the environment includes configuring the grid, modifying JMS
resources, and installing the ITSOMediator application in the environment.
5.2.1 ITSOFinanceGrid configuration

The ITSOFinanceGrid is deployed in a WebSphere Extended Deployment V6.1 environment, using the Catalog Service which automatically lays out data (primaries and replicas) in a balanced fashion across the nodes in the cluster. The steps for creating ITSOFinanceGrid in a dynamic cluster are as follows:

1. Create a dynamic cluster in a WebSphere Extended Deployment environment.
2. Install WebSphere eXtreme Scale.
3. Create an application EAR file and the following two configuration files in the META-INF directory.
   - objectGrid.xml
     The XML file that specifies the details of the grids to be created, the backing maps, and so on.
   - objectGridDeployment.xml
     The XML file that specifies the partitioning and replication policies to be applied to the backing maps defined in the grid deployment file.

Note: The case sensitive configuration file names must objectGrid.xml and objectGridDeployment.xml for the eXtreme Scale runtime to recognize and launch the Grid.
4. Install the application into the dynamic grid.

When the application starts, the eXtreme Scale runtime looks for the two XML files in the EAR file and if they are found, the eXtreme Scale container is started and registered with the Catalog Service based on the specification files provided.

**objectGrid.xml**

The objectGrid.xml file contains the grid information.

- The `<objectGrid>` element specifies the name of the grid as ITSOFinanceGrid.
- The listener related for the grid is attached to the grid as a Java Bean with the name “ObjectGridEventListener.” The class that implements the listener interface is specified in the class name.
- The properties for the bean are specified
- The `<backingMap>` elements specify the maps that are to be handled by the grid.

*Figure 5-8  ITSOMediator deployment environment*
The maps have pluginCollectionRef which is linked to the plug-ins that are related to the BackingMap and the classes that implement them. For example, Loader is a plug-in for the BackingMap FinanceMap.

The locking strategy is specified. It can be Optimistic or Pessimistic. By specifying Pessimistic, local JVM cache is not maintained and no local locking is done. Usually the Optimistic locking strategy is the default where the Grid partitions are distributed and have to be accessed remotely. In our example, the grid and the application accessing the grid are colocated, resulting in the Pessimistic lock strategy setting.

Example 5-19  objectGrid.xml specifying ObjectGrids

```xml
<objectGrids>
  <objectGrid name="ITSOFinanceGrid">
    <bean id="ObjectGridEventListener"
      className="com.ibm.itso.finance.mediator.MediatorAgent">
      <property name="serverDestFactoryJndiName"
        type="java.lang.String" value="jms/ITSOConFactory1" description="" />
      <property name="serverDestQJndiName"
        type="java.lang.String" value="jms/ITSOFinanceDataQ" description="" />
      <property name="clientDestFactoryJndiName"
        type="java.lang.String" value="jms/ITSOConFactory" description="" />
      <property name="requestThreadWait"
        type="java.lang.String" value="100" description="" />
      <property name="serverMessageWait"
        type="java.lang.String" value="15000" description="" />
    </bean>

    <backingMap name="FinanceMap"
      pluginCollectionRef="FinanceMap" />
    <backingMap name="RequestMap"
      pluginCollectionRef="RequestMap" lockStrategy="PESSIMISTIC" />
  </objectGrid>
</objectGrids>
```

The <backingMapPluginCollections> element details the collection of plug-ins.

For each map we specify the <backingMapPluginCollection> with id as the name specified in pluginCollectionRef which is the name of the Map for simplicity. Here we specify the plug-in collection for FinanceMap.
For FinanceMap, the loader is a plug-in and is specified as a JavaBean with properties.

The loader bean is identified by the name as Loader for the grid to be pre-loaded and it has a class name which specifies the name of the class implementing the Loader interface.

Example 5-20  objectGrid.xml specifying BackingMapPluginCollections

```xml
<backingMapPluginCollections>
  <backingMapPluginCollection id="FinanceMap">
    <bean id="Loader"
      className="com.ibm.itso.finance.mediator.MediatorLoader">
      <property name="serverDestFactoryJndiName"
        type="java.lang.String" value="jms/ITSOConFactory1" description="" />
      <property name="serverDestQJndiName"
        type="java.lang.String" value="jms/ITSOFinanceDataQ" description="" />
      <property name="preLoadLimit" type="java.lang.String"
        value="20" description="" />
      <property name="serverMessageWait"
        type="java.lang.String" value="15000" description="" />
    </bean>
  </backingMapPluginCollection>
  <backingMapPluginCollection id="RequestMap" />
</backingMapPluginCollections>
```

objectGridDeployment.xml

objectGridDeployment.xml specifies the deployment properties, including information about partitions, replicas number of servers to start only. You do not have to specify the ip address of the nodes, machine names, or the cluster name. This helps the dynamic cluster to be flexible. It can dynamically handle more nodes in the clusters without changing the files.

- `<objectGridDeployment>` element specifies the name of the Grid. In this sample application it is ITSOFinanceGrid.

- Specify the MapSet with maps as FinanceMap and RequestMap.
Specify the number of partitions, replications, and number of initial containers. The number of initial containers is optional and specifies that n number of containers must be activated. During the design stage, it is better to decide on keys, maps, and partitions; specifically on how the data must be partitioned across to efficiently communicate with the Grid.

Example 5-21  ObjectGridDeployment.xml sample

```xml
<?xml version="1.0" encoding="UTF-8"?>
<deploymentPolicy xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://ibm.com/ws/objectgrid/deploymentPolicy
   ../deploymentPolicy.xsd"
   xmlns="http://ibm.com/ws/objectgrid/deploymentPolicy">

   <objectgridDeployment objectgridName="ITSOFinanceGrid">
      <mapSet name="FinanceMapSet" numberOfPartitions="3"
              minSyncReplicas="1" maxSyncReplicas="1" maxAsyncReplicas="0"
              numInitialContainers="3">
         <map ref="FinanceMap" />
         <map ref="RequestMap" />
      </mapSet>

   </objectgridDeployment>

</deploymentPolicy>
```

Note: If you specify the Maps inside a single MapSet, you ensure that the set of data for the maps are distributed together across the partitions. This means that partition1 will have a part of both FinanceMap and Request Map.

On the other hand, if you do not specify so, they may be split across. For example, partition1 will have part of FinanceMap only and partition2 will have part of RequestMap.

In our example, both RequestMap and FinanceMap use the same customer id as key and they must be in the same partition for the Specific Mediator Agent to pick the request from RequestMap and Data from FinanceMap quicker.

The other advantage with this approach is that when the MediatorAgent which is activated by request to the RequestMap; checks against the local partition for Data and it knows that the data is not in this partition, it can be sure that the data will not be in the grid.
5.2.2 Install the mediator application

In section 4.2, “Setting up the base application environment” on page 89, we installed and configured the base finance application environment. We tested the Web application ITSOFinance and it communicated with the ITSOFinanceData application through a JMS queue to retrieve the customer finance and banking data. In this section we will modify the base environment and introduce the mediator application ITSOMediatorApp.

Before making any changes we must first stop the applications and servers that were installed and run in section 4.2, “Setting up the base application environment” on page 89. From the administrative console, take the following steps:

1. Click **Servers → Application Servers** to get to the Application servers page shown in Figure 5-9 on page 149.
2. Select **ITSOFinanceDataServer** and **ITSOFinanceServer** and click **Stop**.

3. Click **OK** on the Stop Servers page to confirm that you want to stop the servers.

4. Click **OK** on the Server Status Feedback page after the servers have stopped.

---

---

**Figure 5-9**  Running application servers
With the servers stopped, we are ready to start the installation and configuration process for the mediator application. We have to perform the following steps:

1. Create the mediator dynamic cluster.
2. Update messaging services.
3. Install the ITSOMediator application.
4. Run the mediation application.

### 5.2.3 Create the mediator dynamic cluster

We must create a dynamic cluster to host the ITSOMediator application and the grid. From the administrative console, perform the following steps:

1. Select **Servers → Dynamic clusters**.
2. Click **New** to create a new cluster.
3. Select the Server type on the “Select a dynamic cluster server type” page. Leave the Server type as **WebSphere application server**.
4. Click **Next**.

![Image](image_url)

**Figure 5-11 Select dynamic cluster server type**

The second step is the selection of the membership method. On “Select the membership method” page, perform the following steps:

1. Enter **MediatorDC** in the Dynamic cluster name dialog box.
2. Verify that **Prefer local enabled** is checked.
3. Click **Next**.
In the third step, you define the dynamic cluster members. We want to edit the rule for dynamic cluster membership to include at least three nodes but not the ODR node. Perform the following steps:

1. Click **Subexpression builder**.

2. On the Subexpression page. (See Figure 5-13 on page 153.)
   a. Leave the Logical operator as **and**.
   b. Select **Node name** for the Select operand field.
   c. Change operator to **Not Equals**. (<>)
   d. Enter `odrnodenode01` for the name of the ODR node in the Value field.
   e. Click the Generate subexpression button to generate the sub expression.
3. Click **Append** and then **Close**.

**Note:** Click **Preview membership** to confirm that there are at least three members in the DynamicCluster but not the ODR node.

4. Select a dynamic cluster template.
   a. Accept the default defaultXD.
   b. Click **Next**.

5. Specify dynamic cluster specific properties.
   a. Click “Keep multiple instances started at all times” and enter 3 for the Number of instances.
   b. Click **Next**.
6. Click **Finish** on the Summary page.

7. Click **Save** to complete the creation of the MediatorDC dynamic cluster.
5.2.4 Update messaging services

The base application environment setup must be updated to accommodate the mediator. We will also retarget the Activation Specification so that the ITSOFinanceData application will not require any code changes. To do so, we will perform the following operations:

1. Update the service integration bus.
2. Update an existing and configure a new messaging queue.
3. Update an existing and configure a new activation specification.
Update the service integration bus

The first step in updating the Service Integration Bus is to add MediatorDC as a bus member. From the administrative console, perform the following steps:

1. Select Service integration → Buses → Bus members → Add.
   
   The first step in adding a new bus member is to the selection of the bus member from the Select server, cluster or WebSphere MQ server page. To do so:

2. Click the Cluster radio button.

3. Select MediatorDC.

4. Click Next.

5. Select type of message store.
   
a. Accept the File store selected for the message store.

b. Click Next.

![Select MediatorDC as a new bus member](image)
6. Provide the message store properties.
   a. Enter `${LOG_ROOT}/Mediator_DC` for the Log directory path.
   b. Enter `${WAS_INSTALL_ROOT}/PERMSTORE/MEDIATOR_DC` for the Permanent store directory path.
   c. Click Next.

![Integrated Solutions Console - Microsoft Internet Explorer](image)

**Figure 5-17** Provide the message store properties

7. Click Finish on the “Confirm the addition of the new bus member” page to complete the new bus member addition.

8. Save the configuration changes.

**Create new queue destination**

Next we create a new queue destination. From the administrative console, perform the following steps:

1. Select **Service integration → Buses → ITSOBUS → Destinations → New.**
2. Leave the Destination type as **Queue** and click **Next.**
3. Set the queue attributes.
   a. Enter **ITSOFinanceQDest** for the Identifier.
   b. Click **Next**.

![Integrated Solutions Console](image)

Figure 5-18  Set the queue attributes

4. Assign the queue to a bus member.
   a. Select **Cluster=MediatorDC** from the Bus members drop down list.
   b. Click **Next**.

5. Click **Finish** on the “Confirm queue creation” page to confirm the queue creation.

6. Click **Save** to save the configuration changes.
Update the ITSOFinanceQ

To avoid having to make changes to the ITSOFinance application, we must update the ITSOFinanceQ queue to use the newly created destination ITSOFinanceQDest. This queue will be used for communication between the ITSOFinance Servlet and the ITSMediatorApp application. From the administrative console, perform the following steps:

1. Select Resources → JMS → JMS providers.
2. Click Default messaging provider, as in Figure 5-20 on page 160, to select the Default messaging provider.
3. Click **Additional Properties → Queues** on the JMS providers page as shown in Figure 5-21 on page 161.
4. Click **ITSOFinanceQ** to open the ITSOFinanceQ configuration page.
5. On the ITSOFinanceQ configuration page:
   a. Select ITSOFinanceQDest from the Queue name drop down menu.
   b. Click OK and Save to save the changes.
Chapter 5. The eXtreme Scale mediator application

Create a new messaging queue

A new queue is required for communication between ITSOMediatorApp and ITSOFinanceData applications.

**Note:** Resources creation is done at cell level to enable visibility for all nodes

From the administrative console perform the following steps:

1. Select **Resources → JMS → JMS providers**.
2. Verify that the scope is set to cell level. In this base application example it is `cell=dmgrnodeCell01`. (See Figure 5-24 on page 164.)
3. Select **Default messaging provider**, as in Figure 5-24 on page 164.
4. Click **Additional Properties → Queues** on the Default “messaging provider” page to create a new queue.
5. Click **New** to create a new messaging queue.

6. Configure the Queue as shown in Figure 5-26 on page 166.
   a. Enter **ITSOFinanceDataQ** for the Queue name.
   b. Enter **jms/ITSOFinanceDataQ** for the queue JNDI name.
   c. Enter the Connection information.
      i. Select **ITSOBUS** as the bus name.
      ii. Select **ITSOQDest** as the queue name for bus destination.
      iii. Click **Apply** and **Save** to save the queue configuration.
Update the base application activation specification

In order not to modify the ITSOFinanceData application, we must update the ITSOActSpec to point to the newly created ITSOFinanceDataQ. From the administrative console, perform the following steps:

1. Select Resources → JMS → JMS providers.
2. Select Default messaging provider, as in Figure 5-27 on page 167.
Figure 5-27  Select the Default messaging provider

3. Click **Additional Properties → Activation specifications** on the JMS providers page, as shown in Figure 5-28 on page 168.
4. Click **ITSOActSpec** to open the General Properties page.
5. On the ITSOActSpec General Properties page:
   a. Change the Destination JNDI name to \texttt{jms/ITSOFinanceDataQ}.
   b. Click \textbf{Apply} and \textbf{Save} to save the changes to the activation specification.
Create the mediator activation specification

To create the mediator activation specification, start from the administrative console and follow these steps:

1. Select Resources → JMS → JMS providers.

2. Verify that the scope is set to cell level, in this base application example it is cell=dmgrnodeCell01. (See Figure 5-31 on page 171.)

3. Click Default messaging provider to select the Default messaging provider as in Figure 5-31 on page 171.
4. Click **Additional Properties → Activation specifications** to create a new activation specification as in Figure 5-32 on page 172.
5. Click **New** to create a new activation specification.

6. Configure the activation specification as shown in Figure 5-33 on page 173.
   a. Enter **ITSOMediatorActSpec** for the name.
   b. Enter **jms/ITSOMediatorActSpec** for the JNDI name.
   c. Enter the destination information.
      i. Select **Queue** as the destination type.
      ii. Enter **jms/ITSOFinanceQ** as the Destination JNDI name.
      iii. Select **ITSOBUS** as the bus name.
   d. Click **Apply** and **Save** to save the activation specification.
5.2.5 Install the ITSOMediator application

Now that the setup is complete, the ITSOMediator Application must be installed. Installation of the application is similar to the installation of ITSOFinance and ITSOFinanceData applications. The ITSOMediatorApp is installed on the MediatorDC. The ITSOMediatorApp application is packaged as ITSOMediatorApp.ear.

**Note:** Before you start the installation process, place the ear file in a folder on the local drive for example in c:\apps

From the Integrated Services Console follow these steps:

1. Select **Applications → Enterprise Applications → Install.**
2. Specify the path to the local folder containing the ear file and click **Next.**
3. Select installation options.
   a. Accept the default settings on the “Select installation options” page.
   b. Click Next.

4. Map modules to dynamic cluster.
   a. Click the Select check box to select Finance module.
   b. Click the Clusters and Servers drop down menu and select MediatorDC as the cluster to host the application.
   c. Click Apply next to the Clusters and Servers list.
   d. Verify that the server name associated with the Finance module is MediatorDC.
   e. Click Next.

5. Review the summary on the Summary page.

6. Click Finish after reviewing and making any changes.

7. Click Save to save the installation options.

5.3 Testing the mediation application

Before you can run the mediated finance data application, the servers and dynamic cluster must be started. The applications installed must also be started. Start ITSOFinanceDataServer and ITSOFinanceServer application servers. When the servers are up, you can start the MediatorDC dynamic cluster members in a similar manner. The ITSOMediatorApp preloads data from the ITSOFinanceData application into the grid partitions when it starts up.

Test the mediation application

Enter the URL for the application as http://<Your ODR node machine name>/Finance/FinanceServlet. The application processes requests for data as follows:

- Customers 0 through 19 are pre-loaded into partitions.
- Customers 20 through 29 are not pre-loaded. The initial request will fetch them from the ITSOFinanceData application. Subsequent requests should get the information from the grid.
Customers 30 through 39 are not pre-loaded. The initial request will fetch them from the ITSOFinanceData application. ITSOFinanceData randomizes the version for customers 30–39 simulating the possibility that the data could be changed from some other source, requiring the mediator to check and reload. The version checking result in entries being reloaded again from the ITSOFinanceData application though the data is already in the grid.

![Finance Application - Version 1.0.0](image)

**Finance Application - Version 1.0.0**

**Customer Name**

Customer ID: 2

Customer Name: Billy


Data from ObjectGrid - processed in partition #2

**Personal Information**

Tax ID: 2-2-1002

Figure 5-34  *The WebSphere eXtreme Scale mediator application*

The mediation application displays the source of the retrieved data for each request, indicating the partition that processed it.
Additional material

This paper refers to additional material that can be downloaded from the Internet as described.

Locating the Web material

The Web material associated with this paper is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser at:

ftp://www.redbooks.ibm.com/redbooks/REDP4398

Alternatively, you can go to the IBM Redbooks Web site at:

http://www.ibm.com/redbooks

Select the Additional materials and open the directory that corresponds with the IBMRedpaper form number, REDP4398.
Using the Web material

The additional Web material that accompanies this paper includes the following files:

<table>
<thead>
<tr>
<th>File name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4398.zip</td>
<td>4398 Zipped Code Samples for Finance data application</td>
</tr>
</tbody>
</table>

System requirements for downloading the Web material

The following system configuration is recommended:

- **Hard disk space:** 40 GB Disk
- **Operating System:** Windows® XP
- **Processor:** Minimum 1 GHz
- **Memory:** Minimum 1 GB

How to use the Web material

Create a subdirectory (folder) on your workstation, and unzip the contents of the Web material zip file into this folder.

Description of sample code

Table 5-1 describes the contents of the 4398.zip file after unzipping.

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ITSOFinance.ear</td>
<td>The ear file for the ITCSO Finance client application</td>
</tr>
<tr>
<td>ITSOFinanceData.ear</td>
<td>The base ITSO Finance Data application</td>
</tr>
<tr>
<td>ITSOMediatorApp.ear</td>
<td>The ITSO Finance mediator application</td>
</tr>
</tbody>
</table>
## Abbreviations and acronyms

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>CRUD</td>
<td>Create, Read, Update and Delete</td>
</tr>
<tr>
<td>DB2®</td>
<td>IBM universal Data Base 2</td>
</tr>
<tr>
<td>DCS</td>
<td>Distribution and Consistency Services</td>
</tr>
<tr>
<td>EAR</td>
<td>Enterprise ARchive</td>
</tr>
<tr>
<td>EJB</td>
<td>Enterprise JavaBean</td>
</tr>
<tr>
<td>ESB</td>
<td>Enterprise Service Bus</td>
</tr>
<tr>
<td>HA Manager</td>
<td>High Availability Manager</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transfer Protocol</td>
</tr>
<tr>
<td>IBM</td>
<td>International Business Machines Corporation</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>ITSO</td>
<td>International Technical Support Organization</td>
</tr>
<tr>
<td>Java EE</td>
<td>Java Platform Enterprise Edition</td>
</tr>
<tr>
<td>JCA</td>
<td>Java EE Connector Architecture</td>
</tr>
<tr>
<td>JMS</td>
<td>Java Messaging Service</td>
</tr>
<tr>
<td>JNDI</td>
<td>Java Naming and Directory Interface</td>
</tr>
<tr>
<td>JSP</td>
<td>Java Server Page</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>LDAP</td>
<td>Lightweight Directory Access Protocol</td>
</tr>
<tr>
<td>MDB</td>
<td>Message Driven Bean</td>
</tr>
<tr>
<td>ODR</td>
<td>On-Demand Router</td>
</tr>
<tr>
<td>POJO</td>
<td>Plain Old Java Object</td>
</tr>
<tr>
<td>PSSB</td>
<td>Partitioned Stateless Session Bean</td>
</tr>
<tr>
<td>SPT</td>
<td>Stream Processing Technology</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
</tr>
<tr>
<td>WAR</td>
<td>Web ARchive</td>
</tr>
<tr>
<td>WAS</td>
<td>WebSphere Application Server</td>
</tr>
<tr>
<td>XML</td>
<td>eXtensible Markup Language</td>
</tr>
<tr>
<td>XTP</td>
<td>Extreme Transaction Processing</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
</tr>
<tr>
<td>TSO</td>
<td>Technical Support Organization</td>
</tr>
<tr>
<td>WAS</td>
<td>WebSphere Application Server</td>
</tr>
<tr>
<td>XTP</td>
<td>Extreme Transaction Processing</td>
</tr>
<tr>
<td>XML</td>
<td>eXtensible Markup Language</td>
</tr>
<tr>
<td>XTP</td>
<td>Extreme Transaction Processing</td>
</tr>
</tbody>
</table>

© Copyright IBM Corp. 2008. All rights reserved.
Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this paper.

Online resources

These Web sites are also relevant as further information sources:

- WebSphere Extended Deployment Version 6.1 Information Center
  
  http://publib.boulder.ibm.com/infocenter/wxdinfo/v6r1/index.jsp

- ObjectGrid Version 6.1 Documentation Wiki
  
  http://www.ibm.com/developerworks/wikis/display/objectgrid

How to get Redbooks

You can search for, view, or download Redbooks, Redpapers, Technotes, draft publications and Additional materials, as well as order hardcopy Redbooks, at this Web site:

http://www.ibm.com/redbooks

Help from IBM

IBM Support and downloads

http://www.ibm.com/support

IBM Global Services

http://www.ibm.com/services
Index

A
activation specification 86, 109, 111, 126, 169, 171–172
AgentManager 58, 61–62
aggregation 62
application server 92
asymmetric clustering 2

B
BackingMap 18–22, 24–25, 31–33, 35–39, 66, 72–73, 142, 145–146
BackingMap plug-in 66
backingMapPluginCollection 145

C
cache 18, 70
Cache Loader 18
catalog server 8
client ObjectGrid instance 26
ClientClusterContext 24
ClientReplicableMap 24
ClientReplicableMap.Mode 25
com.ibm.websphere.objectgrid 23–25
connection 140
connection factory 103, 140
container services 4
containsKey 34–35
COPY_ON_READ 31
COPY_ON_READ_AND_COMMIT 31–32
COPY_ON_WRITE 32
CopyKey 19
CopyMode 19, 25, 37
core group management 10
core group manager 10
core group server 11
createObjectQuery 46
createQuery 52

D
DataGrid API 21, 23, 59
defaultXD 153
deployment descriptor 86
destination 99, 157, 165, 172
distribution and consistency services (DCS) 11
dynamic cluster 143, 150, 152, 174

E
EJB workload partitioning 2
entity map 12
entity maps 12
entity metadata descriptor file 45
entity schema 40
EntityAgentMixin 58, 61
EntityManager 38–40
EntityManager API 21, 30
EntityManager Query API 52
EntityMap API 23
EntityMetaData 53
EntryErrorValue 59
Evictor 66
eXtreme Scale plug-in 66
Extreme Transaction Processing (XTP) 2

F
flush 35

G
getAddress 55
getCustomerInfo 84
getEntityManager 39
getForUpdate 35
getJavaMap 34
getResultIterator 50, 53
getResultMap 50–51, 54
getTransaction 40
getVersionInfo 85
ObjectGridManager 24, 26–27
ObjectGridManagerFactory 25
ObjectMap 12, 20–21, 23–24, 31–37, 46–47, 50–52, 54, 123, 133, 138
ObjectMap API 12, 21, 23, 32, 35
ObjectMap operation 12
ObjectMaps 12
ObjectQuery 46, 48, 50
ObjectQuery API 45
ObjectTransformer 66
ODR 153
On Demand Router (ODR) 117
on demand router (ODR) 2
onMessage 84
Optimistic 145
OptimisticCallback 66

P
parallel processing 70
partition 5, 32, 58, 71, 133, 174
partition query 58
partitioned stateless session bean (PSSB) 2
PartitionHandlerLocal interface 2
partitionID 134
Partitioning facility 2
partitioning facility 2
partitioning key 16
PartitionManager 24
Pessimistic 145
placement service 9
Plain Old Java Object (POJO) 38
primary shard 6
processVersionMessage 85
put 34, 37
putAll 34

Q
Query API 21, 23, 45
Query object 54
queue 81, 106, 157–158, 163–165

R
ReadOnly 19
Redbooks Web site 181
Contact us xii
ReduceGridAgent 59
remove 34, 37

replica shard 6
RequestMap 134–135
rollback 31

S
server ObjectGrid instance 28
service integration bus 95, 155–156
Session 21–22, 24, 29, 31, 39, 54, 61
session 20
Session instance 28
setCopyMode 31, 37
shard 6–8, 28, 133
shardActivated 133
shardDeActivated 135
ShardEvents 29
stream event 56
stream map 56
stream maps 57
Stream Processing Technology (SPT) 56
stream query 56–57
SubjectSource 66
SubjectValidation 66
synchronous replica 6

T
thread 31, 134
TimeToLive 20
touch 37
transaction cache 36
transaction context 22
TransactionCallback 66
TtlEvictorType 20
TTLType 25
Tuple 54–55
TxID 24

V
valueInterfaceClass 32
view map 57

X
XTP 2
WebSphere eXtreme Scale Mediator Pattern

Solution design considerations

The IBM® WebSphere® Extended Deployment provides dynamic and high performance computing infrastructure across range of application servers and execution environments. In addition it enhances the utilization and operational management of the infrastructure. The WebSphere eXtreme Scale feature provides high-end caching and transaction partitioning capabilities. WebSphere eXtreme Scale delivers performance improvements across a wide range of application scenarios.

Application pattern

This IBM Redpaper publication describes the concepts, programming model and implementation of the mediator application pattern using WebSphere eXtreme Scale. The WebSphere eXtreme Scale mediator pattern promotes loose coupling between architectural components while maintaining unified interfaces. Usually large and distributed systems are fairly complex making it difficult and expensive to change. The WebSphere eXtreme Scale mediator pattern enables the introduction of the eXtreme Scale cache into distributed messaging and service oriented applications, without altering the interfaces and behavior, to provide significant improvement in performance, availability and scalability.

Sample implementation

INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

For more information:
ibm.com/redbooks

REDP-4398-00