

# Pivotal GemFire Developer Course

Instructor-led Training

Version 9.0.2a

Pivotal

# Copyright Notice

Copyright © 2017 Pivotal Software, Inc. All rights reserved. This manual and its accompanying materials are protected by U.S. and international copyright and intellectual property laws.

Pivotal products are covered by one or more patents listed at <http://www.gopivotal.com/patents>.

Pivotal is a registered trademark or trademark of Pivotal Software, Inc. in the United States and/or other jurisdictions. All other marks and names mentioned herein may be trademarks of their respective companies. The training material is provided “as is,” and all express or implied conditions, representations, and warranties, including any implied warranty of merchantability, fitness for a particular purpose or noninfringement, are disclaimed, even if Pivotal Software, Inc., has been advised of the possibility of such claims. This training material is designed to support an instructor-led training course and is intended to be used for reference purposes in conjunction with the instructor-led training course. The training material is not a standalone training tool. Use of the training material for self-study without class attendance is not recommended.

These materials and the computer programs to which it relates are the property of, and embody trade secrets and confidential information proprietary to, Pivotal Software, Inc., and may not be reproduced, copied, disclosed, transferred, adapted or modified without the express written approval of Pivotal Software, Inc.

# GemFire Developer Course

A Look at Client-side and Server-side  
Development for GemFire

# Course Objectives

After this class, you should be able to do the following:

- Describe GemFire and its role in an enterprise
- Understand the Client/Server architecture
- Understand the programming model for building GemFire client applications
- Be able to take advantage of event processing, function execution and cache management
- Be familiar with the Spring Data GemFire project

# Course Introduction

- Course Duration: 4 days
- Course is designed for Windows, Linux and Mac
  - 50% Theory
  - 50% Lab
- Students should have the following
  - Lab documentation
  - Student handout
  - Lab installation files either on USB drives or provided by instructor

# Course Prerequisite

- Knowledge of Core Java
- Knowledge of Object modeling
- Working knowledge of Spring Tool Suite (STS)/Eclipse
- Basic familiarity with GemFire – attend the GemFire Introduction course

# Mobile Phones

- Please turn to silent mode, if possible!
  - Vibrate mode may still cause an interruption
- If you need to take a call, please just slip out quietly

# Course Agenda: Day 1

- GemFire Architecture Review
- Configuring the Server in Client/Server
- Server Side Regions – Replicated & Partitioned
- Working with the Client



# Course Agenda: Day 2

- Querying for data
- Configuring Custom Partitioned Regions
- Cache Management
- Server-side Event Handling



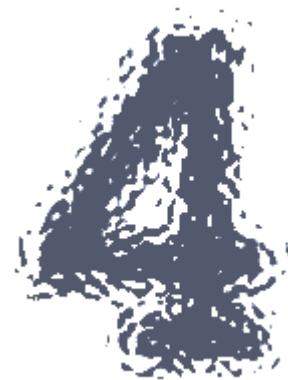
# Course Agenda: Day 3

- Client-side Event handling
- Data Serialization
- Transaction Management



# Course Agenda: Day 4

- Writing and Registering Functions
- Function execution
- Using Spring Data GemFire



# GemFire Terminology, Topologies & Use Cases

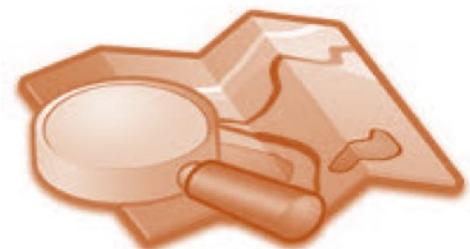
# Objectives

After completing this lesson, you should be able to:

- Define how GemFire functions as a IMDG
- Describe GemFire's contributions to the enterprise system
- Define the meaning of GemFire terminology
- Describe common GemFire topologies
- Describe common use cases for GemFire

# Lesson Road Map

- **GemFire IMDG**
- GemFire Terminology
- GemFire Common Topologies
- GemFire Use Cases



# GemFire - The Enterprise Data Grid

distributed memory based data management platform

artner - In Memory Data Grid (IMDG)

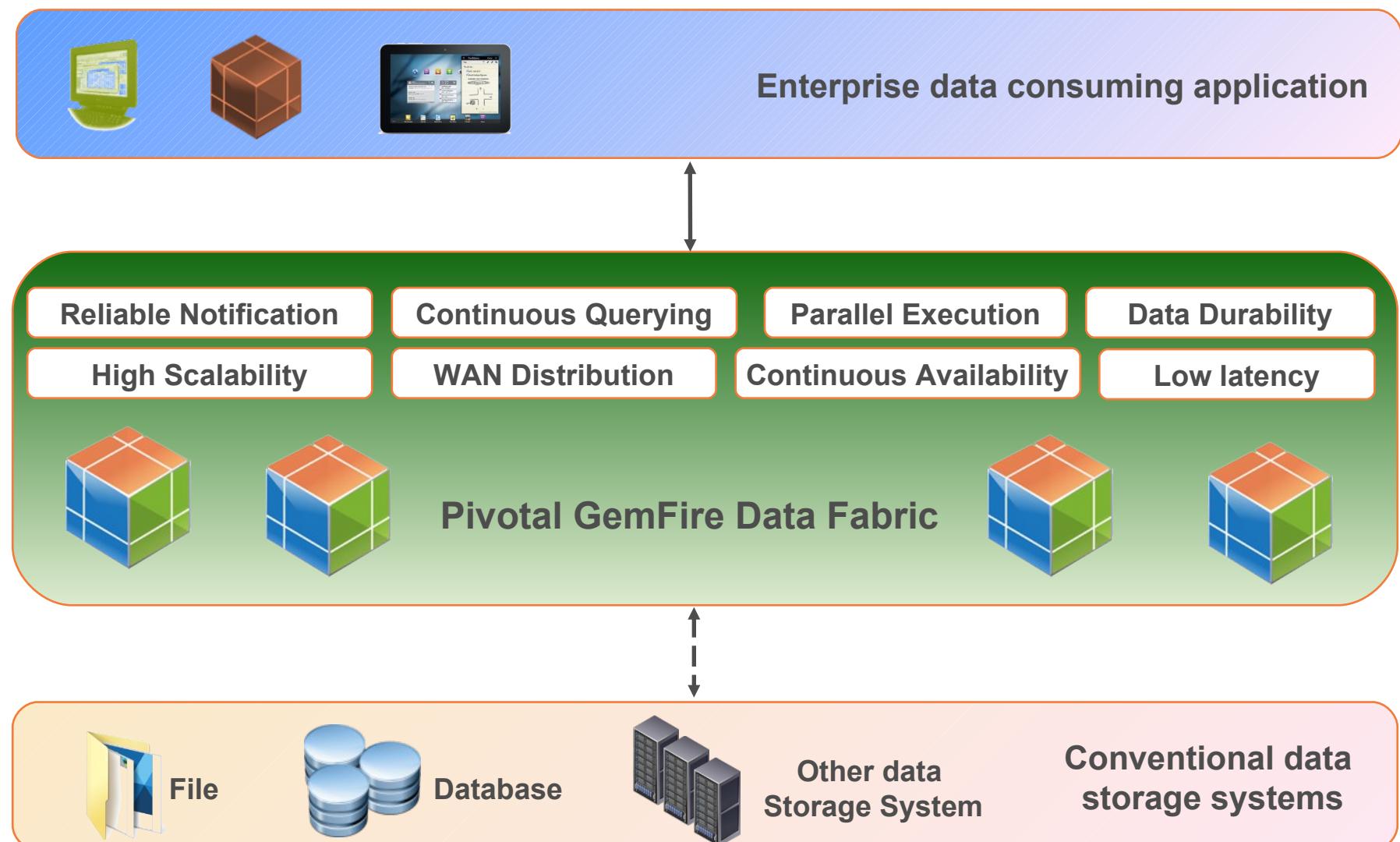
rovides continuous availability, high performance, and linear scalability for data intensive applications

lows for configurable data consistency

event driven data architecture



# GemFire – The Enterprise Data Grid



# Pivotal Big Data Suite

## Data Processing



Spring XD



Spark



Pivotal HD

## Advanced Analytics



Pivotal  
Greenplum  
Database



Pivotal  
HAWQ

## Apps at Scale



Pivotal  
GemFire



Redis



RabbitMQ



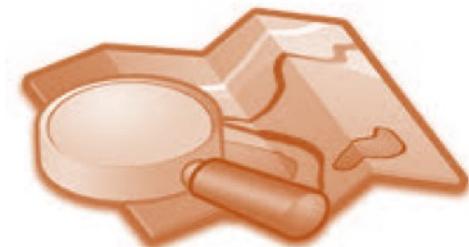
Pivotal  
Cloud Foundry



Pivotal Big Data Suite  
on Pivotal Cloud Foundry

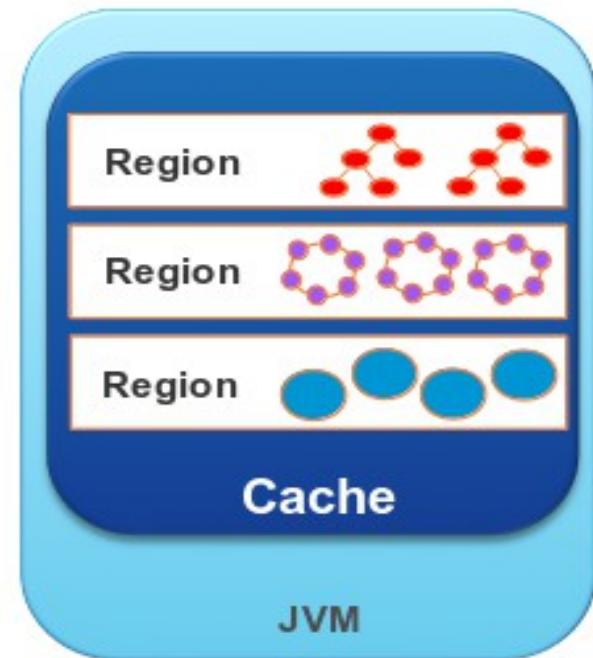
# Lesson Road Map

- GemFire IMDG
- **GemFire Terminology**
- GemFire Common Topologies
- GemFire Use Cases



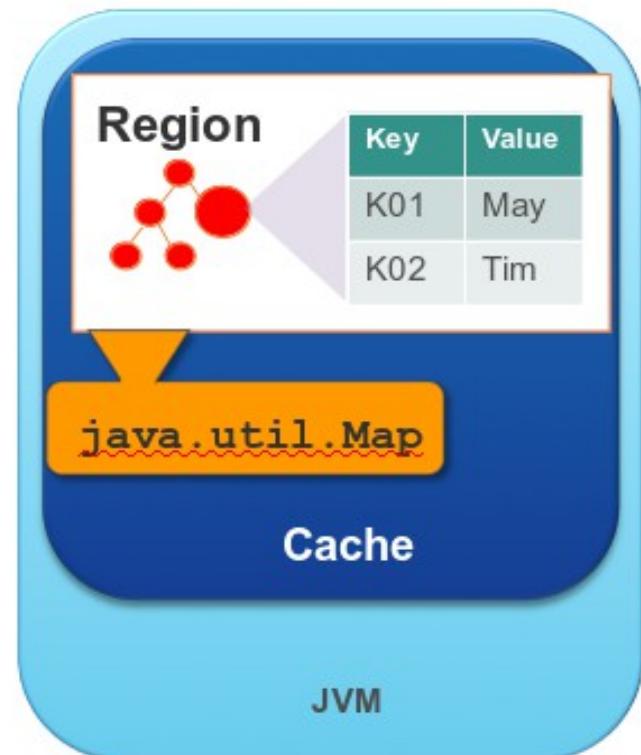
# Cache

- Cache provides in-memory data storage and management
- Cache configuration
  - XML declaration
  - gfsh
  - Java API
- Consist of Regions (distinct data sets)



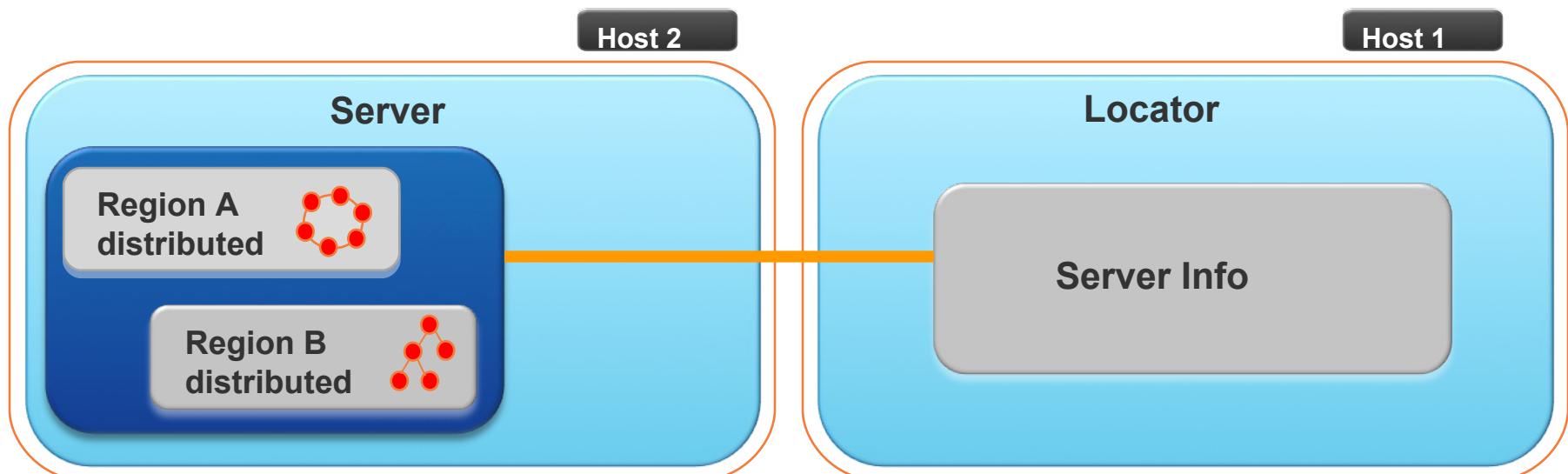
# Region

- Region groups cached objects into hierarchical namespaces
- Data objects are stored in a Region as key/value pairs
- Keys must be unique within a given region
- Manages a set of data via configuration



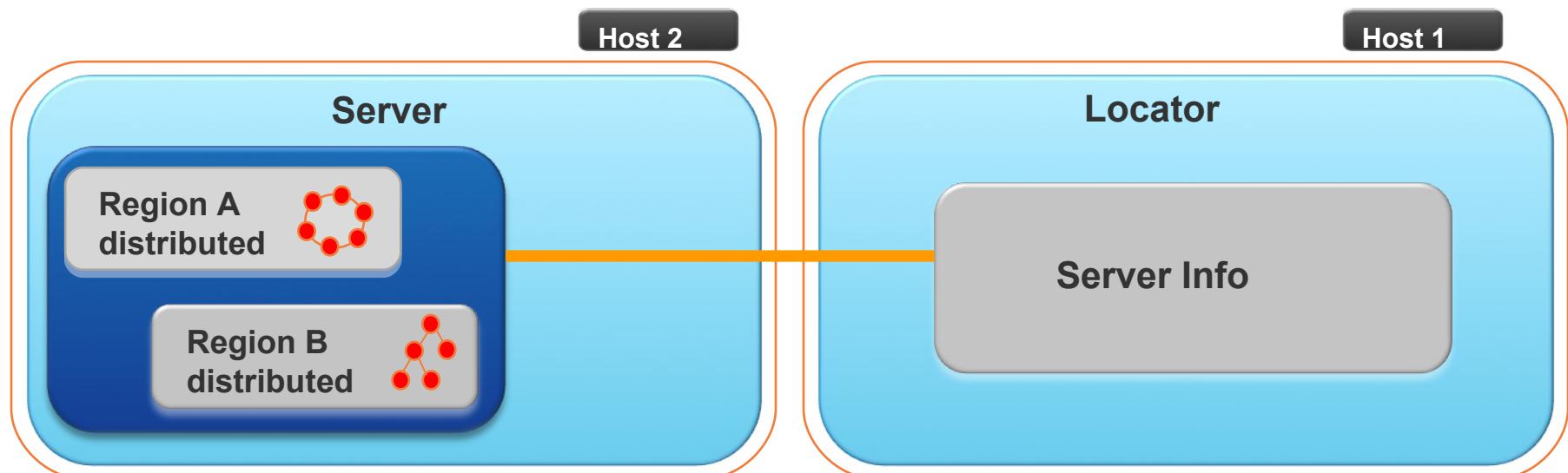
# Server

- Server host cache
- Cache made up of zero or more regions
- Process that services client requests
- Communicates status information to Locator
- Also called Cache Server



# Locator

- Are used for peer discovery by the servers
- Are used for server discovery by the clients
- Are specified in a Pool instance in the client applications
- Give dynamic server information to clients
- Provide server connection load balancing & fault tolerance

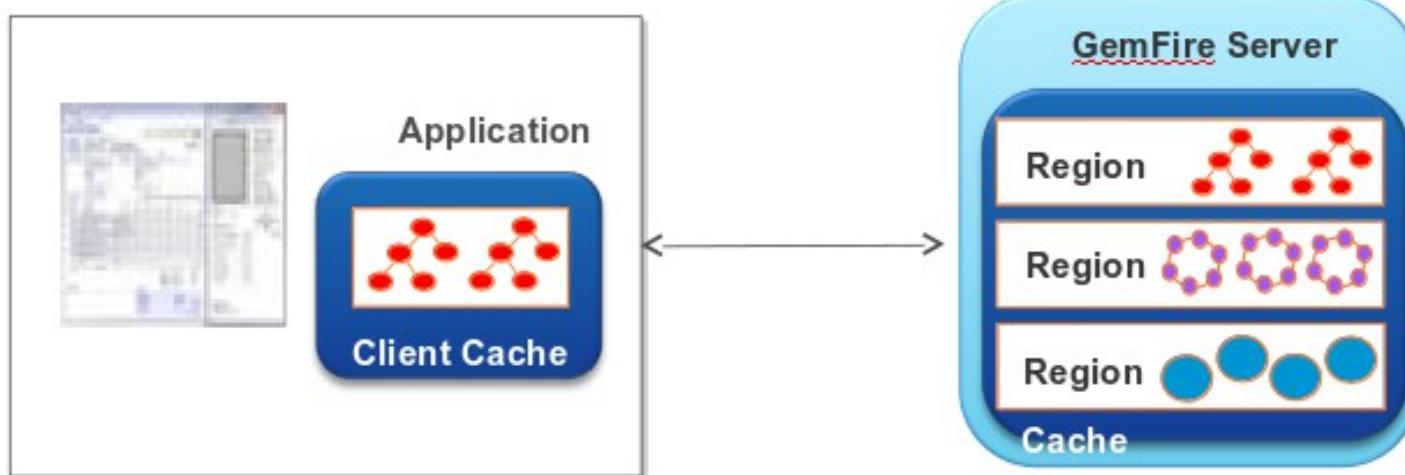


# Distributed System & Membership

- Member
  - A process that has created a connection, aka 'joined', to a GemFire distributed system
- Distributed System
  - One or more GemFire system members that have been configured to communicate cache events with each other, forming a single logical system
- Who is a member?
  - GemFire Servers
  - GemFire Locators
  - Clients are NOT members

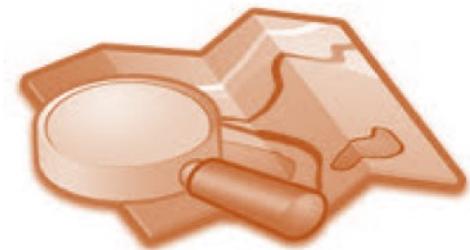
# Client Cache

- Client is connected to a set of GemFire servers via the locator
- Clients can access data on servers AND can optionally keep a local copy
- If the client registers for changes, the server notifies the client on change

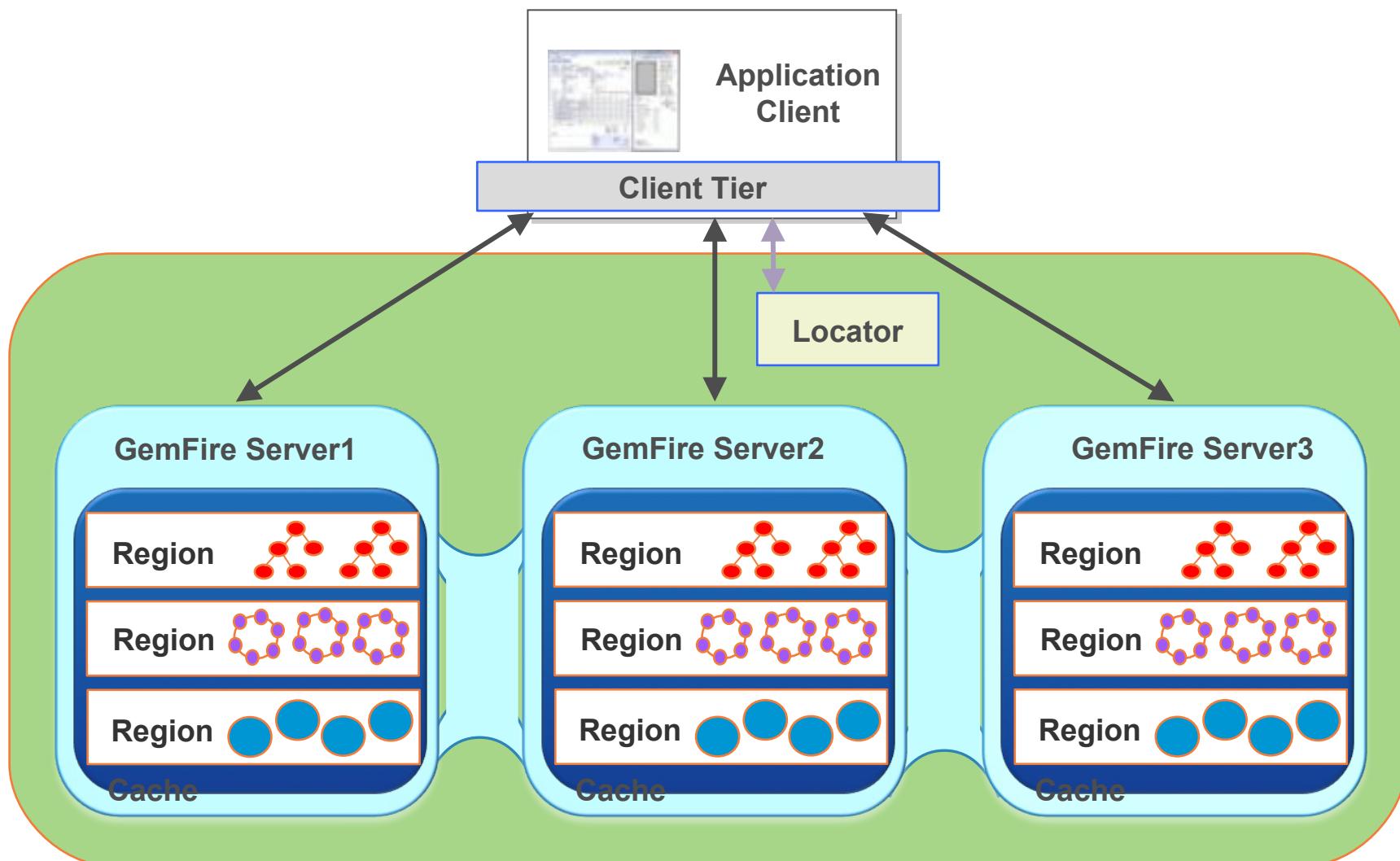


# Lesson Road Map

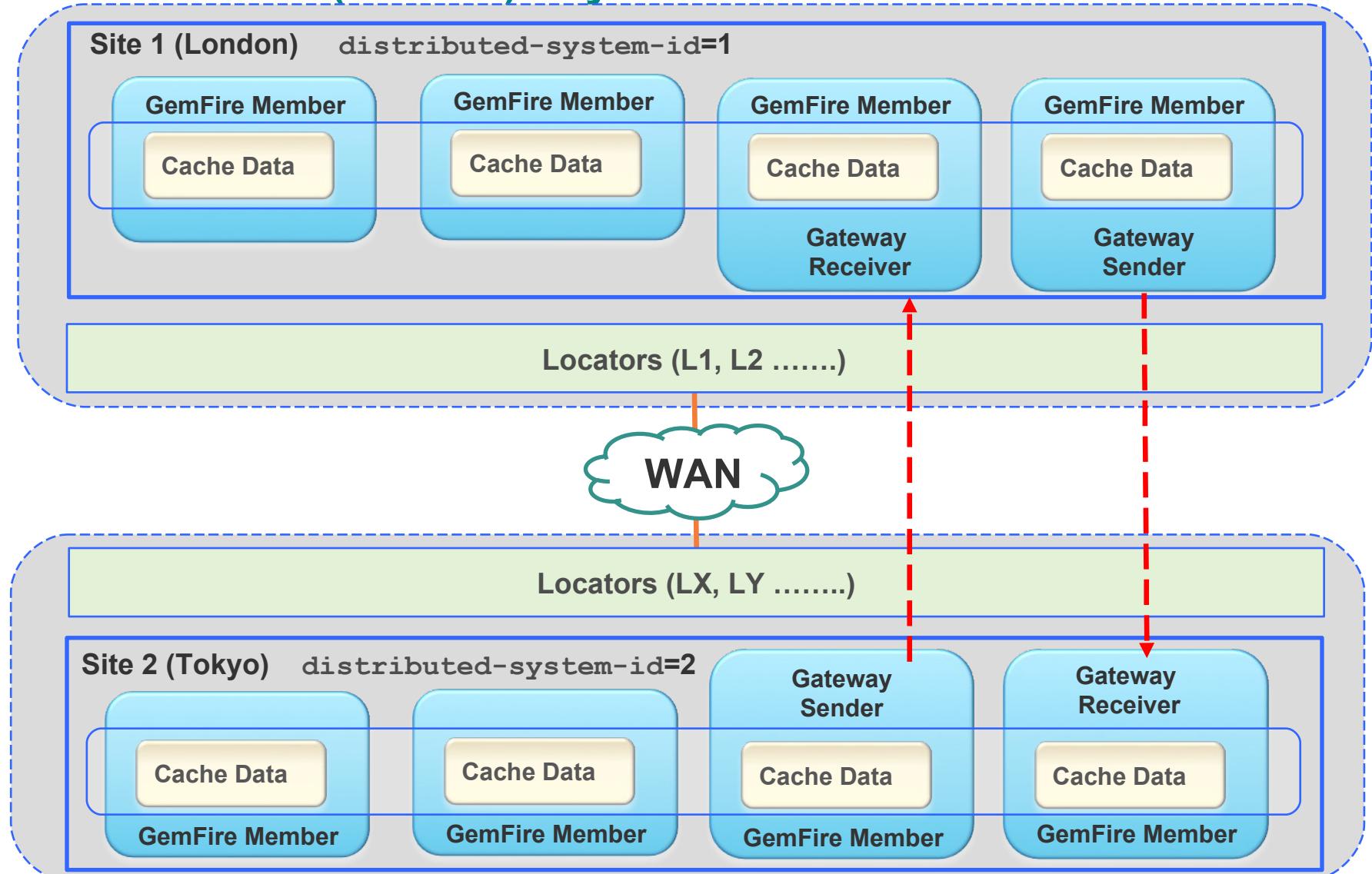
- GemFire IMDG
- GemFire Terminology
- **GemFire Common Topologies**
- GemFire Use Cases



# Client-Server Topology

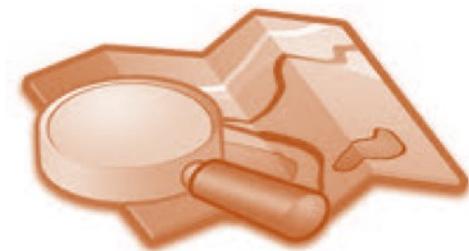


# Multi-site (WAN) System

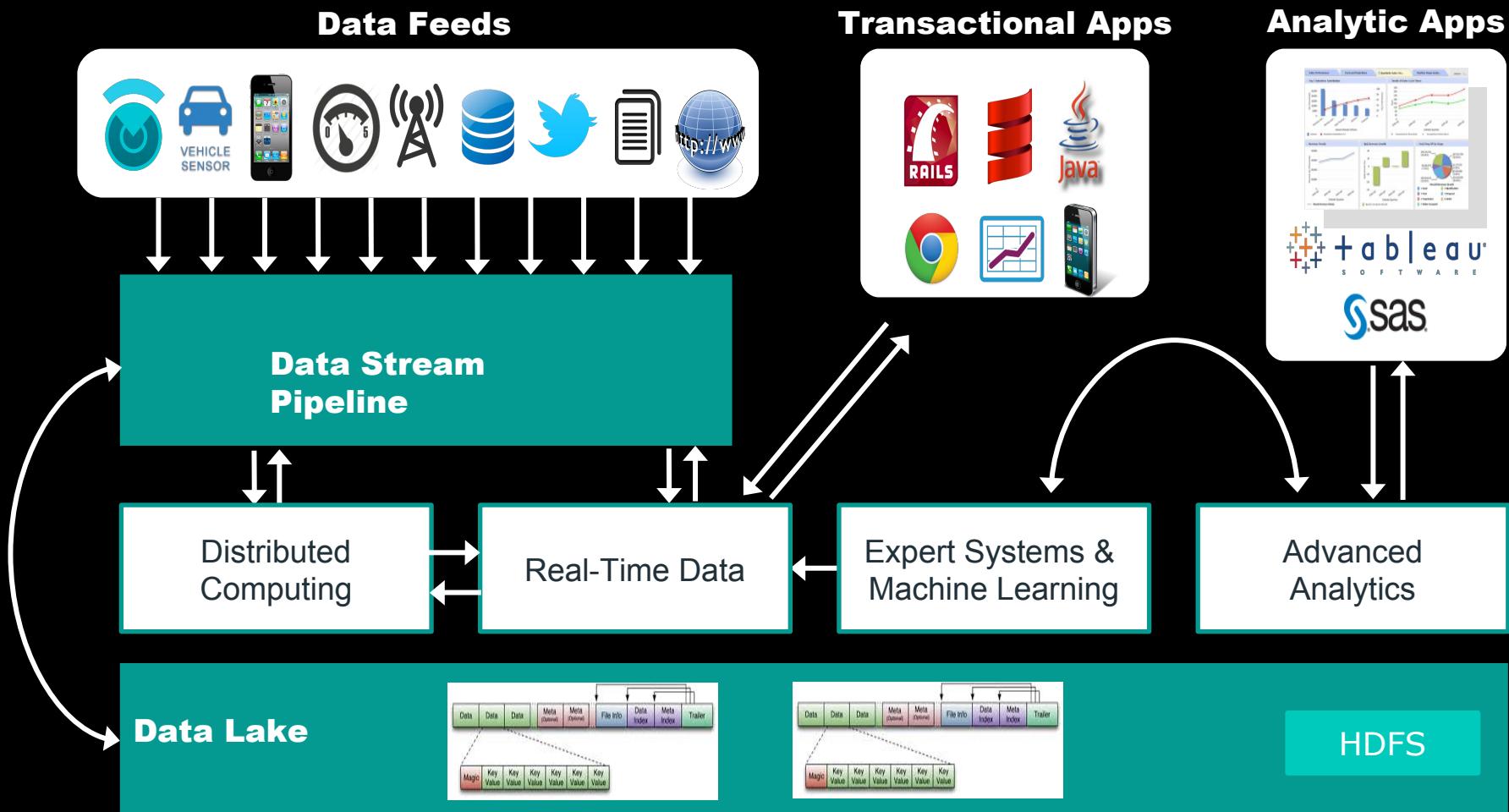


# Lesson Road Map

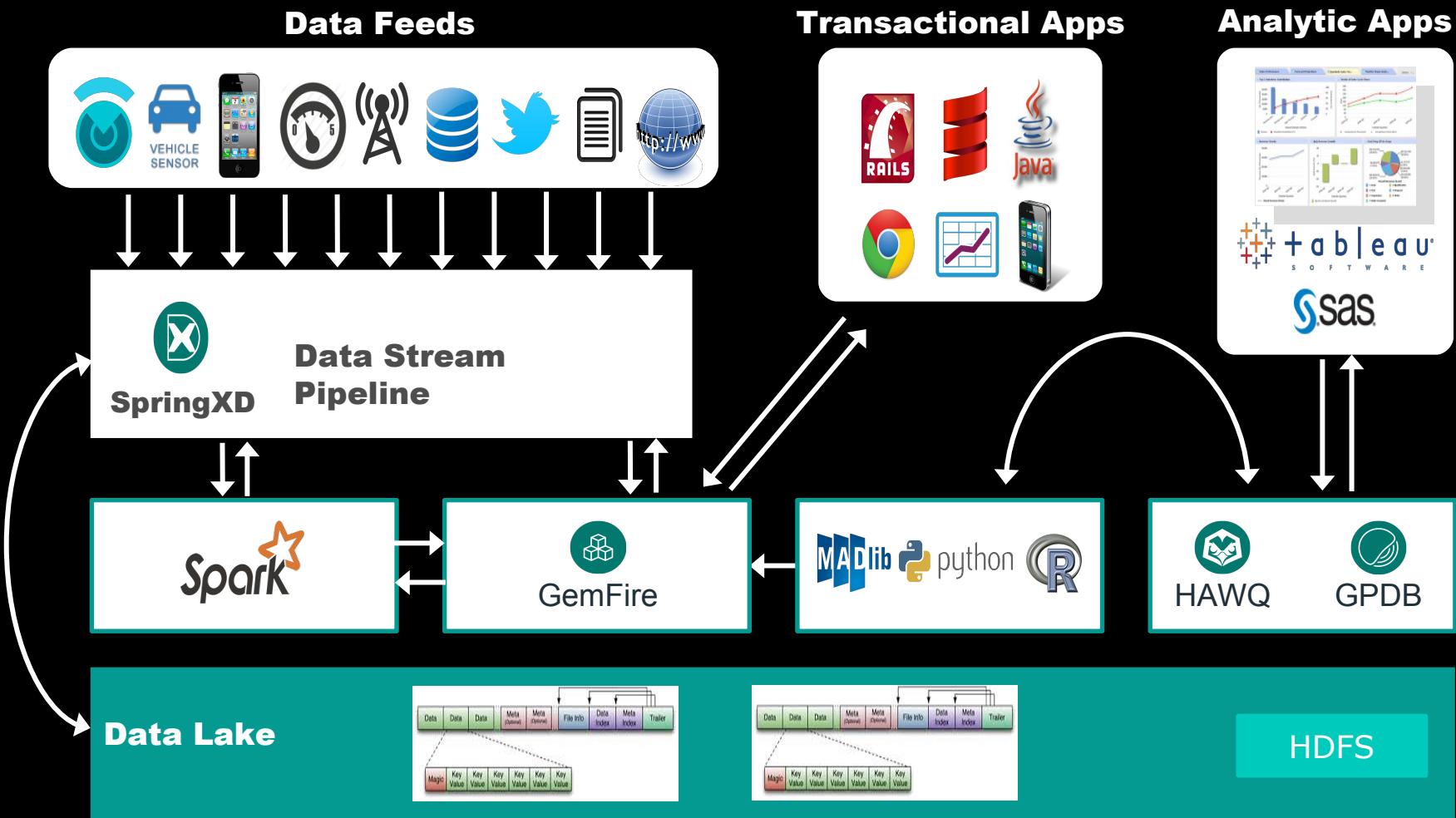
- GemFire IMDG
- GemFire Terminology
- GemFire Common Topologies
- **GemFire Use Cases**



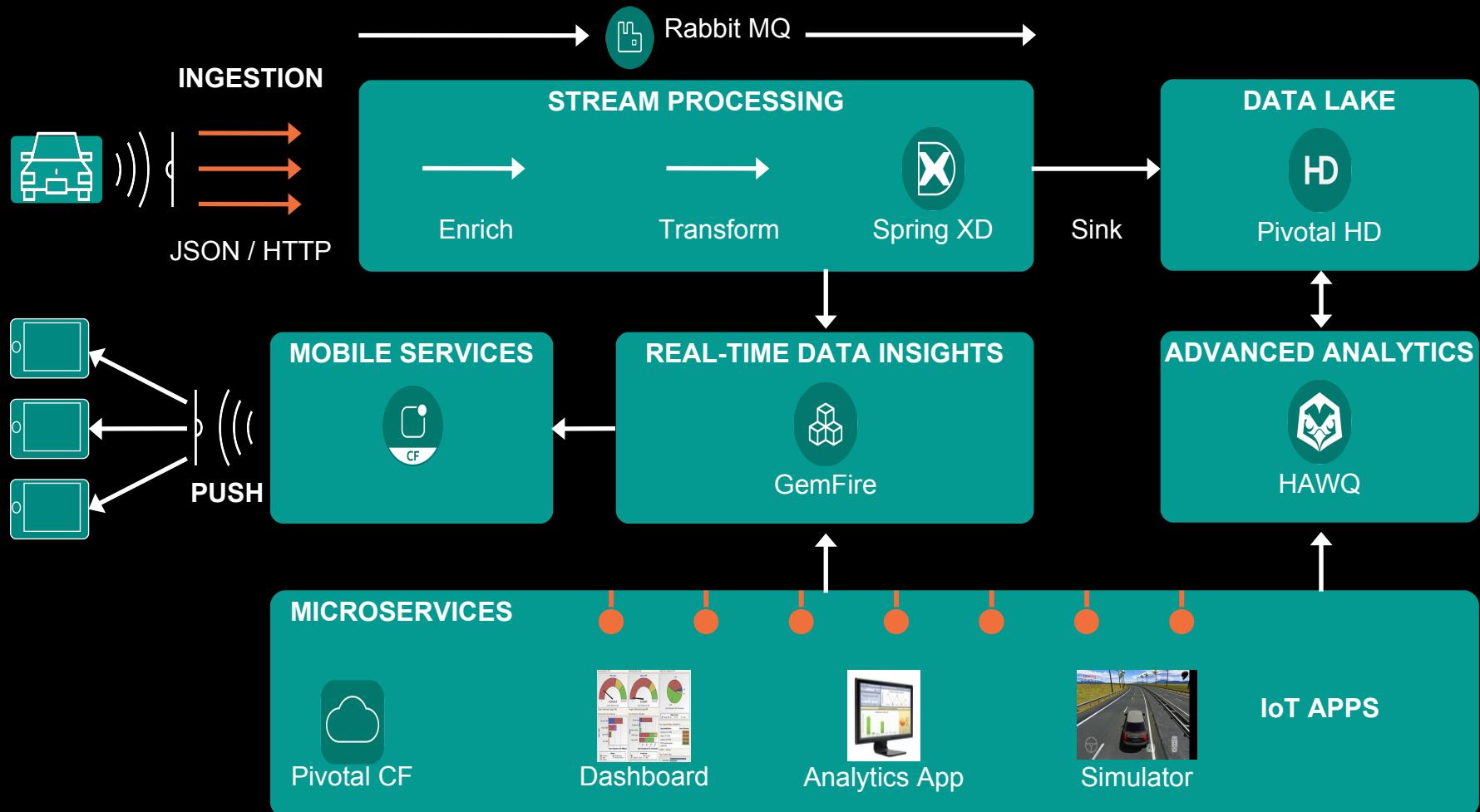
# Data Streaming Reference Architecture



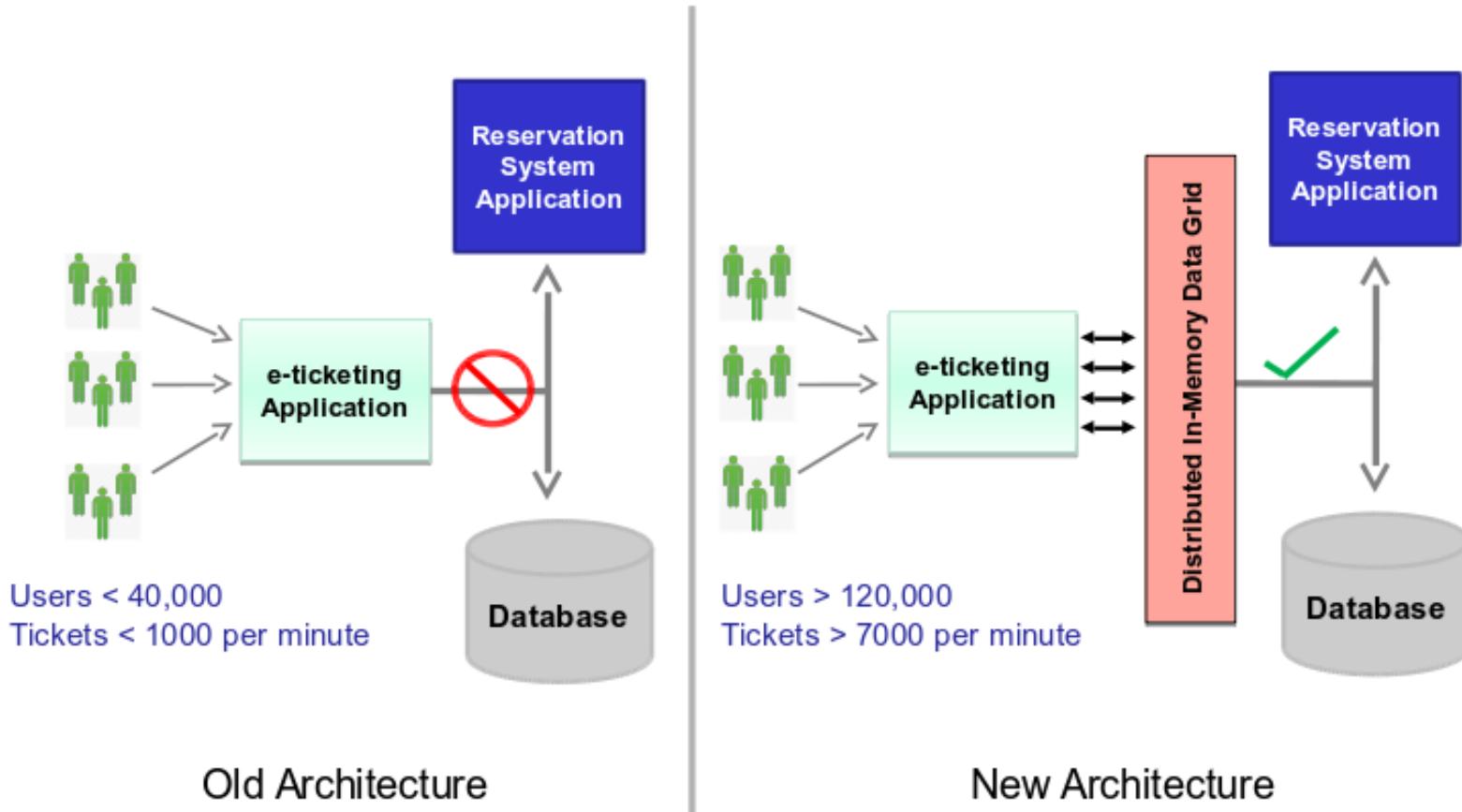
# Data Streaming Reference Architecture



# The Connected Car Architecture



# Use Case – Online Reservation System



# Use Case - Foreign Exchange Trading System

- Low-latency trade insertion
- Permanent archival of every trade
- Rapid, Event-based financial position calculation
- Distribution of updates globally
- Consistent global views of positions
- High Availability
- Disaster Recovery

# Review of Objectives

**You should be able to do the following:**

- Define how GemFire functions as a IMDG
- Describe the relationship between Caches and Regions
- Describe function of Cache Server and Locator as members of a Distributed System
- Describe GemFire Client-Server and Multi-Site WAN deployment topologies
- Describe common use cases for GemFire IMDG

# Lab

**In this lab, you will:**

- 1.**Install GemFire and the lab environment
- 2.**Finalize lab environment configuration
- 3.**Start servers and execute a test client

# Pivotal

A NEW PLATFORM FOR A NEW ERA

# Configuring the Server Side of Client/Server

# Learner Objectives

After this lesson, you should be able to do the following:

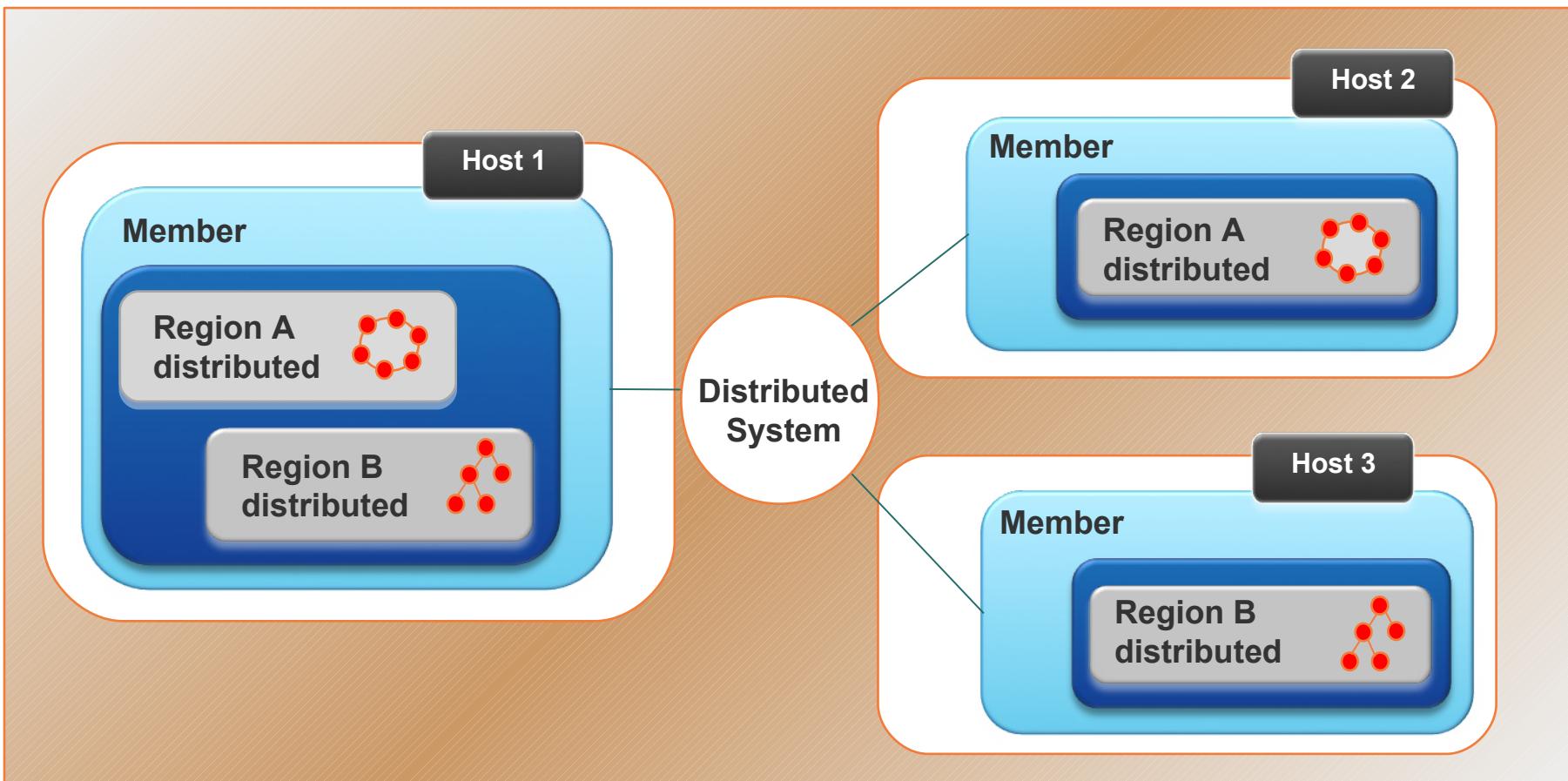
- Describe the components of GemFire distributed systems
- Describe the GemFire client/server architecture
- Explain logical grouping of servers/members
- Describe client/server data flow
- Describe working with gfsh (gee-fish)

# Lesson Road Map

- **GemFire Distributed Systems**
- Client/Server Architecture
- Working with gfsh

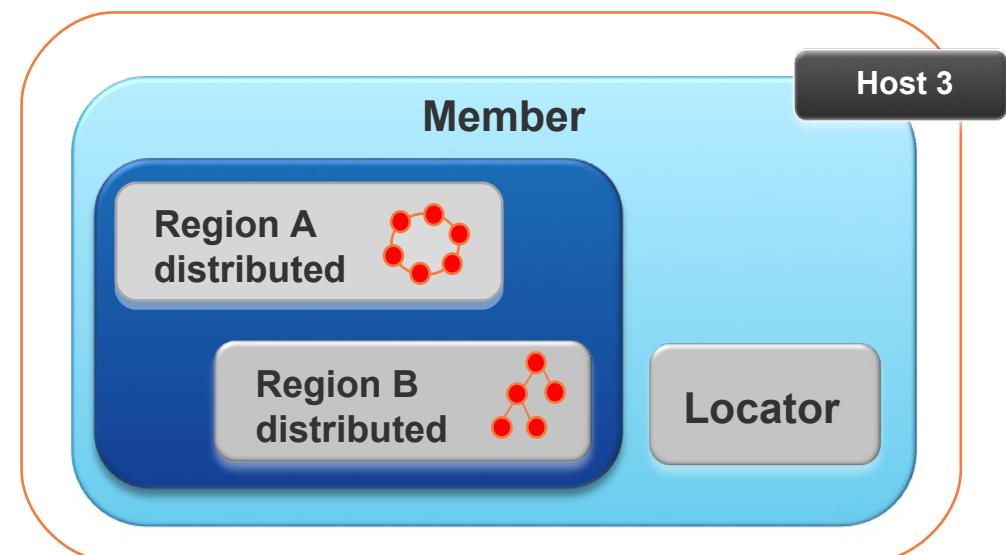


# GemFire Distributed System



# Distributed System: Locators

- Locators:
  - Used for peer discovery by the servers
  - Used for server discovery by the clients
  - Specified in a *Pool* instance in client applications
  - Provide dynamic server information to clients
  - Enable server connection load rebalancing and fault tolerance

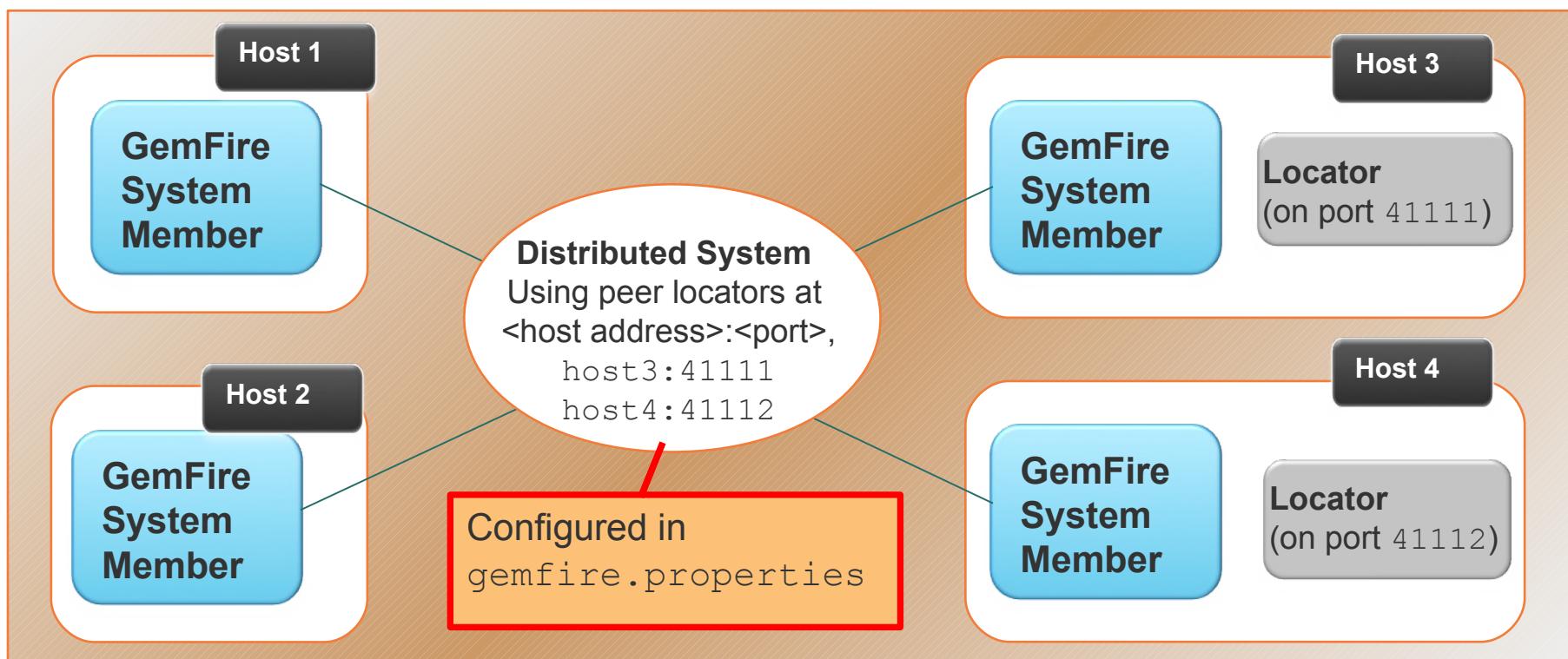


# Distributed System: Member Discovery

- The discovery process:
  - Is performed by the locator process
  - Uses TCP for discovery
  - Updates membership dynamically

# Distributed System: Locators for Peers

- Locators can be configured in two main ways:
  - In gemfire.properties file
  - Using gfsh argument (--locators=...)



# Architecture: Locator Configuration

- Locator port specified on startup

```
gfsh> start locator --name=locator1 --port=41111
```

- Cache-servers must use *same* peer discovery port
  - Configured in **gemfire.properties** configuration file

```
mcast-port=0      # Multicast no longer used  
locators=host3[41111]
```

- Use multiple locators in a cluster for high availability & failover

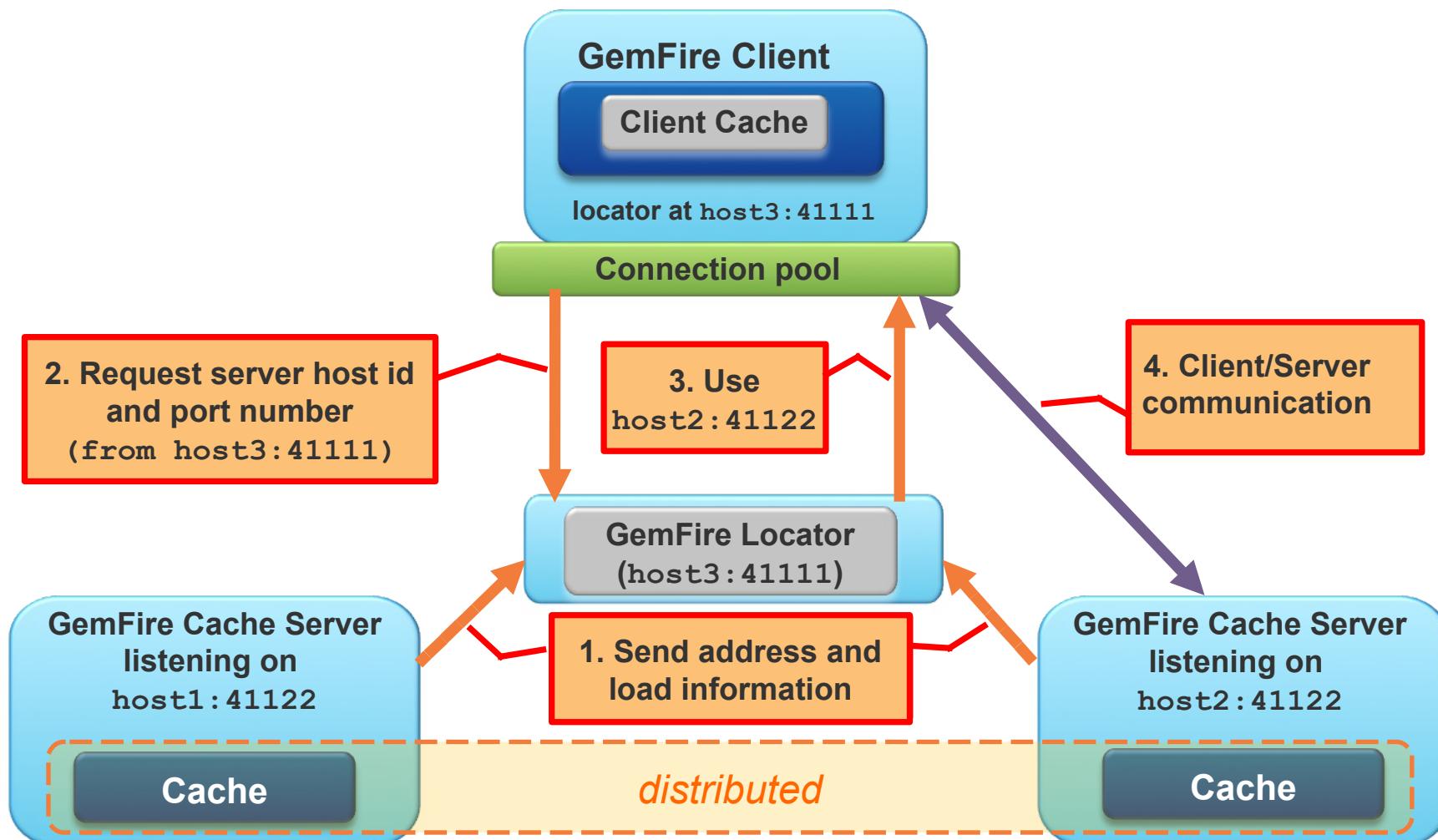
```
mcast-port=0      # Multicast no longer used  
locators=host3[41111],host4[41112]
```

# Lesson Road Map

- **GemFire Distributed Systems**
- **Client/Server Architecture**
- **Working with gfsh**



# Review: Client – Locator – Server Architecture



# Architecture: Server Configuration

- Servers may specify their port number
  - Using gfsh or in their configuration XML

```
gfsh> start server --name=server1  
      --properties-file=gemfire.properties --server-port=41122
```

- Or in their XML configuration

The diagram illustrates the configuration of a server port. On the left, a grey box contains the gfsh command to start a server named 'server1' with properties file 'gemfire.properties' and server port '41122'. On the right, a larger grey box shows the equivalent XML configuration in 'serverCache.xml'. It includes the 'cache-xml-file' attribute, a comment indicating 'Multicast no longer used', the 'locators' attribute, and an ellipsis. A callout arrow points from the 'server-port' attribute in the XML to the 'server-port' value in the gfsh command.

```
cache-xml-file=serverCache.xml  
mcast-port=0      # Multicast no longer used  
locators=host3[41111]  
...  
  
<cache>  
  <cache-server server-port="40404"/>  
  ...  
</cache>
```

# Configuring Cluster to use Locators

- Multi-cast is enabled by default – enabled by the mcast-port property
- You will almost always disable this by setting mcast-port=0 and then specifying locators
  - Set via properties file

```
locators=host3[41111]
mcast-port=0      # Multicast no longer used
```

- Set via gfsh

```
gfsh> start server --name=server1 --locators=host3[41111]
--mcast-port=0 --server-port=41122
```

# Lesson Road Map

- **GemFire Distributed Systems**
- **Client/Server Architecture**
- **Working with gfsh**



# gfsh – GemFire Shell

- A command line tool that ships with GemFire
  - Interface similar to unix-like shell commands
  - Allows you to view Region information, like bucket distribution in partitioned Regions
  - Can be used to browse and edit data in GemFire
  - Added by 6.6, redone in 7.0



# gfsh: Starting / Stopping Locators

- Started individually through **gfsh** command-line utility
  - Use own **gemfire.properties** files for configuration

```
gfsh> start locator  
      --name=locator1  
      --port=41111  
      --dir=locator  
      --properties-file=gemfire.properties  
      --initial-heap=50m  
      --max-heap=50m
```



Created if necessary,  
where locator puts its  
logs and other files

- Stopping a locator:

```
gfsh> stop locator --name=locator1
```

# gfsh: Starting / Stopping Servers

- Started individually through gfsh command-line utility
  - Use own **gemfire.properties** files for configuration

```
gfsh> start server  
  --name=server1  
  --locators=host3[41111]  
  --server-port=41122  
  --dir=server1  
  --properties-file=gemfire.properties  
  --initial-heap=50m  
  --max-heap=50m
```

Set to zero, locator will choose port dynamically for the server

- Stopping a server:

```
gfsh> stop server --name=server1
```

# gfsh – Misc Commands

- Connect to the JMX Agent via a locator
  - Once connected, full access to gfsh functionality

```
gfsh> connect --locator=localhost[41111]
```

- List all regions

```
gfsh> list regions
```

- List the partition and bucket information

```
gfsh> show metrics --region=/Customer  
          --member=server1  
          --categories=partition
```

# What can I do with gfsh?

## Start/Stop commands

- connect
- disconnect
- start (locator, server)
- start (gateway-receiver/sender)
- start (pulse, jconsole, etc)
- stop (locator, server)
- stop (gateway-receiver/sender)

## Informational commands

- debug
- describe (member, region,...)
- echo
- help / hint / history
- list (members, regions, ...)
- netstat
- show (metrics, log, ...)
- status
- version

## General Maintenance commands

- alter (disk-store, region, etc)
- change loglevel
- clear defined indexes
- close
- configure
- create (index, region,...)
- define index
- deploy
- destroy
- encrypt password
- export
- pause/resume gateway-sender
- gc
- import
- pdx rename
- run
- set variable
- shutdown
- sleep
- undeploy

## Region/Entry commands

- execute function
- get
- locate entry
- put
- query
- rebalance
- remove

## DiskStore Operations

- backup disk-store
- compact
- revoke missing-disk-store
- upgrade offline-disk-store
- validate offline-disk-store

# Lab

In this lab, you will

- Configure and start a locator for member discovery
- Configure and start a GemFire server
- Gain familiarity with `gfsh` commands

# Review of Learner Objectives

**You should be able to do the following:**

- Describe the components of GemFire distributed systems
- Describe the GemFire client/server architecture
- Explain grouping of servers
- Describe client/server data flow
- Describe working with gfsh utility

# Pivotal

BUILT FOR THE SPEED OF BUSINESS

# Replicated and Partitioned Regions

# Learner Objectives

After this lesson, you should be able to do the following:

- Describe and configure partitioned regions
- In partitioned regions, describe how data entries are placed in buckets based on hashing
- Describe how partitioned regions behave in the case of a hardware failure
- Describe and configure replicated regions

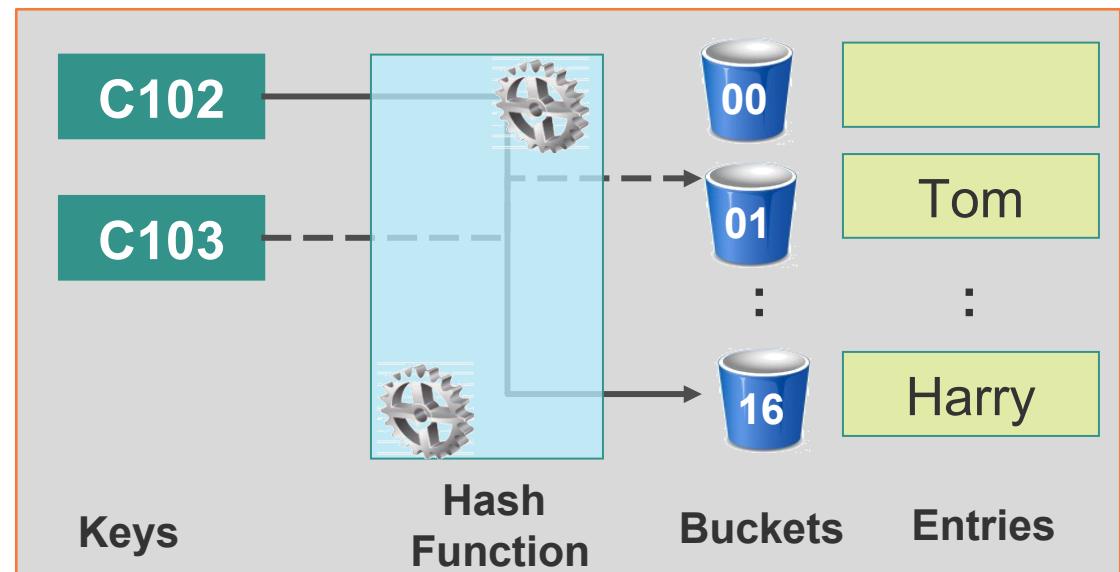
# Lesson Road Map

- Cache Basics
- Partitioned Regions
- Partitioned Region Recovery
- Caveats to Partitioned Regions
- GemFire Replication



# Java HashMaps and Buckets

- Java HashMaps are structures created for fast access to Collections of Objects
- Buckets
  - Object keys are “hashed”, using their `hashCode()` method and placed into a “bucket”
  - `key.hashCode() mod number_of_Buckets = bucket`
- Good bucket distribution can be achieved by having a prime number as the number of buckets

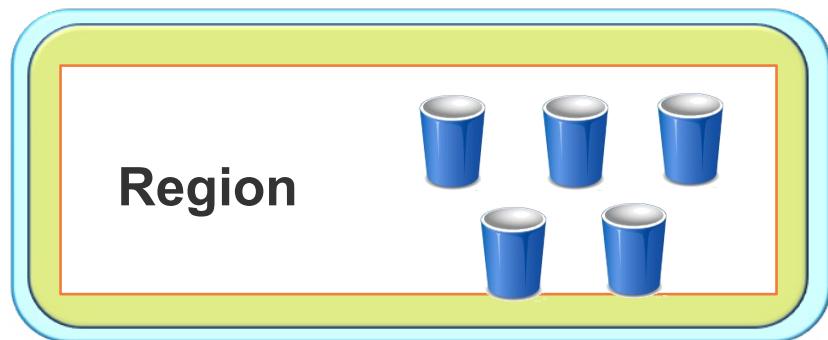


# Hashing

- Some function or algorithm to map object data to some representative integer value
- This value, or hash code, can then be used as a way to narrow down our search when looking for an item in a Map
  - Multiple values are expected to have the same “hash”
  - A good hashCode should provide uniform distribution to reduce number of multiple keys in a given bucket
  - Even distribution across the platform is the goal
  - Bucket entries are traversed in order

# Regions and Buckets

- Regions implement `java.util.HashMap`
- Region uses the same concept of buckets to sort Keys
- Default number of buckets is 113 (prime)
- In Partitioned Regions, buckets are distributed across nodes.
  - Key value pairs are routed to bucket based on `hashCode()`
- Custom partitioning can be achieved via `PartitionResolvers`
  - Keep all related data together within the same JVM



# Lesson Road Map

- Cache Basics
- **Partitioned Regions**
- Partitioned Region Recovery
- Caveats to Partitioned Regions
- GemFire Replication



# Partitioned Regions

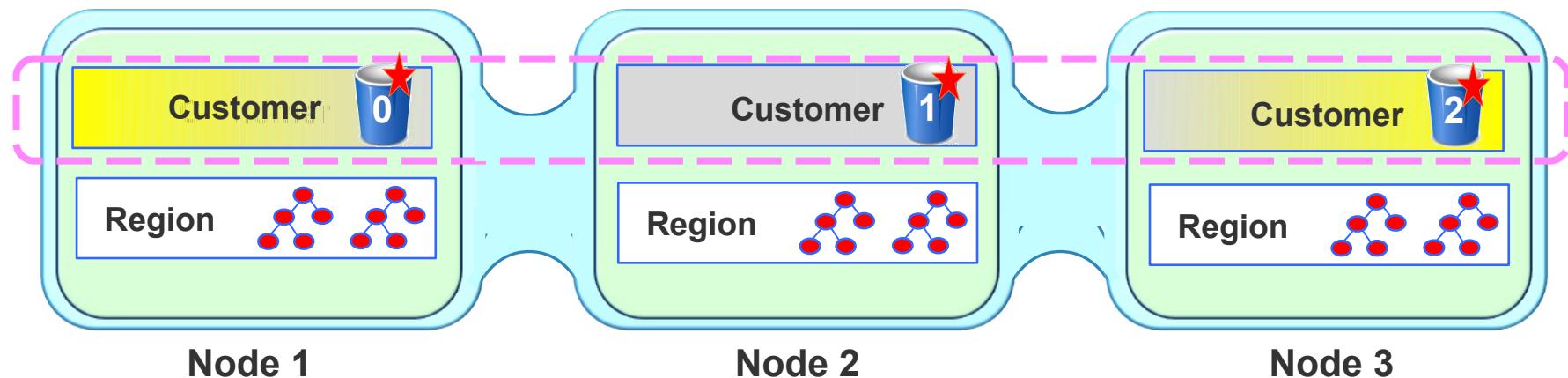
- A Region that is not contained in a single member.
- The data set spans multiple members
- A Region bucket set is partitioned across nodes in the distributed system.
  - Each member contains a subset of primary of buckets for a partition region
  - Each member may also contain redundant copies of buckets
- Accommodates data sets that are too large to hold on a single member
- Best suited for:
  - Large data sets (100's of GB)
  - Data sets that will need to scale.
  - Write heavy data sets.

# When do I need partitioning?

- Large data sets
  - Store data sets that are too large to fit in a single VM, single cache member
  - 1:Many relationships
  - Suitable for 100's of GB
- High availability
  - Buckets in partitioned regions can be replicated across nodes
  - Redundancy reduces/removes data loss on cache member failure
- Scalability
  - Add more nodes, at runtime, to expand data capacity
  - Increased processing capability using Function Execution by routing process to data location
  - Near linear scalability

# Basic Partitioning

```
<region name="Customer">  
  <region-attributes refid="PARTITION">  
  </region-attributes>  
</region>
```

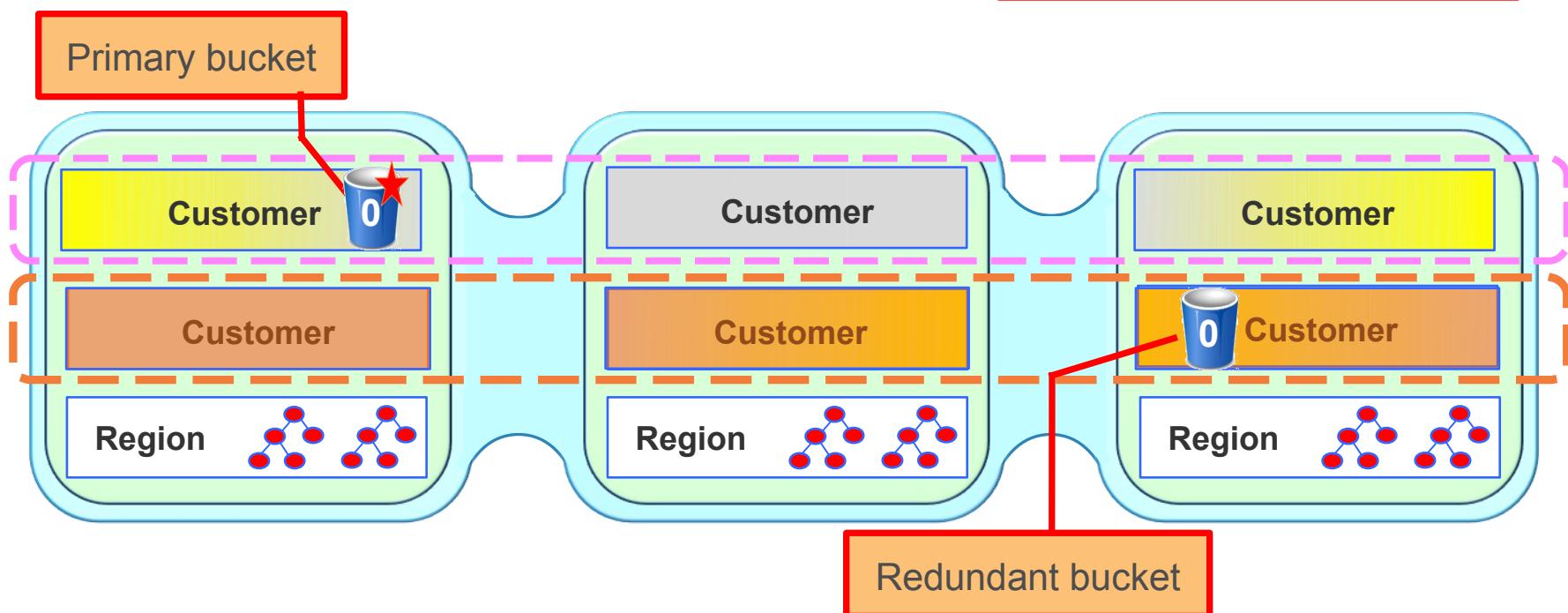


*All updates are synchronous, and consistency is guaranteed*

# Partition Redundancy

```
<region name="Customer">  
  <region-attributes refid="PARTITION">  
    <partition-attributes redundant-copies="1"/>  
  </region-attributes>  
</region>
```

Number of redundant copies



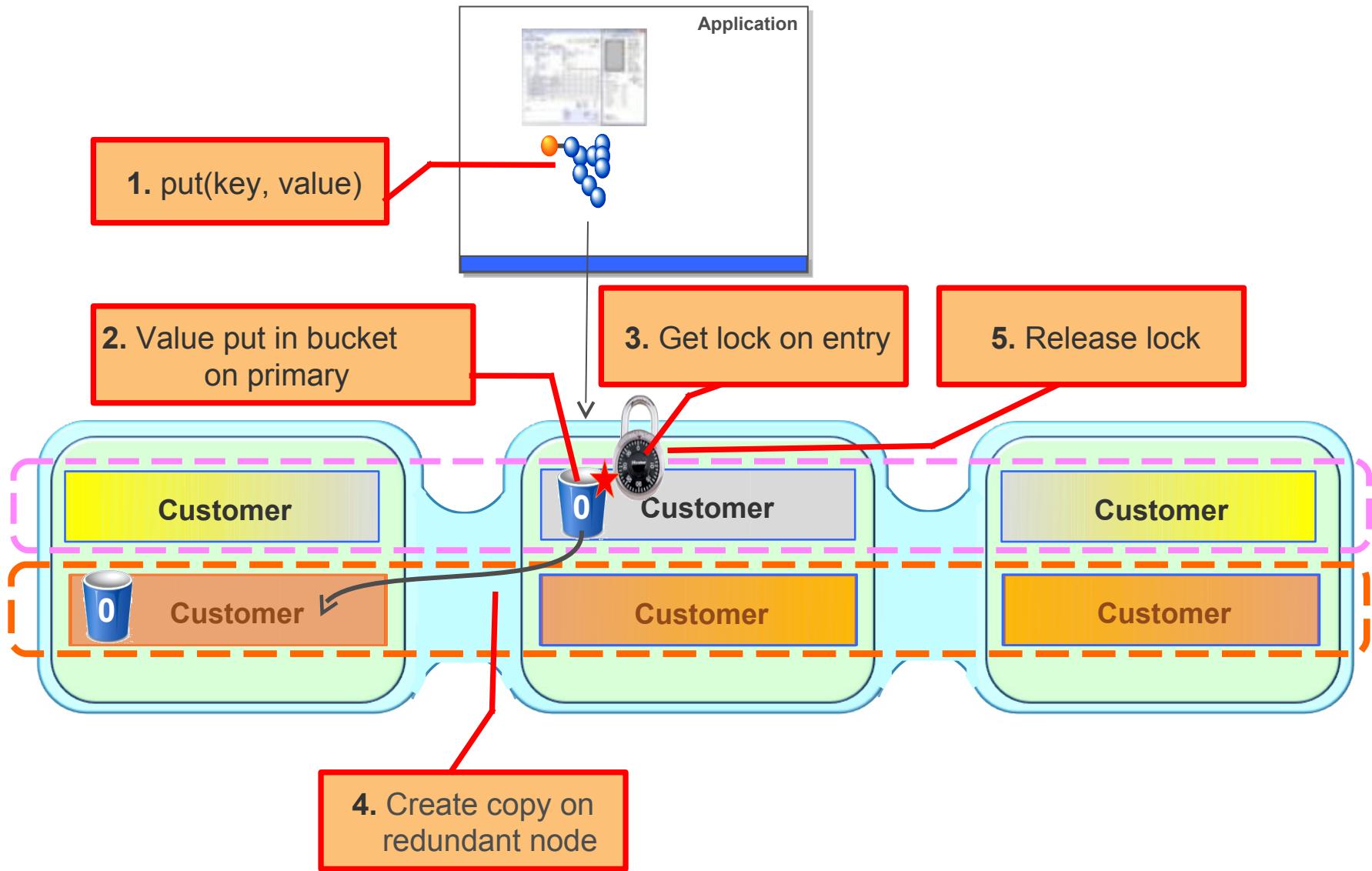
# Partition Redundancy - Java

```
RegionFactory rf =
    cache.createRegionFactory(RegionShortcut.PARTITION);

PartitionAttributesFactory paf = new PartitionAttributesFactory()
    .setTotalNumBuckets(7)
    .setRedundantCopies(1)
    .setStartupRecoveryDelay(5000);

rf.setPartitionAttributes(paf.create());
Region custRegion = rf.create("Customer");
```

# Putting a New Entry in a Partitioned Region



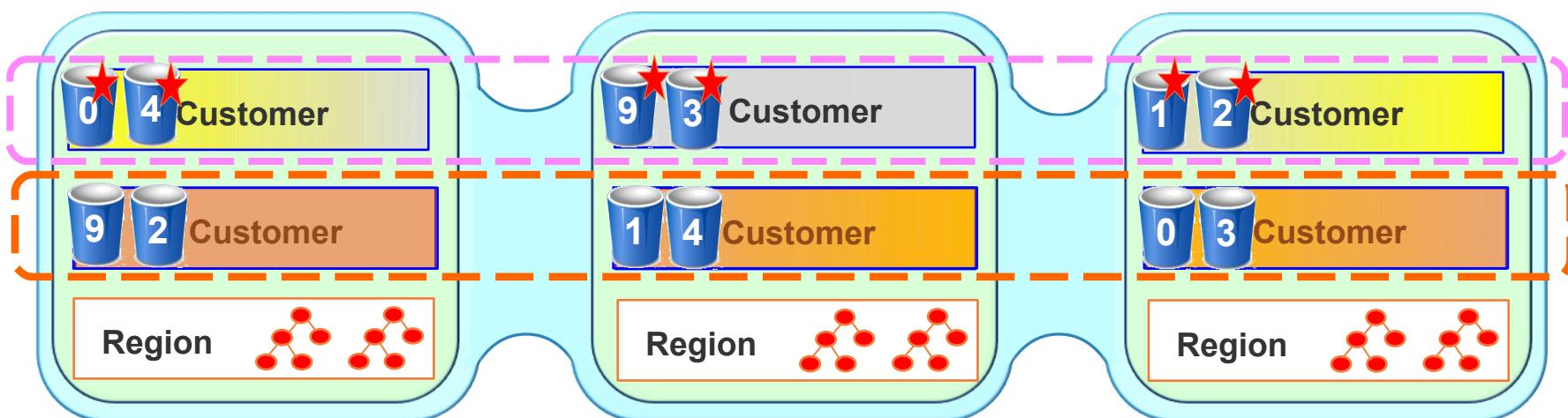
# Lesson Road Map

- Cache Basics
- Partitioned Regions
- **Partitioned Region Recovery**
- Caveats to Partitioned Regions
- GemFire Replication



# Bucket Distribution

- Buckets are made redundant across the nodes in the distributed system
- Buckets are allocated across nodes to ensure balance

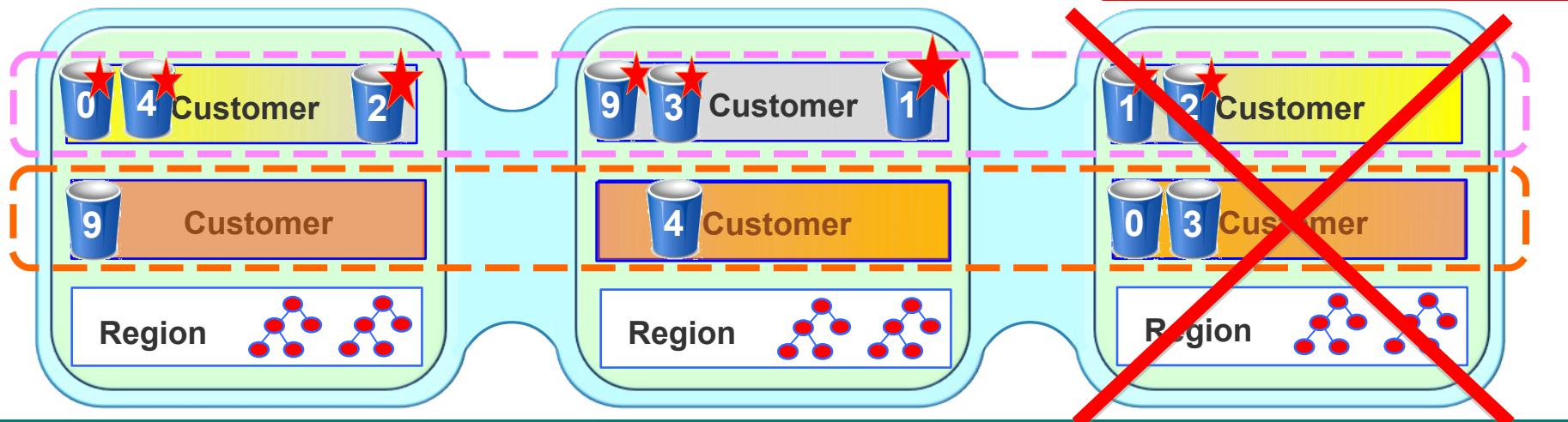


# Bucket Promotion on Failure

- If a node fails, one redundant copy of buckets will be instantly promoted to primary copies

```
<region name="Customer">  
    <region-attributes refid="PARTITION">  
        <partition-attributes redundant-copies="1"  
            total-num-buckets="113"  
            recovery-delay="50000"/>  
    </region-attributes>  
</region>
```

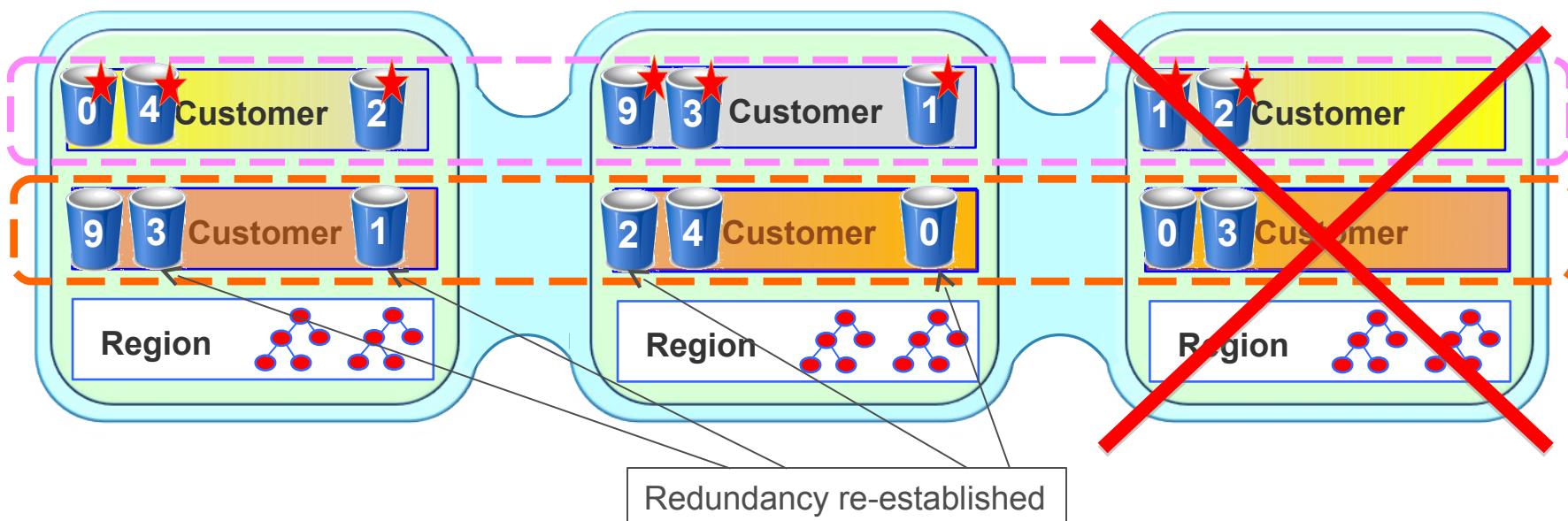
Default: -1 (don't recover)



Pivotal™

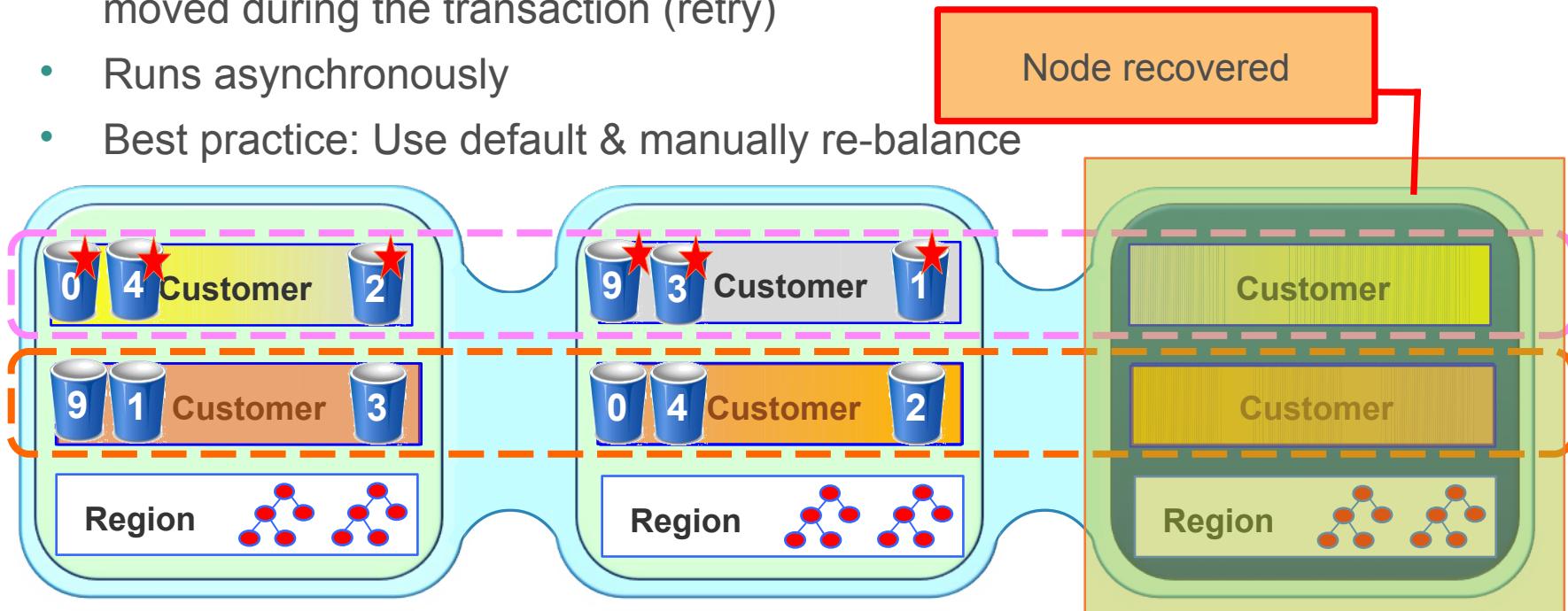
# Bucket Redundancy after Recovery Delay

- Use recovery-delay to allow time to restart the failed node, without incurring the work of the recovery
- Network latency can cause a node to appear to have failed
- If a node has failed in the eyes of the membership coordinator, it will not be allowed to bring its data back in
- It will be loaded with a fresh copy from remaining nodes



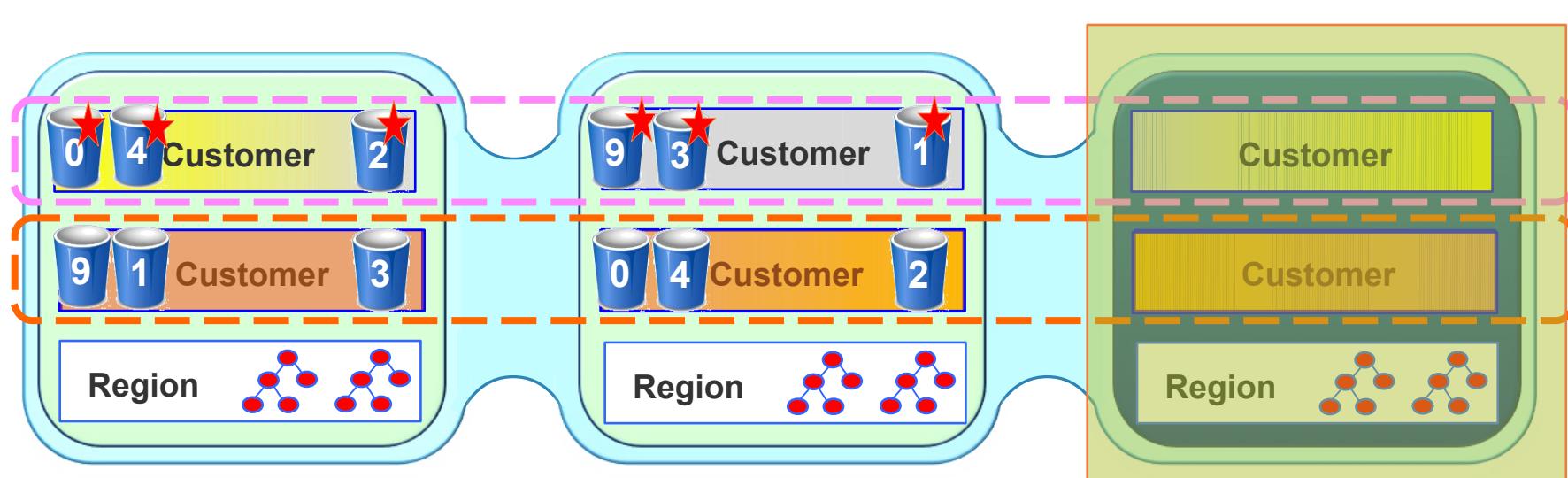
# Rebalancing

- For node start ups or adding capacity
- Buckets are rebalanced.
  - Do not rebalance during peak times
  - For large data sets do not use recovery delay – rebalance manually
- If transactions are running, they could receive an Exception if the data is moved during the transaction (retry)
- Runs asynchronously
- Best practice: Use default & manually re-balance



# Rebalancing – Command line

```
gfsh> rebalance [<attribute name> = <attribute value>]*
```



# Lesson Road Map

- Cache Basics
- Partitioned Regions
- Partitioned Region Recovery
- **Caveats to Partitioned Regions**
- GemFire Replication



# Caveats of Partitioned Regions

- Transaction data must be located on the same host
- Synchronous redundant copy updates, maintain data consistency
- The larger the ratio of buckets to data entries, the more evenly the load can be spread across the members
- Number of buckets  $20-50 * \text{number of servers}$  you expect to have for the region
- Guidelines:
  - More entries than buckets
  - Use a prime number
  - Look for even distribution – important when handling member failures

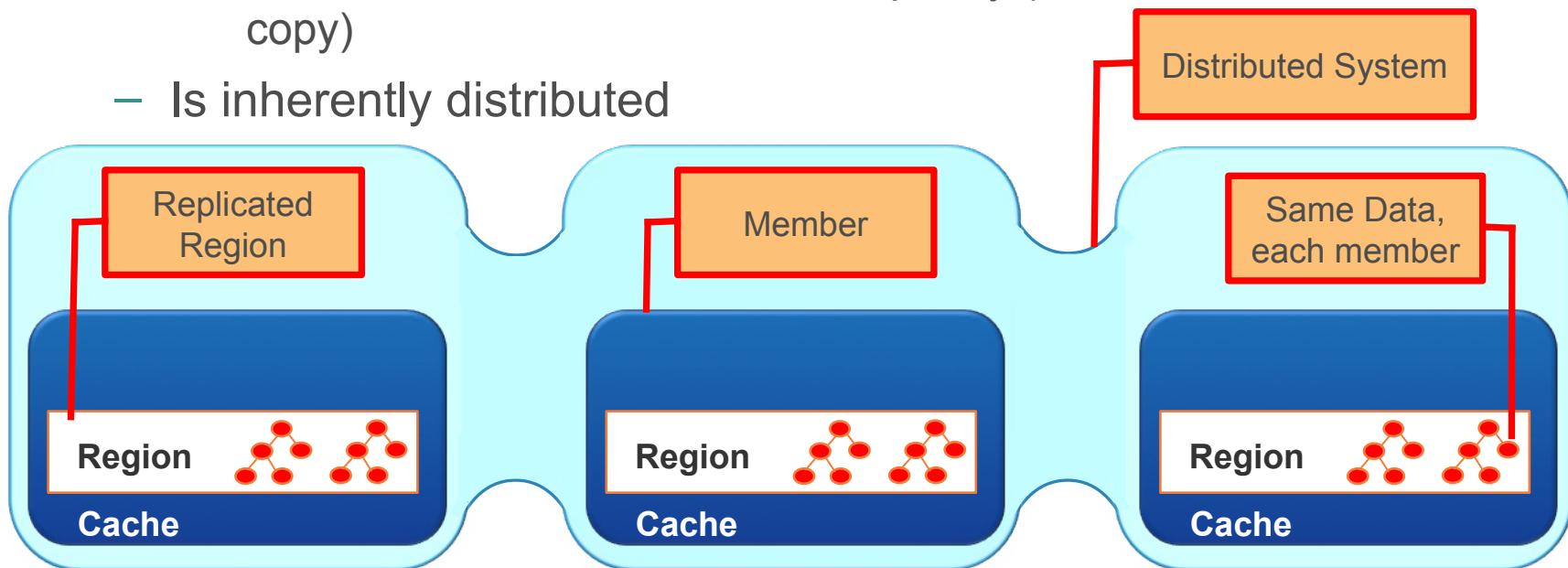
# Lesson Road Map

- Cache Basics
- Partitioned Regions
- Partitioned Region Recovery
- Caveats to Partitioned Regions
- **GemFire Replication**

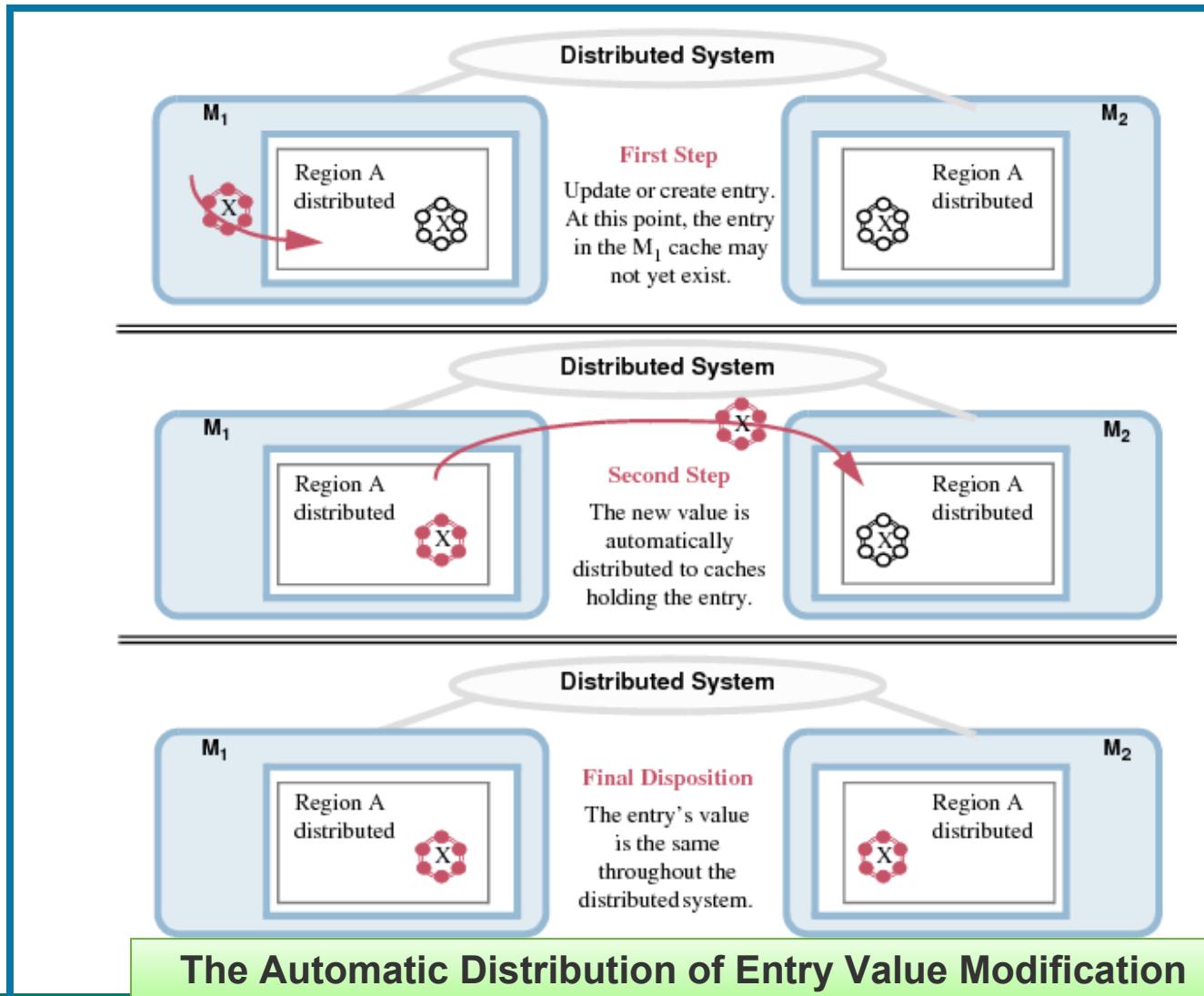


# What is a Replicated Region?

- The entire region is replicated across GemFire nodes, with all data
- Best suited for
  - Read heavy, **small** data set (reference or lookup data)
  - Data sets that can be entirely contained in a single member
  - Data that needs to be accessed quickly (each member holds a local copy)
- Is inherently distributed



# Replicated Regions: Overview



# Relevant Replicated Region Scopes

Scope	Semantics
<b>distributed-no-ack</b>	<ul style="list-style-type: none"><li>Updates are sent to all members, asynchronously, without acknowledgement, without locking</li><li>Best performance, but updates can pass each other, resulting in inconsistent data</li><li>Network disruptions can cause problems</li></ul>
<b>distributed-ack (default)</b>	<ul style="list-style-type: none"><li>Updates are sent synchronously, in parallel, to all members, and acknowledged back to sender</li><li>Data eventually consistent based on vector clocks</li><li>No explicit locking – but none really needed</li></ul>
<b>global</b>	<ul style="list-style-type: none"><li>All load, create, put, invalidate, and destroy operations on the Region are performed with a distributed lock on the key</li><li>Not as significant of a feature given new eventual consistency</li><li>The slowest option</li></ul>

# Creating a Replicated Region

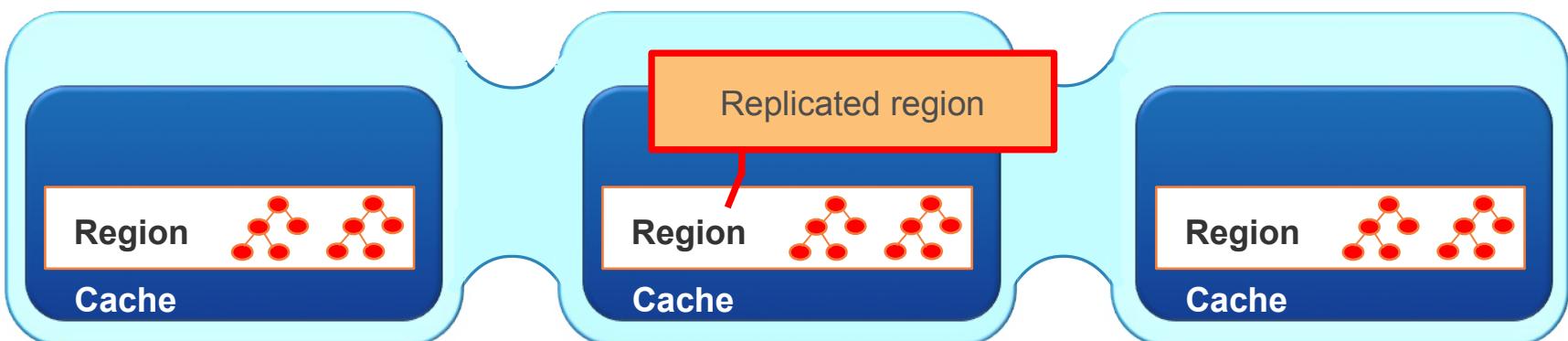
Declare in XML:

```
<region name="MyRegion">
    <region-attributes refid="REPLICATE"
        scope="distributed-no-ack"/>
<region>
```

REPLICATE is the default region type

Or in Java:

```
Region region = cache
    .createRegionFactory(RegionShortcut.REPLICATE)
    .setScope(Scope.DISTRIBUTED_NO_ACK).create("MyRegion");
```



# Partitioned or Replicated Regions?

- Partitioned
  - Store data sets that are too large to fit in a single VM, single cache member
  - 1 to Many relationships
  - Suitable for 100's of GB
  - Represents most typical cases so start with Partitioned regions
- Replicated
  - Read heavy, **small** data set (reference or lookup data)
  - Ideal for Many to Many relationships
  - Data sets that can be entirely contained in a single member
  - Represents the more rare case
  - CANNOT horizontally scale

# Lab

## In this lab, you will

- 1.Create a replicated region across multiple servers
- 2.Test the failover and refresh of data on recovered nodes
- 3.Create a partitioned Region using default partitioning
- 4.Create redundant partitions
- 5.Use gfsh to view the Partitioning

# Review of Learner Objectives

**You should now be able to do the following:**

- Describe and configure partitioned regions
- In partitioned regions, describe how data entries are placed in buckets based on hashing
- Describe how partitioned regions behave in the case of a hardware failure
- Describe and configure replicated regions

# Pivotal

BUILT FOR THE SPEED OF BUSINESS

# Working With the Client Cache

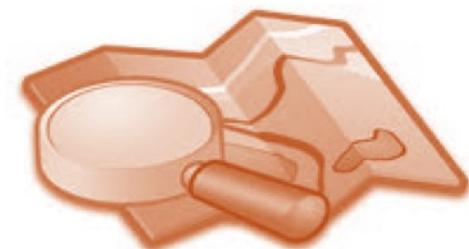
# Learner Objectives

After this lesson, you should be able to do the following:

- Configure the client cache
- Describe the relationship between client regions and corresponding server regions
- Understand how to interact with the cache
- Understand how updates work in cache

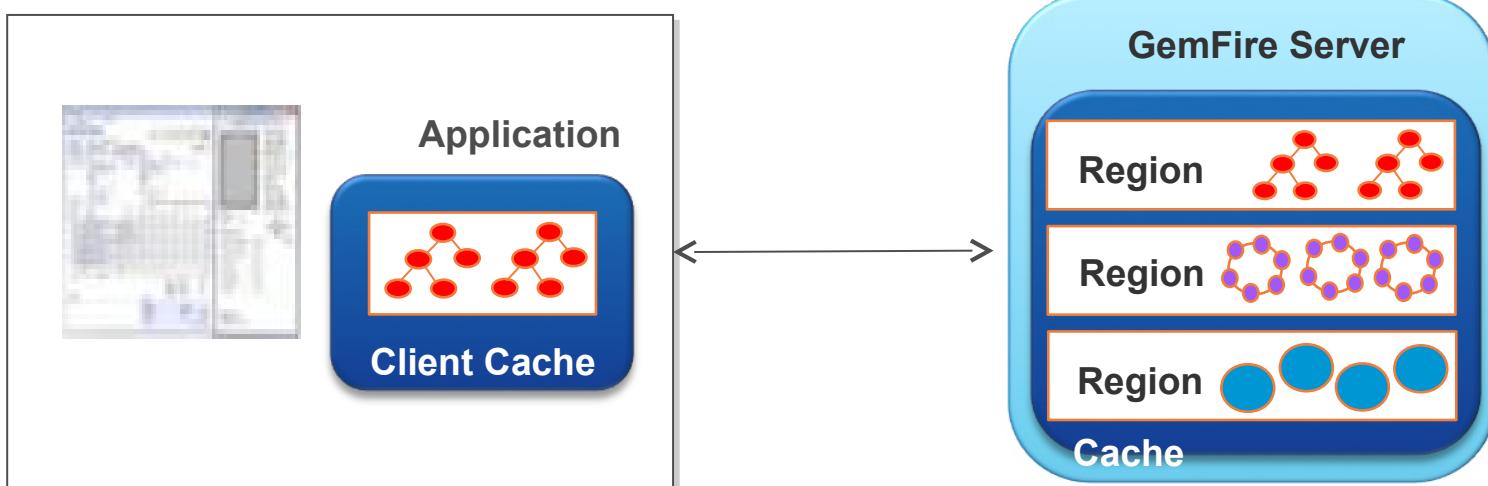
# Lesson Road Map

- **Client Cache and Connection Pools**
- Client Region Types
- Interacting with Server Side Regions
- CRUD Operations

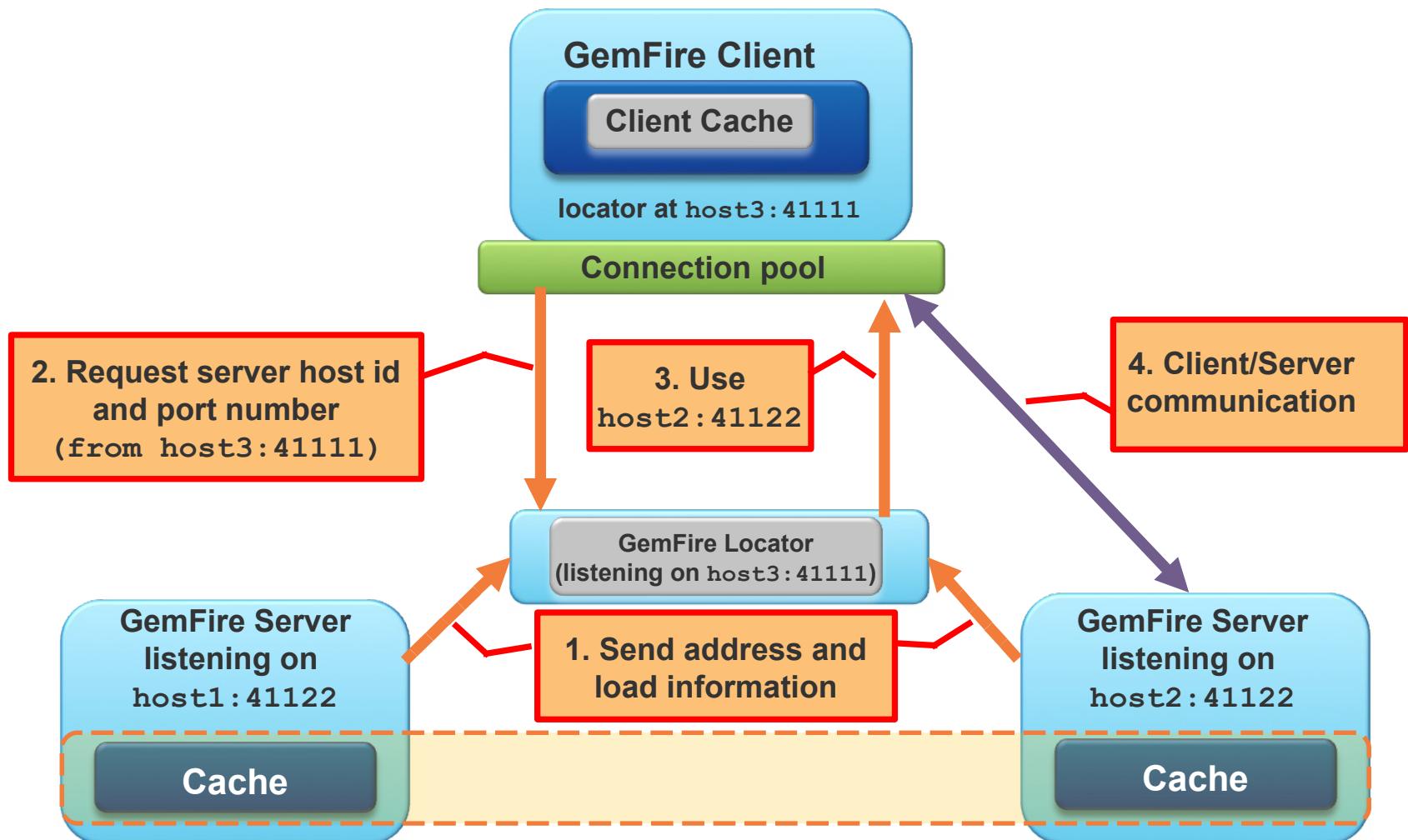


# Review: Client Cache

- Connected to a GemFire server or set of servers
  - host and port information provided by the locator
- Can have their own local copy of data
  - Can register for changes,
  - Server notifies the client as needed when the data changes

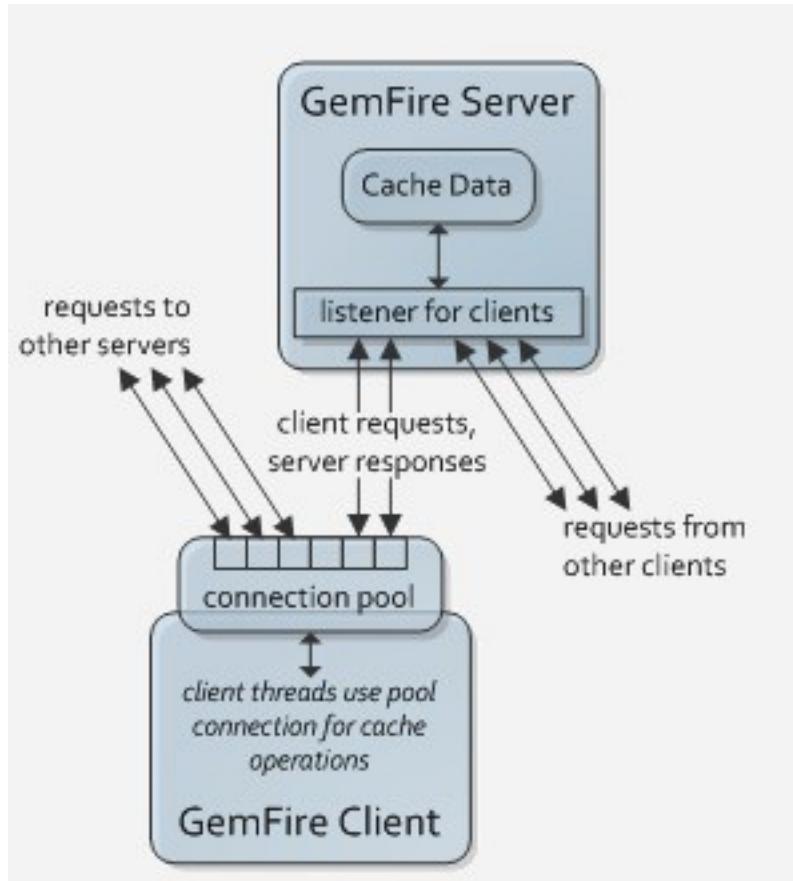


# Review: Client – Locator – Server Architecture



# Client Connection Pools

- Manages client connections to servers
  - Used to send operations from client
  - Connect to host/port of server(s)
- Generally configured with one or more locators



# How Pool Chooses Server Connection

- Use one or more Locators
  - Provides elasticity
  - Can provide connection to ‘least loaded’ server
  - If more than one locator provided, client randomly chooses
  - Best option when available servers in cluster are dynamic

# Configuring the Client Cache

The client's cache.xml file contains the connection pool and region-attributes configuration details:

```
<client-cache>
    <pool name="locatorPool">
        <locator host="host3" port="41111"/>
    </pool>
    <region name="Inventory">
        <region-attributes refid="PROXY" pool-name="locatorPool" />
    </region>
</client-cache>
```

More than one locator possible

Optional when only one pool defined

# Managing Connections

- New Pool connection added when:
  - Number of open connections less than `min-connections` setting
  - Thread needs a connection and all open connections are in use
- Pool requests server connection info from locator
  - Connection request sent to server
- Connections closed when:
  - Client receives connectivity exception from server
  - Server unresponsive (read timeout or no response from ping)
  - Idle connections > `min-connections` setting

# Locators and Server Load

- Pool configured with locator(s) can adjust to changing server load
  - Each connection maintains internal lifetime counter
  - Periodically, the connection checks back with locator
  - Locator may offer a new least loaded server
  - All done automatically and behind the scenes
  - Allows clients to respond seamlessly and dynamically to changes in the cluster

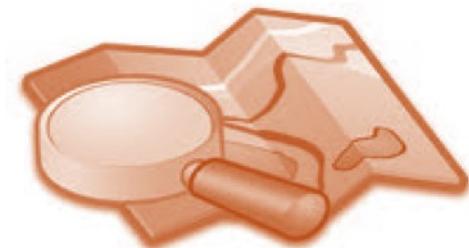
# Additional Client Pool Options

Property	Description	Default
idle-timeout	Maximum time, in milliseconds, a pool connection can stay open without being used when there are more than min-connections in the pool. -1 = no idle timeout	5000
min-connections	Minimum number of pool connections to keep available at all times. If set to 0 (zero), no connection is created until an operation requires it.	1
max-connections	Maximum number of pool connections the pool can create. If set to -1, there is no maximum. The setting must indicate a cap greater than min-connections.	-1
pr-single-hop-enabled	Indicates whether to use metadata about the partitioned region data storage locations to decide where to send data requests. Allows a client to send a data operation directly to the server hosting the key. Otherwise, the client contacts any available server and that server contacts the data store.	true
ping-interval	How often to communicate with the server to show the client is alive, in milliseconds. Pings are only sent when the ping-interval elapses between normal client messages.	10000
retry-attempts	Number of times to retry a client request before giving up. If one server fails, the pool moves to the next, and so on until it is successful or it hits this limit. -1 = the pool tries every available server once.	-1
server-group	Logical named server group to use from the pool. null = use the global server group to which all servers belong.	null

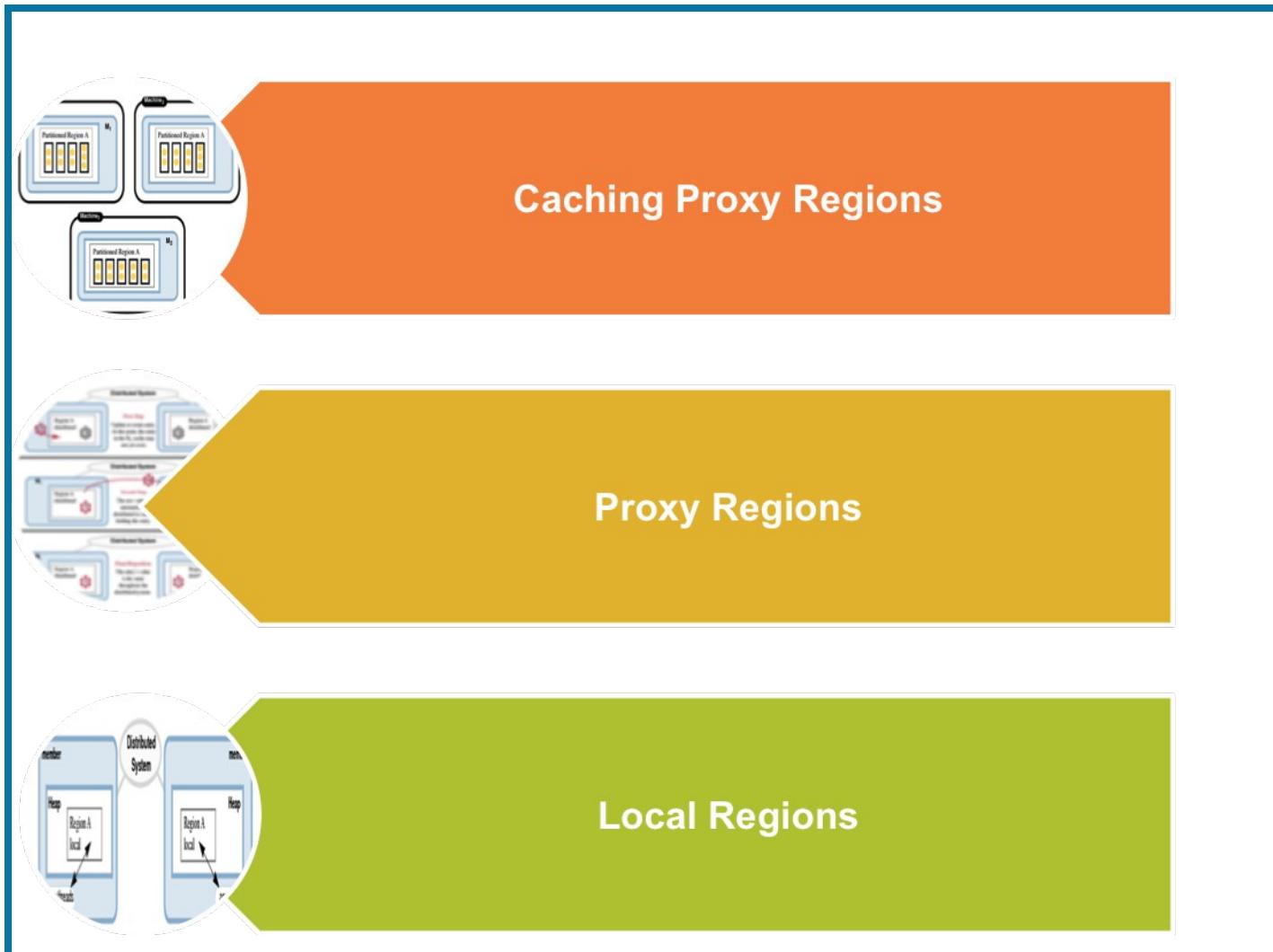
Not a complete list. See user's guide, <client-cache> Element Reference for more details.

# Lesson Road Map

- Client Cache and Connection Pools
- **Client Region Types**
- Interacting with Server Side Regions
- CRUD Operations



# Types of Regions: Review



# Region Shortcuts

- GemFire provides region shortcut settings, with preset region configurations for the most common region types.
- For the easiest configuration, start with a shortcut setting and customize as needed.
- You can also store your own custom configurations in the cache for use by multiple regions
- Defined in RegionShortcut API:

<http://gemfire.docs.pivotal.io/latest/javadocs/japi/com/gemstone/gemfire/cache/client/ClientRegionShortcut.html>

Example:

```
<region refid="LOCAL"/>
```

Region Shortcut

# Client Cache Using Region Shortcuts

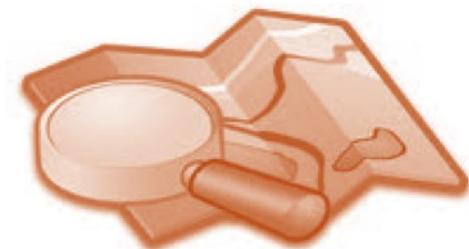
```
<client-cache>
  <pool name="locatorPool">
    <locator host="host3" port="41111"/>
  </pool>
  <region name="Inventory">
    <region-attributes refid="PROXY" pool-name="locatorPool"/>
  </region>
  <region name="clientsPrivateR">
    <region-attributes refid="LOCAL"/>
  </region>
</client-cache>
```

# Client Regions with Region Shortcuts

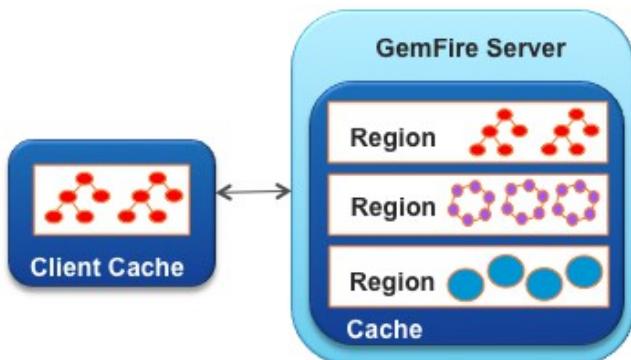
- LOCAL
  - Local region – not distributed
- PROXY
  - Region acts as a proxy to server side region by same name
  - No data kept locally
  - All operations, such as get(), put() require call to server (extra network hop)
- CACHING\_PROXY
  - Data kept locally, but also forwards to server
  - Gets avoid call to server if key is in the client side region
- Additional shortcuts explored later

# Lesson Road Map

- Client Cache and Connection Pools
- Client Region Types
- **Interacting with Server Side Regions**
- CRUD Operations



# ClientCache - API



- Starting point for access to GemFire
  - Generally interact with Regions
  - Entry point is via ClientCache
  - One ClientCache instance per process
- Classes in  
com.gemstone.gemfire.cache.client

```
// Create the cache without using a cache xml file
ClientCache cache = new ClientCacheFactory().create();

// Create the cache and setting properties
ClientCache cache = new ClientCacheFactory()
    .set("cache-xml-file", "xml/clientCache.xml")
    .set("log-level", "config")
    .create();
```

# Setting Properties

- Many properties available for configuring GemFire client
- Three main ways to set properties, in priority order:
  - Command line properties (passed as –D arguments)
  - Via gemfire.properties file
  - Through ClientCacheFactory class
- See Reference section of GemFire User's Guide for detailed list of properties
  - Some not supported for Client Cache
    - *Example:* mcast-port, async-max-queue-size, ...
  - Any of these can be set using above three approaches

# Setting startup parameters

In gemfire.properties:

```
log-level=config  
durable-client-timeout=400  
cache-xml-file=ClientCache.xml
```

- Allows you to set some configuration at start up of overall process
- Does not require re-compile to change properties
- Can specify cache-xml for Cache configuration

# Getting or Creating Regions

- Regions created by cache-xml-file definition can be simply fetched via API

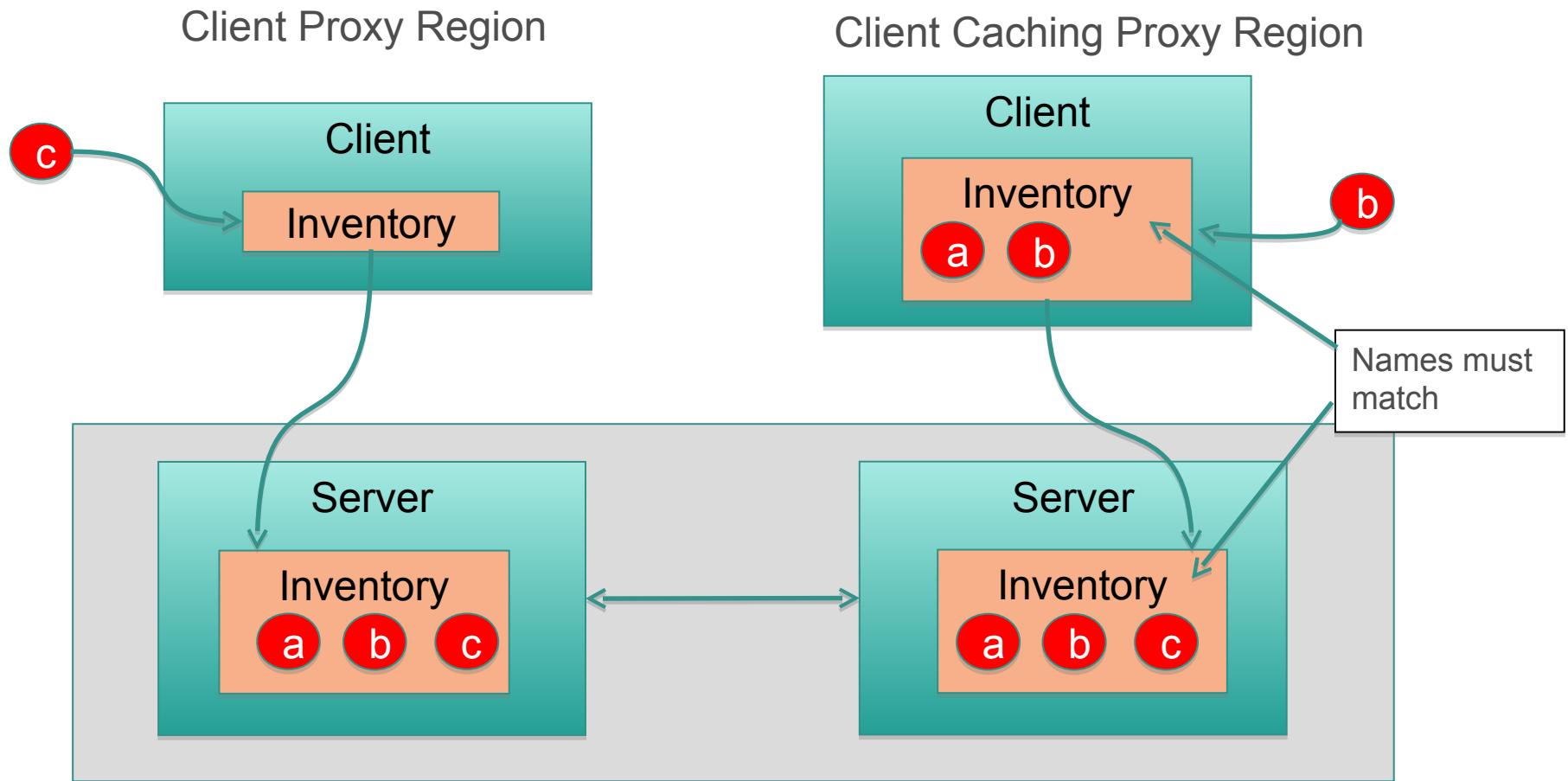
```
ClientCache cache = new ClientCacheFactory()
    .set("cache-xml-file", "xml/clientCache.xml")
    .create();
Region<Integer, Customer> customers = cache.getRegion("Customer");
```

- Regions can also be created programmatically

```
// Create the cache without using a cache xml file
ClientCache cache = new ClientCacheFactory().create();

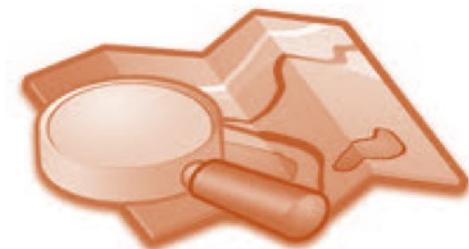
// Create a region dynamically using the APIs
Region<Integer, Customer> customers =
    (Region<Integer, Customer>) cache.createRegionFactory()
    .create("Customer");
```

# Relationship of Client and Server Region



# Lesson Road Map

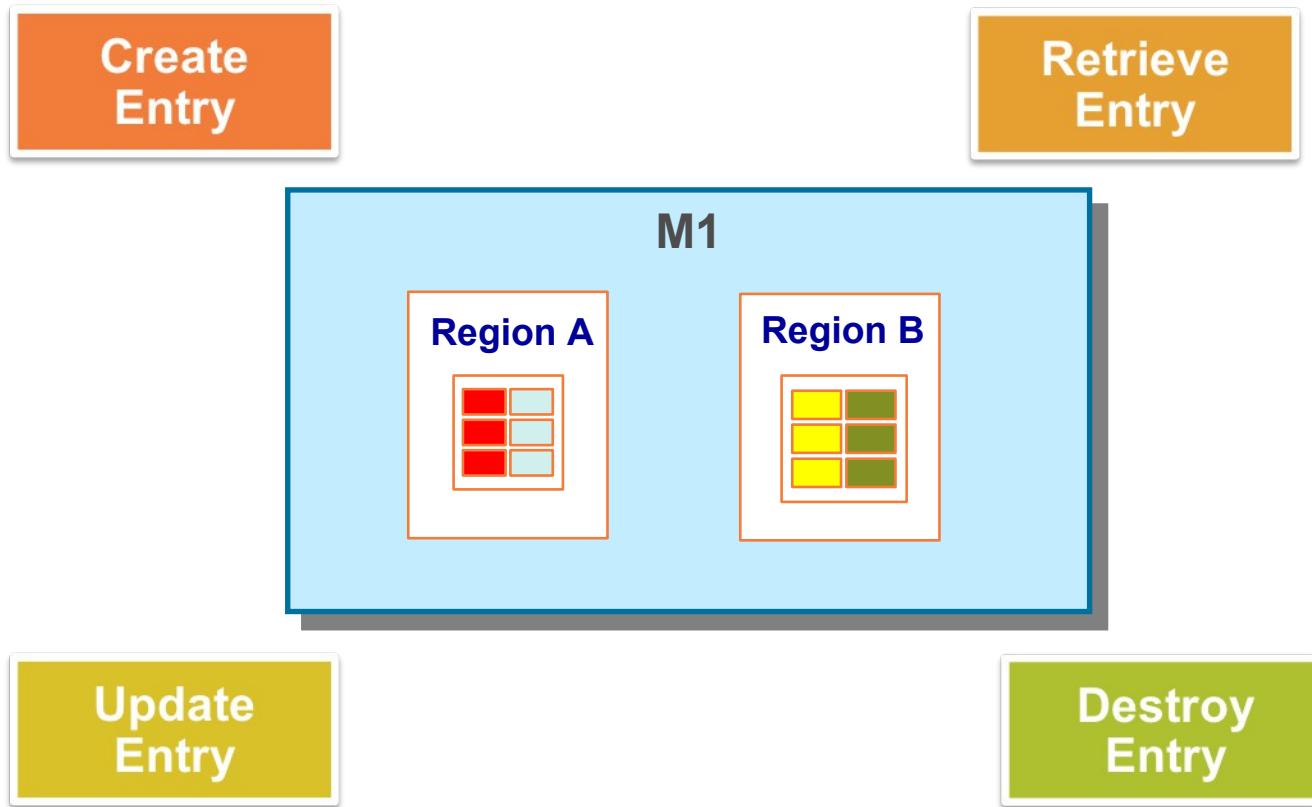
- Client Cache and Connection Pools
- Client Region Types
- Interacting with Server Side Regions
- **CRUD Operations**



# Storage Overview

- Storage involves a HashMap approach
- Keys & Values must be serializable – only comes into play when interacting with server cache
- Keys must implement hashCode() and equals()
- Typically, data architect or server side developer will define & provide

# “CRUD” Operations



# GemFire Region API

- GemFire's API has been influenced by a number of sources
  - JSR-107 attempts to define a standard caching API
  - Additional functionality added as needed
  - Some methods overlap in behavior
- JSR 107 Cache API
  - Standard CRUD operations (get, put, remove)
  - Atomic operations (putIfAbsent, getAndReplace, etc)
  - Not all found on Region API (ex getAndPut, getAndRemove)

# Create or Update Entry

To create or update entries in a Region in the cache, you can use the following Gemfire APIs:

API	JSR 107	Description
create(key, value)	No	Creates new entry in the Region or throws EntryExistsException if key already exists
put(key, value)	Yes	Creates or updates key and value in the Region
putAll(Map)	Yes	Create or update an entire set of entries in the Region from the entries in the Map
putIfAbsent(key, value)	Yes	Atomic action: put if not there – either return existing value or new value
replace(key, value)	No	Atomic action: update if there but do nothing if not – either return updated value or null. This is the way to achieve “update” semantics

Note: All of these methods create or update the entries in both the local cache AND the server side cache unless you are performing them on a “LOCAL” Region.

# Retrieve Entry

To retrieve entries from a Region in the cache, you can use the following Gemfire APIs:

API	JSR 107	Description
get(key)	Yes	Returns the value associated with this key from the Region or null if this key is not present in the Region
getAll(keys)	Yes	Returns a Map containing all of the keys and values from the Region based on the collection of keys passed as the argument

# Destroy Entry

To destroy entries from a Region, you can use the following Gemfire APIs:

API	JSR 107	Description
destroy(key)	No	Removes the entry with the specified key from the Region in the local cache and all copies in the servers.
destroy(key, value)	No	Atomic action: Removes the entry for the key only if currently mapped to the given value
invalidate(key)	No	Sets the entry for the key to the “invalid” state on the Region in the local cache and all copies in the servers. Subsequent attempts to get(key) will return null.
remove(key)	Yes	Removes the entry with the specified key from the Region in the local cache and all copies in the servers.
remove(key, value)	Yes	Atomic action: Removes the entry for the key only if currently mapped to the given value

# Local Destroy Entry

To destroy entries from a Region in the local cache only, you can use the following Gemfire APIs:

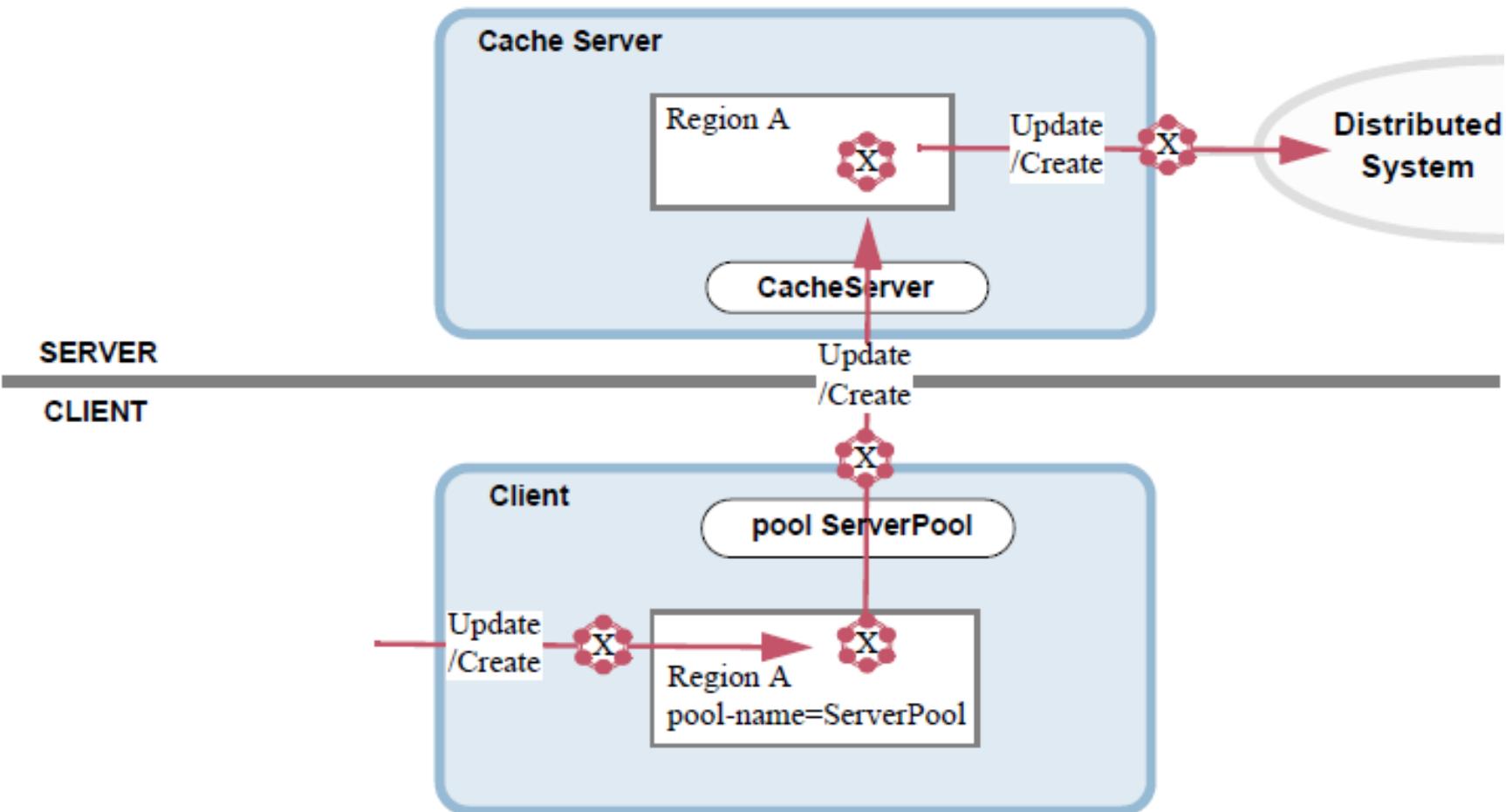
API	JSR 107	Description
localDestroy(key)	No	Destroys the entry with the specified key from the local cache only
localInvalidate(key)	No	Sets the entry for the key to the “invalid” state in the local cache only. Subsequent attempts to get(key) will return go to the server to re-fetch the entry.

# Other interesting methods on Region

Method	Description
containsKey(key)	Returns true if key exists in region
entrySet	Returns a map (key & value) of all entries in local region. Objects can't be modified. If using a client region (ex CACHING_PROXY), then it will be entries in client region.
keySet	Returns a map of keys in the region. If using a ClientRegion region, it will be the local key set
keySetOnServer	Returns the entire set of keys for the region on the server

\* Not an exhaustive list

# Client-Initiated Data Flow



# Lab

**In this lab, you will:**

- 1.**Start servers and execute a test client
- 2.**Configure a client cache XML file to interact with the servers
- 3.**Implement client code to initialize client cache and interact with servers

# Pivotal

BUILT FOR THE SPEED OF BUSINESS

# Querying for Data

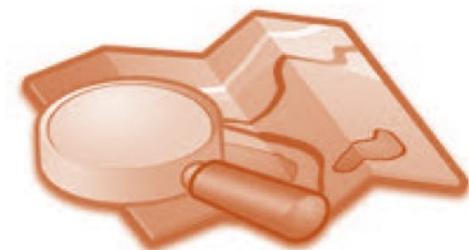
# Learner Objectives

**After this lesson, you should be able to do the following:**

- Describe the difference between relational and object queries
  - Describe how to get access to the GemFire Query Service
  - Use OQL to access objects in GemFire

# Lesson Road Map

- **QueryService and OQL Overview**
- OQL Detail
- Advanced Query Topics
- Indices
- DataBrowser



# Accessing GemFire via a query

- Very similar to JDBC calls and result sets
  - Ask for a QueryService from the cache
  - Pass in a query String
  - Iterate over results
- Failures throw an Exception
- Same rules apply
  - Don't pull back unnecessary results
  - Be judicious in how you use resources

# Java for Queries

```
String queryString = //my query  
QueryService qs = cache.getQueryService();  
Query q = qs.newQuery(queryString);  
SelectResults<?> results = (SelectResults<?>)q.execute();
```

GemFire com.gemstone.gemfire.cache.query	JDBC java.sql
QueryService	Connection
Query	Statement
SelectResults (Collection)	ResultSet (Set)

# Getting the QueryService

- Query a local client cache

```
QueryService qryService =  
    clientCacheInstance.getLocalQueryService();
```

- Query the client's server caches for the specified Pool

```
QueryService qryService =  
    clientCacheInstance.getQueryService(poolName);
```

# Iterating over results

- Results are of type  
com.gemstone.gemfire.cache.query.SelectResults
- SelectResults implements java.util.Collection

```
String queryString = //my query
QueryService qs = cache.getQueryService();
Query q = qs.newQuery(queryString);
SelectResults <Customer> results =
    (SelectResults <Customer>)q.execute();
for(Customer c : results) {
    System.out.println("Customer is: " + c);
}
```

# Lesson Road Map

- QueryService and OQL Overview
- **OQL Detail**
- Advanced Query Topics
- Indices
- DataBrowser



# Object Query Language (OQL)

- Query language derived from SQL to be used with object oriented databases
- Created by the Object Data Management Group (ODMG) as the standard for querying objects
- Can be used to:
  - Do mathematical calculations
  - Walk through nested objects
  - Reference objects within Regions
- Used to retrieve an object from the Region when you don't already have the key

# Simple Query

	GemFire	Relational
<b>IMPORT</b>	Classes the query will use directly	Everything is a table
<b>SELECT</b>	Object(s) to return	Columns to return
<b>FROM</b>	Collections to input to WHERE	Tables to input to WHERE
<b>WHERE</b>	Conditions, objects	Conditions, rows

```
IMPORT com.bookshop.domain.BookOrder;  
  
SELECT DISTINCT custOrder  
  
    FROM /BookOrders.values custOrder TYPE BookOrder  
  
    WHERE custOrder.customerNumber=5543
```

# IMPORT Statements

- Allows you to provide custom Java classes to the query engine
- Passed in the same string as the query to be executed
- Can provide an alternate name when you have conflicting class names
- Primitive and built-in types do not require an IMPORT statement (int, float, etc...) – Integer is the exception

Fully qualified class name,  
accessible from the classpath

```
IMPORT com.myFolder.Customer;  
IMPORT com.myFolder.Customer AS myCustomer;
```

Alternate class name  
(Optional)

# Invoking methods on Objects

Query engine looks for the attribute following a priority

- public method getX()
- public method x()
- public field x

Size method on a collection  
= myBookOrders.size()

```
SELECT DISTINCT * FROM /Customers WHERE myBookOrders.size >= 2
```

```
SELECT DISTINCT * FROM /Customers c  
WHERE c.lastName.startsWith('Walsh')
```

Customer

```
private Integer custNumber  
private String firstName  
private String lastName  
private List myBookOrders
```

Equivalent to  
c.getLastName().startsWith("Walsh")

# SELECT Statement returning an Object

- Returns a Collection of a specific object type

```
IMPORT com.bookshop.domain.Customer;  
SELECT DISTINCT c  
FROM /Customer.values c TYPE Customer
```

Returns all Customer objects  
From the /Customer region

- Use Collection as normal – No O/R mapping required

```
SelectResults<Customer> results =  
    (SelectResults<Customer>)q.execute();  
  
Iterator<Customer> i = results.iterator();  
  
while (i.hasNext()) {  
    System.out.println("Customer is: " + i.next());  
}
```

# SELECT Statement Projection List

- A list of object values with "basic" types can be declared similarly to SQL

Selects customerNumber, firstName, and lastName from all customer Objects

```
IMPORT com.bookshop.domain.Customer;
SELECT DISTINCT c.customerNumber, c.firstName, c.lastName
FROM /Customer.values c TYPE Customer
```

- A 'Struct' is created that is similar to a Properties object

```
SelectResults <Struct> results =
        (SelectResults <Struct>)q.execute();
Iterator <Struct> i = results.iterator();
while (i.hasNext()) {
    Struct struct = (Struct) i.next();
    int key = ((Integer) struct.get("customerNumber")).intValue();
    String firstName = (String) struct.get("firstName");
}
```

# FROM Clause

- Establishes the Collections that the rest of the query will use
- Comma delimited list of Collections
- Use Iterator variables to be referenced elsewhere in the query
- Explicitly type the members of the Collection
- Can be Regions, or other Collections, just need to implement `java.util.Collection`

```
IMPORT com.bookshop.domain.Customer;
IMPORT com.bookshop.domain.Address;
SELECT DISTINCT theAddress
  FROM /Customers.values c TYPE Customer,
       c.addresses theAddress TYPE Address
 WHERE theAddress.country='US'
```

Explicitly typing the collection,  
Declared in IMPORT

Iterator variable

# WHERE Clause

- WHERE clause expressions are boolean conditions that are evaluated for each element in the collection
- Filters elements in the FROM clause. Without it, the SELECT gets all members of the collection.
- Can optimize search by *indexing* on an attribute.
- If the attribute is a reserved word, place it in double quotes
- Cannot do joins across partitioned regions
- With an "AND", place most restrictive condition first

```
SELECT DISTINCT p
  FROM /Persons p TYPE Person,
       /Flowers f TYPE Flower
 WHERE p.name = f.name
```

Creates a Join

# Query Parameters

- Can define parameters in a query to execute multiple times
- Parameters are defined by \$1, \$2, etc...
- Object array with runtime parameters handed to "execute"

```
// Selects all Customers where myBookOrders.size > $1 and status = $2
qryService.newQuery( "SELECT DISTINCT * "
    + "FROM /Customers "
    + "WHERE myBookOrders.size > $1 AND status = $2") ;

Object[] params = new Object[2];
params[0] = new Integer(2);
params[1] = "active";

SelectResults results = query.execute(params);
```

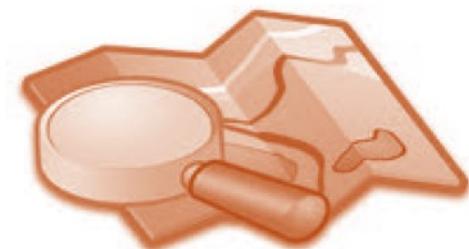
Note: myBookOrders is relative to  
the current object from Customers  
region

# OQL Extras

- Comments
  - ‘--’ (double dash) to comment a line
  - /\* block comments here \*/
- OQL keywords (SELECT, IMPORT) are not case sensitive
- Identifiers such as attribute names, method names, and path expressions are case-sensitive.

# Lesson Road Map

- QueryService and OQL Overview
- OQL Detail
- **Advanced Query Topics**
- Indices
- DataBrowser



# Region Querying Attributes

- You can query region's attributes
  - ex *select s from /RegionName.size s;*
  - Generally, look at Region API definition

Attribute	Description	Example
/RegionName.keySet	Returns the Set of entry keys only	<code>select * from /Customers.keySet</code>
/RegionName.entrySet	Returns Set of objects in the region (key & value)	<code>select key, value from /Customers.entrySet</code>
/RegionName.values	Returns the Collection of entry values	<code>select * from /Customers.values c where c.status = 'active';</code>
/RegionName	Returns the Collection of entry values (same as values)	<code>select * from /Customers</code>
/RegionName.size	Returns the number of entries in the region (note: This would be in the client cache)	<code>select s from /Customers.size s;</code>

# More on Bind Parameters

- It's possible to parameterize the region (path)
- The parameter must resolve to a collection
  - Could be the region itself
  - Could be some other collection object that would be queried

```
qryService.newQuery( "SELECT DISTINCT * "
    + "FROM $1 "
    + "WHERE myBookOrders.size > $2 AND status = $3");
Object[] params = new Object[3];
params[0] = cache.getRegion("Customers");
params[1] = new Integer(2);
params[2] = "active";
SelectResults results = query.execute(params);
```

# Using IN and SET

- IN – Boolean expression: is expr1 found in expr2 where expr2 is a set of values
  - Ex: 2 in SET (1,2,3) = true
- What can be in SET
  - Literal values (integers, strings, decimals)
  - Sub-query

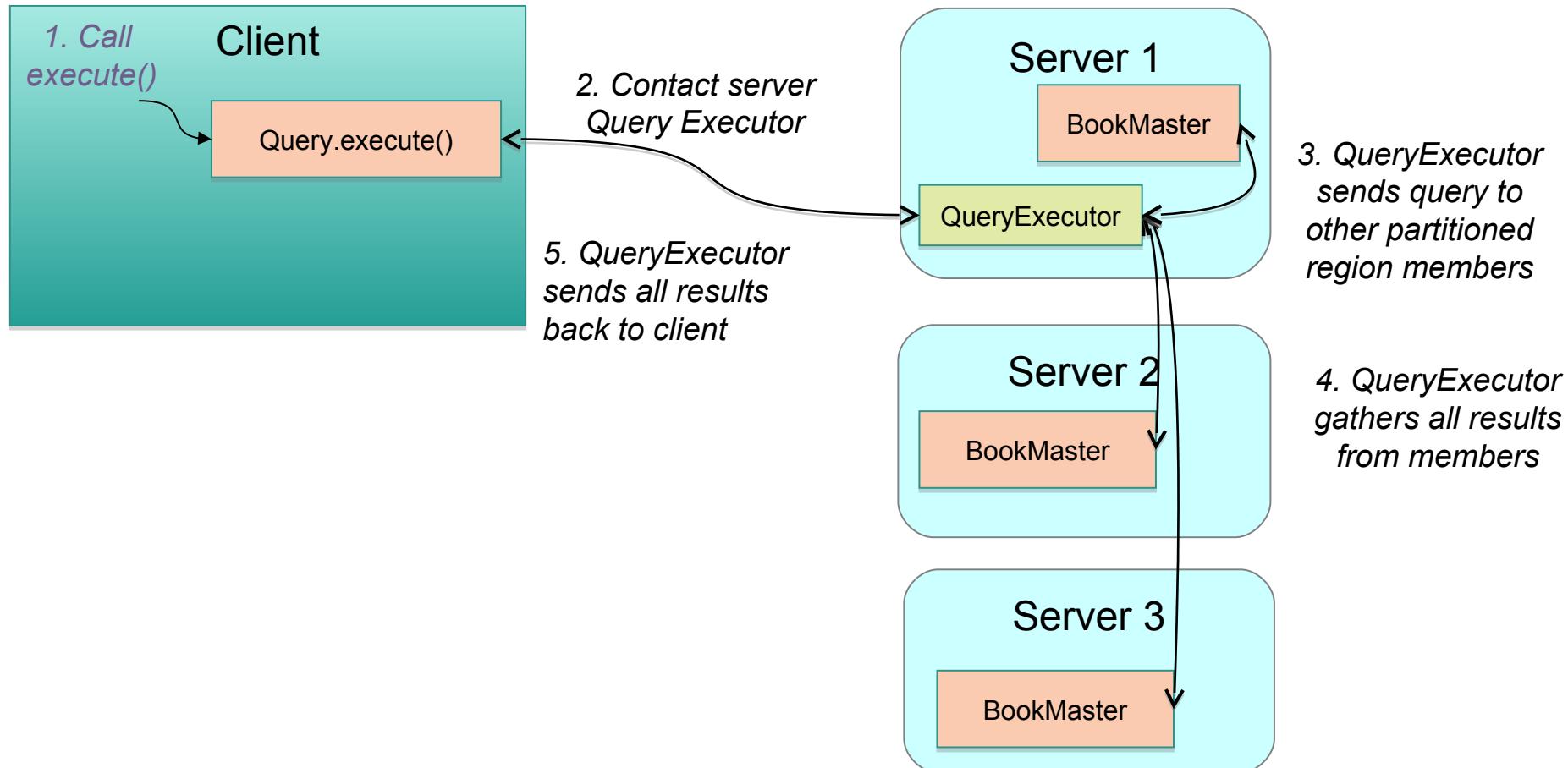
```
SELECT * from /Region1 WHERE id IN
    (SELECT id from /Region2 WHERE status='active');
```
- Sets can also be compared
  - When result being compared is a Set:  

```
SELECT * from /REGION1.entrySet WHERE e.setProperty = SET(1,2,3)
```
  - When result being compared is a List:  

```
SELECT * from /REGION1.entrySet
    WHERE e.listProperty.containsAll(SET(1,2,3));
```

e.setProperty is a  
field of type Set

# Query Execution and Partitioned Regions



# Partitioned Regions

- Partitioned region can be the only region in the FROM clause.
- If a partitioned region is configured for redundancy, the query is distributed only to the primary buckets.
- To query a partitioned region the system distributes the query to all buckets across all of the region's data stores, then merges the result sets and sends back the query results.
- Since partitioned regions contain large data sets, result sets may overwhelm the calling member's memory if they are not appropriately restricted
  - Consider using functions in cases like these, or rethink your OQL



To query a specific node in a partitioned region, use functions.

# Queries in Partitioned Regions

- Targeting specific nodes
  - May want to use functions (coming later) to target ‘data aware’ function to query from the server side
- Improving performance using Key indices
  - Queries can be optimized by targeting specific servers
  - Using Functions to execute query & use an index
- Joins across partitioned regions
  - Not supported from the client side
  - Can be executed using equi-join from server side function
- When using ORDER BY, the field(s) specified must be specifically called out in a projection list

# Query Language Restrictions

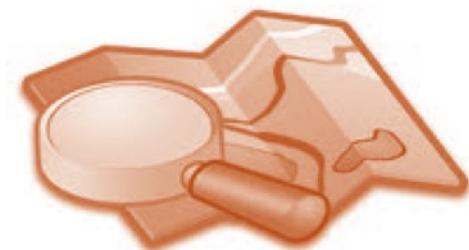
- GROUP BY not supported
- Can't invoke static methods in where clause
  - Ex: select \* from /Region where value = Obj.staticMethod;
  - You can use an enum on the right side
- ORDER BY only supported when using DISTINCT
- No pagination – common workaround:
  - Select keys matching where clause
  - Build list of keys representing ‘page’
  - Call getAll(keys) on region to fetch page of entries

# Well structured queries

- Similar rules apply as to relational – make sure queries make sense
- Consider what amount of data your query will generate on your client
- Reduce the operational set as quickly as possible.
- Do performance and set testing on OQL

# Lesson Road Map

- QueryService and OQL Overview
- OQL Detail
- Advanced Query Topics
- **Indices**
- DataBrowser



# Caveats of Using Indices

- In general, an index improves query performance if the FROM clauses of the query and index match exactly
  - If there is not an exact match, performance may degrade
  - Do performance testing if you add multiple indices on a region
- Indices consume memory to maintain, increasing the object size of the object containing the index
- Each update of an attribute in the index requires a recalculation, consuming processing time
  - Rethink an index that will be updated frequently
- The first index that the FROM clause reaches will be the one that gets used – currently

# GemFire Query Index Types

- Range Index
  - Default index type
  - Primarily used when evaluation is something besides equality
- Key Index
  - Used in partitioned regions where specified field is the key
  - Only works for equality queries
  - Can NOT be used in equi-join queries
- Hash Index
  - Improved memory utilization for index on a field
  - Only supports equality queries
  - Implemented as a hash on the field and a pointer to the object in the region

# Range Index

- Best when query involves non-equality comparisons
- Default index type

gfsh

```
create index --name=titleIndex --expression=title  
--region=/BookMaster
```

cache.xml

```
<cache>  
    <region name="BookMaster">  
        <index name="titleIndex" from-clause="/BookMaster"  
              expression="title" />  
    </region>  
</cache>
```

Java

```
qryService.createIndex("titleIndex", "title", "/BookMaster");
```

# Key Index

- Great when a given field is the key for entry
- Only supports equality comparisons

gfsh

```
create index --name=myKeyIndex --type=key --expression=itemNumber  
          --region=/BookMaster
```

cache.xml

```
<cache>  
    <region name="BookMaster">  
        <index name="myKeyIndex" from-clause="/BookMaster"  
              expression="itemNumber" key-index="true" />  
    </region>  
</cache>
```

Java

```
qryService.createKeyIndex("myKeyIndex", "itemNumber",  
                         "/BookMaster");
```

# Hash Index

- Efficient index storage
- Only supports equality comparisons

gfsh

```
create index --name=titleIndex --type=hash --expression=title  
          --region=/BookMaster
```

cache.xml

```
<cache>  
    <region name="BookMaster">  
        <index name="titleIndex" from-clause="/BookMaster"  
              expression="title" type="hash" />  
    </region>  
</cache>
```

Java

```
qryService.createHashIndex("titleIndex", "title", "/BookMaster");
```

# Creating Multiple Indexes

- Handy when creating multiple indexes on the same region
- Avoids iterating over entries in a region for each index

## gfsh

```
gfsh> define index --name=titleIndex --expression="title" --region=/BookMaster  
gfsh> define index --name=authorIndex --expression="author" --region= /BookMaster  
gfsh> create defined indexes
```

## Java

```
qryService.defineIndex("titleIndex", "title", "/BookMaster");  
qryService.defineIndex("authorIndex", "author", "/BookMaster");  
List<Index> indexes = queryService.createDefinedIndexes();
```

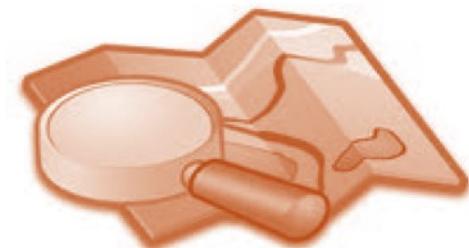
# Query Index Hints

- GemFire provides a way to provide ‘hints’ to the query engine
  - Good when multiple indexes defined BUT want to prefer one (or more)
  - Multiple indexes can be specified

```
String query1 = "<HINT 'authorIndex' > select * from /BookMaster " +  
                  "WHERE author = 'Clarence Meeks'"  
  
String query2 = "<HINT 'authorIndex', 'priceIndex' > " +  
                  "select * from /BookMaster " +  
                  "WHERE author = 'Clarence Meeks' and retailCost > 10.0"  
  
Query q = qryService.newQuery(query1);  
  
..."
```

# Lesson Road Map

- QueryService and OQL Overview
- OQL Detail
- Advanced Query Topics
- Indices
- **DataBrowser**



# GemFire DataBrowser

- GemFire DataBrowser utility enables you to query data interactively with OQL ad-hoc queries
- Was a stand-alone tool but now part of Pulse
- Perform tasks, such as:
  - Execute ad-hoc queries on any data region on any cache node in the distributed system
  - Configure the maximum number of rows returned by any query to avoid excessive resource use on the server
  - Simultaneously execute multiple continuous queries on one or more cache servers in the distributed system

# GemFire DataBrowser

Data Browser now just a tab on Pulse

myBookOrders	lastName	test	customerNumber	firstName	primaryAddress
java.util.ArrayList	Powell	0	5598	Kari	com.gemstone.gemf...
java.util.ArrayList	Garcia	0	6024	Trenton	com.gemstone.gemf...
java.util.ArrayList	Wax	0	5543	Lula	com.gemstone.gemf...

# Starting Pulse from gfsh

- Connect to the locator by using gfsh prompt:

```
gfsh> connect --locator=host3[41111]
```

- Launch GemFire Pulse by using the following command:

```
gfsh> start pulse
```

# Lab

**In this lab, you will:**

1. Learn to use the Query Service
2. Build a series of OQL Statements
3. Perform a Join using OQL

# Pivotal

BUILT FOR THE SPEED OF BUSINESS

# Custom Partitioning

# Learner Objectives

After this lesson, you should be able to do the following:

- Describe the difference between standard and Custom Partitioning
- Configure Custom Partitioning for GemFire
- Describe Co-location of Regions and when to use it

# Lesson Road Map

- **Custom Partitioning**
- Collocation

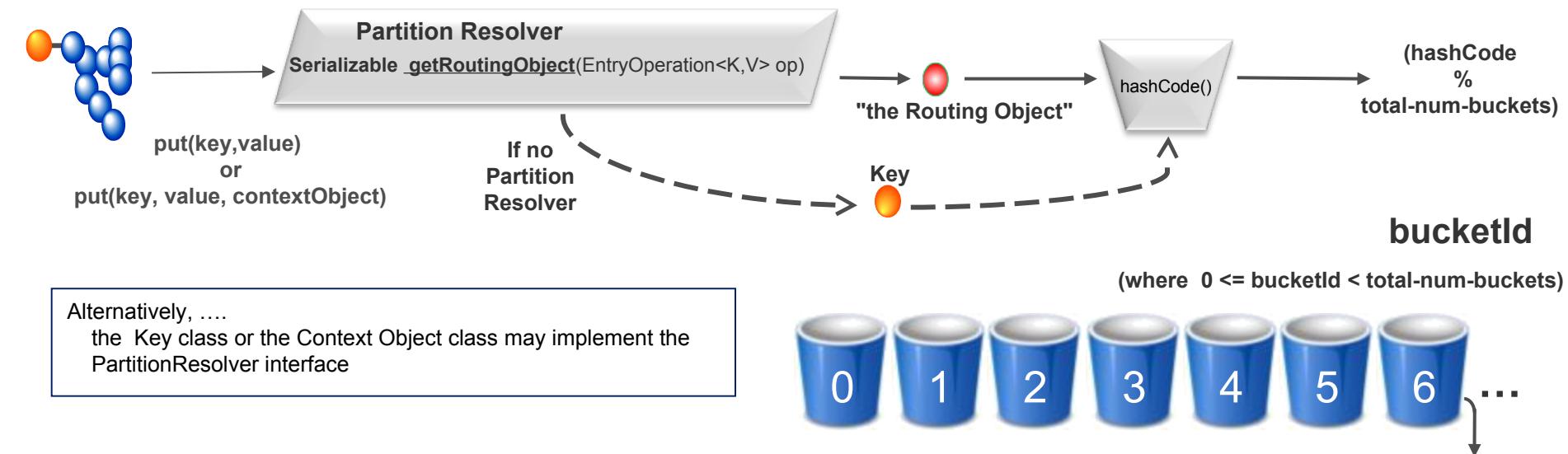


# Custom Partitioning – Why?

- Put related data on a single host
  - Single Region
    - Similar data together within a cache member.
  - Multi Region
    - Similar regions/data together within a cache member.
    - "Co-location" of data from two different regions.
- Transactions
  - Execute transaction across data items that are co-located with in the same JVM.
- Function Execution
  - Execute application processing on related data in a single JVM.
  - Remove unnecessary process hops.

**Replaces default hashing policy**

# Partition Buckets and Custom PartitionResolvers



# Partition XML

```
<cache>
  <region name="Order">
    <region-attributes refid="PARTITION">
      <partition-attributes redundant-copies="1">
        <partition-resolver name="CustomerPartitionResolver">
          <class-name>myPackage.CustomerPartitionResolver</class-name>
        </partition-resolver>
      </partition-attributes>
      <region-attributes>
        <region>
      </region>
    </region-attributes>
  </region>
</cache>
```

# Standard Custom Partitioning

- Entries are grouped into buckets, based on custom logic, but you do not specify where the buckets reside.
- GemFire always keeps the entries in the buckets you have specified, but may move the buckets around for load balancing.

```
public class OrderKey {  
    private Integer customerId;  
    private Integer orderId;  
    // Implement getters, hashCode and equals  
}
```

```
public class CustomerPartitionResolver implements PartitionResolver,  
Serializable, Declarable {  
    public Serializable getRoutingObject(EntryOperation eo)  
    {  
        OrderKey key = (OrderKey)eo.getKey();  
        return key.getCustomerId();  
    }  
}
```

The customerId portion of the CustomerKey used for bucket calculation

# EntryOperation Object

Your opportunity to understand what the entry is and the circumstance under which it is happening

Return Type	Method	Relevance
Object	getKey()	Returns the Key for the entry
Object	getNewValue()	Not valid here. It may be a get operation.
Operation	getOperation()	The current operation being performed. Includes things like "put if absent", "get", "replace". See the api for <code>com.gemstone.gemfire.cache.Operation</code> for details.
Region	getRegion()	The region that this entry belongs to
boolean	isCallbackArgumentAvailable()	Returns true if a callback argument was supplied on put/get operation
Object	getCallbackArgument()	Returns the callbackArgument passed to the method that generated this event. Returns null if above method returns false.

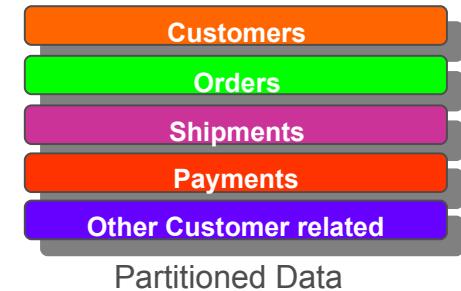
# Lesson Road Map

- Custom Partitioning
- **Co-location**



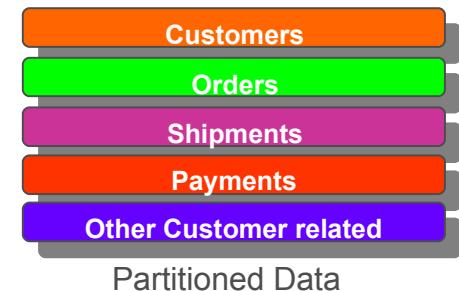
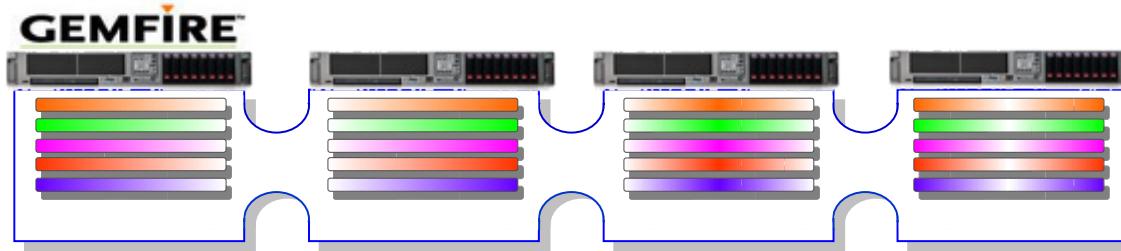
# Co-location of Related Data Items

- Natural Data Relationships
  - Customers have Orders, Shipments and Payments
  - We typically want to access the Shipment, Order and Customer together
- Partition the data by CustomerID for best scalability



# Co-location of Related Data Items

Keep Orders, Shipments and Payments together with the Customer record



- Co-location of related data eliminates the need for distributed transactions
- Partitioned Regions work well for one-to-many and many-to-one relationships

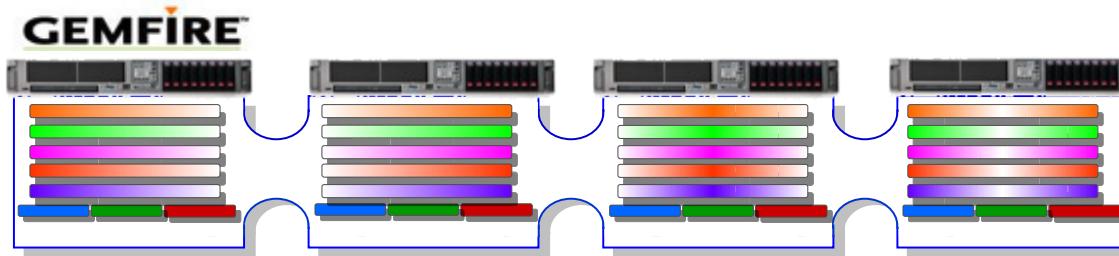
# Many-to-Many Relationships

- Many-to-many relationships usually exist
  - Many customers will look at the same products
  - Pricing data and others may also be accessed by multiple customers



# Many-to-Many Relationships

For best speed we want to have all this reference data accessible in the same JVM as the Customer



- Replicated Regions model many-to-many relationships

# Colocated Regions

- Use to reduce network hops in compute heavy data sets, or iterative operations on related data sets.
- For Custom Partitioned Regions, the PartitionResolvers of the data must be coordinated.

```
<region name="Customers">
    <region-attributes refid="PARTITION_REDUNDANT" />
</region>
<region name="Orders">
    <region-attributes refid="PARTITION_REDUNDANT" >
        <partition-attributes colocated-with="Customers" />
        <partition-resolver name="CustomerPartitionResolver">
            <class-name>myPackage.CustomerPartitionResolver</class-name>
        </partition-resolver>
    </partition-attributes>
</region>
```



# Lab 6

**In this lab, you will**

- 1.Create a Custom Partitioning scheme
- 2.Use gfsh to view how buckets are distributed

# Review of Learner Objectives

**You should be able to do the following:**

- Describe the difference between standard and Custom Partitioning
- Configure Custom Partitioning for GemFire
- Describe Co-location of Regions and when to use it

# Pivotal

BUILT FOR THE SPEED OF BUSINESS

# Cache Management

# Learner Objectives

**After this lesson, you should be able to do the following:**

- Describe how disk stores are used in GemFire
- Understand and configure data expiration in GemFire
- Understand and configure data eviction in GemFire

# How can I manage GemFire memory?

- Region backup
- Expiring stale data
- Efficient memory use with eviction and overflow
- Controlling JVM heap usage with the GemFire resource manager

# Lesson Road Map

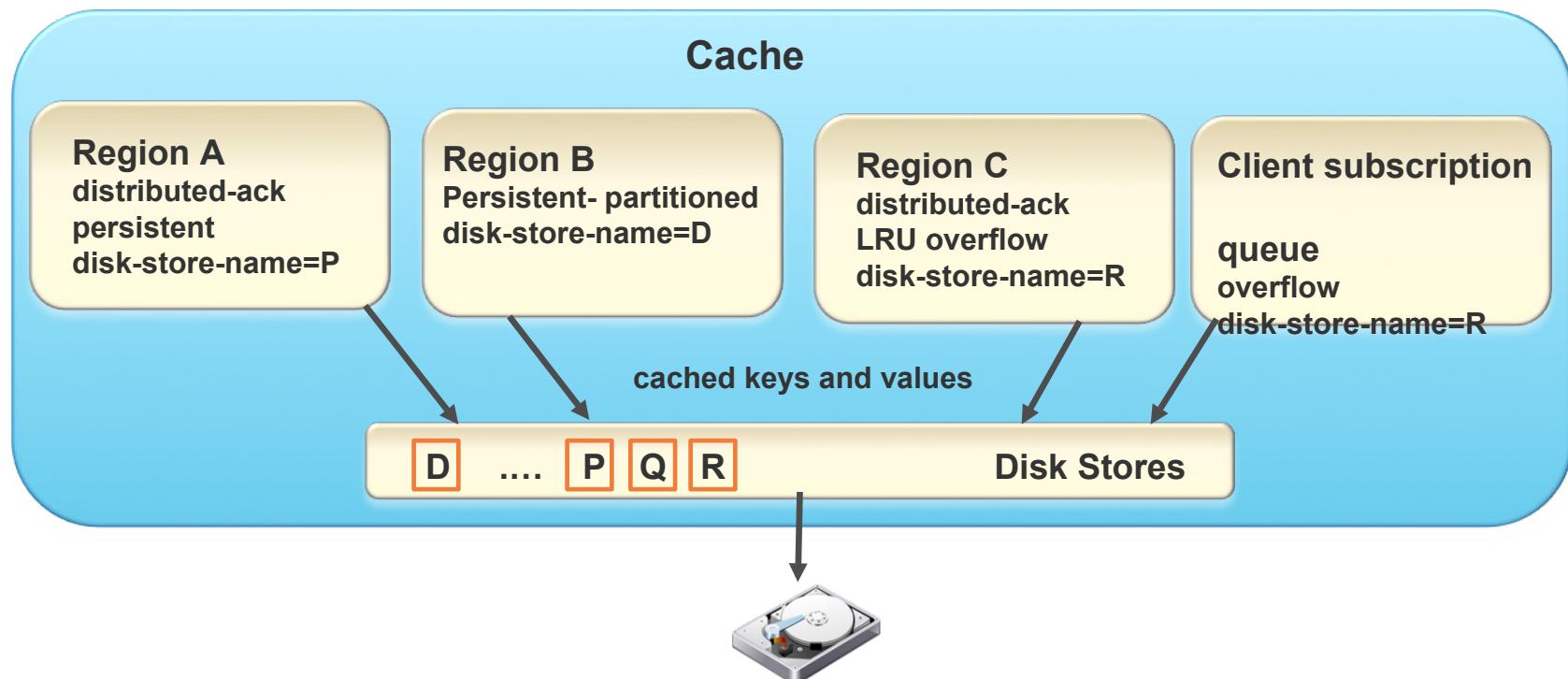
- **Overview of disk stores**
- Expiration
- Eviction
- Resource Manager



# Overview: Disk Stores

Disk storage is available for implementing the following :

- Backup and overflow of the cached data regions
- Overflowing the server's client subscription queues
- Gateway messaging queue persistence



# Overview: Configuring Disk Store

Configures a region to persist its data to disk.  
The data files are written to the directories specified  
in the <disk-dir> elements.

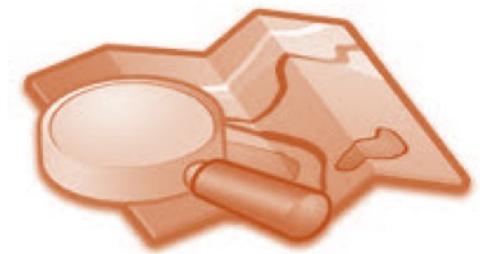
```
<cache>
  <disk-store name="ds1" >
    <disk-dirs>
      <disk-dir >/persistData1</disk-dir>
    </disk-dirs>
  </disk-store>

  <region name="Customer">
    <region-attributes refid="REPLICATE_PERSISTENT"
                       disk-store-name="ds1" disk-synchronous="false">
    </region-attributes>
  </region>
</cache>
```

cache.xml

# Lesson Road Map

- Overview of disk stores
- **Expiration**
- Eviction
- Resource Manager



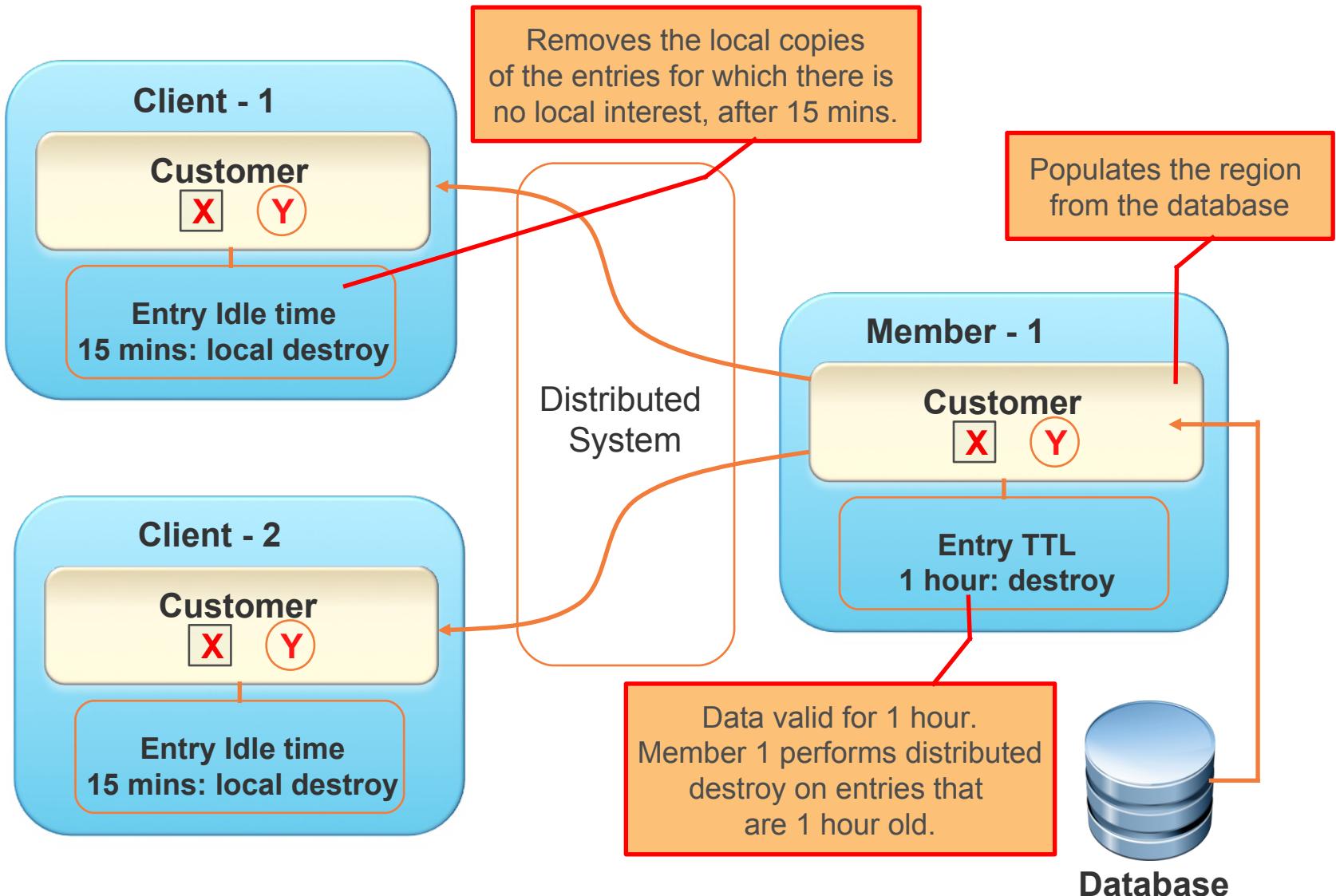
# Data Expiration

- Expiration removes old entries
  - Destroy or Invalidate entries
  - Optimizes memory consumed by Region
- Expiration activities in distributed regions can be distributed or local
  - One cache could control expiration for a number of caches
- Expiration types:
  - Time to live (TTL)
  - Idle timeout
- Expiration actions:
  - destroy
  - local-destroy
  - invalidate (default)
  - local-invalidate

# Partitioned Regions and Expiration

- Executed on the primary copy based on the primary's last accessed and last updated statistics
- Entries may expire sooner than expected
- Primary does not request access statistics from the secondary copies because the performance impact would be too high

# How Expiration Works: Client/Server System



# Configuring Expiration

```
<cache>
  <region name="Customer">
    <region-attributes statistics-enabled="true">
      <entry-idle-time>
        <expiration-attributes timeout="60" action="invalidate"/>
      </entry-idle-time>
    </region-attributes>
  </region>
</cache>
```

`cache.xml`

Set the region's statistics-enabled attribute to true.

Set the expiration attributes by expiration type, with the max times and expiration actions

# Configuring Expiration: Creating Custom Expiration Class

```
package mypackage;
import com.gemstone.gemfire.cache.*;
import static com.gemstone.gemfire.cache.RegionShortcut.*;

public class MyCustomExpiry implements CustomExpiry, Declarable {

    public void init(java.util.Properties props) {
        // cache.xml initialization code goes here
    }

    public ExpirationAttributes getExpiry(Region.Entry entry) {
        if (entry.getKey().equals("alpha")) {
            return new ExpirationAttributes(15, ExpirationAction.INVALIDATE);
        }
        else if (entry.getKey().equals("beta")) {
            return new ExpirationAttributes(3, ExpirationAction.DESTROY);
        }
        else {
            // null tells GemFire to act as if this CustomExpiry task does not exist
            return null;
        }
    }
}
```



Expiration time, in seconds

# Configuring Expiration: Implementing Custom Expiration

```
<cache>
  <region name="Customer">
    <region-attributes statistics-enabled="true">
      <entry-time-to-live>
        <expiration-attributes timeout="0">
          <custom-expiry>
            <class-name>mypackage.MyCustomExpiry</class-name>
          </custom-expiry>
        </expiration-attributes>
      </entry-time-to-live>
    </region-attributes>
  </region>
</cache>
```

`cache.xml`

# Lesson Road Map

- Overview of disk stores
- Expiration
- **Eviction**
- Resource Manager



# Data Eviction

- Data eviction allows entries in a GemFire region to be evicted from memory based on a user defined space limit
- Evicted entries are either destroyed or paged to disk
- The least recently used entry is the one that is evicted

# Eviction: Controlling a Region's Size

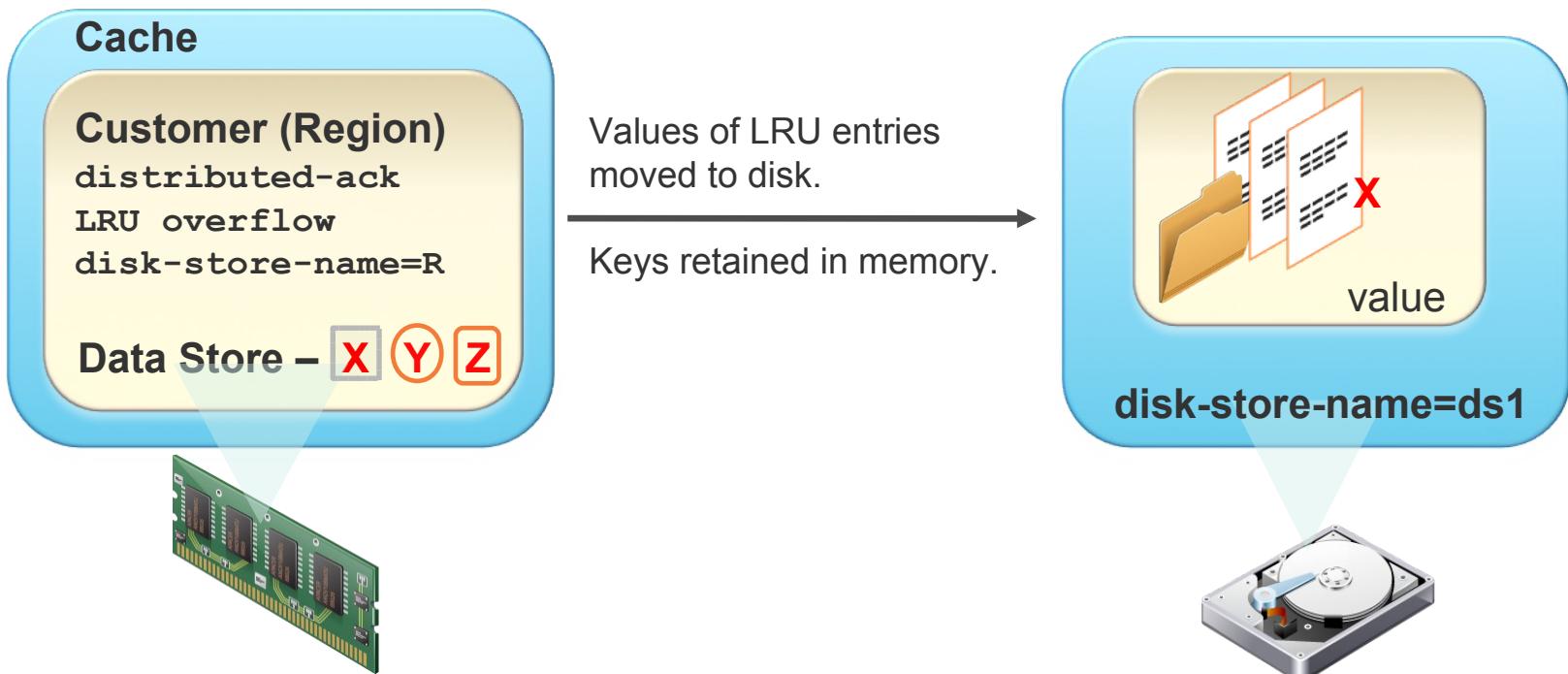
- You use the `<eviction-attributes>` region attribute in `cache.xml` file to control the region's size by removing old entries
- Eviction algorithms:
  - `lru-entry-count`
  - `lru-memory-size`
  - `lru-heap-percentage`
- Eviction actions:
  - `local-destroy`
  - `overflow-to-disk`

# Partitioned Regions and Eviction

- GemFire removes the oldest entry it can find in the bucket where the new entry operation is being performed
- LRU is maintained on a bucket by bucket basis
- Not maintained across the partitioned regions
  - Performance would be impacted
- With heap eviction, each bucket is treated as if it were a separate region, where eviction only works on the bucket, not the region as a whole

# Region Overflow

- Overflow limits region size in memory by moving the values of least recently used (LRU) entries to disk
- For persisted entries, overflowed entries are maintained on the disk just as they are in memory



# Overflow to Disk

```
<region name="Customer">
  <region-attributes disk-store-name="ds1" .../>
    <partition-attributes local-max-memory="1000"
      total-num-buckets="13" />
      <eviction-attributes>
        <lru-memory-size maximum="512"
          action="overflow-to-disk"/>

      </eviction-attributes>
    </region-attributes>
```

cache.xml

# Entry-Count Eviction

```
<cache>
  <region name="Customer">
    <region-attributes>
      <eviction-attributes>
        <lru-entry-count maximum="1000"
                           action="local-destroy" />
      </eviction-attributes>
    </region-attributes>
  </region>
</cache>
```

**cache.xml**

You specify the number of entries to keep in the region.  
If you try to put more in then the oldest entries are evicted.

# Memory size eviction and the ObjectSizer

- Interface:
  - com.gemstone.gemfire.cache.util.ObjectSizer
- The sizer interface defines a method that when called returns the size of the object passed in. (sizeof (Object o))
  - May return hardcoded values for object size if the implementation knows the object size for all objects that are likely to be cached
- Should use a sizer for faster or more accurate method of sizing, other than the default sizer
  - EvictionAttributes.createLRUHeapAttributes (ObjectSizer)
  - EvictionAttributes.createLRUMemoryAttributes (ObjectSizer)

# Memory-Size Eviction

```
<cache>
<region name="Customer">
<region-attributes>
<eviction-attributes>
<lru-memory-size maximum="99" action="local-destroy">
<class-name> mypackage.MyObjectSizer </class-name>
</lru-memory-size>
</eviction-attributes>
</region-attributes>
</region>
</cache>
```

cache.xml

```
public class MyObjectSizer implements ObjectSizer, Declarable {
    public void init(java.util.Properties props) {
        /* initialization code goes here for cache.xml support */
    }
    public int sizeof(Object o) {
        /* simple implementation that knows all instances are 4k */
        return 4*1024;
    }
}
```

Custom Java Class

# Lesson Road Map

- Overview of disk stores
- Expiration
- Eviction
- **Resource Manager**



# Resource Manager

- Manages overall memory in a cacheserver, not just a Region
- Should only use in application where you run the risk of running out of memory
- Two threshold settings
  - Eviction threshold
    - Manager order evictions for all regions with eviction-attributes set to lru-heap-percentage
    - JVM garbage collector removes evicted data
    - Evictions continue until the manager see that heap usage is below the threshold
    - Regular cacheserver activity continues, but new entries must have an equal eviction
  - Critical – set above the eviction threshold
    - Cache activity is refused for anything that may add to heap consumption
    - To allow the garbage collector to catch up
    - Clients may receive a `LowMemoryException` - retry

# Relationship to JVM garbage collection

- Configure GemFire for LRU heap management
  - Set the critical-heap-percentage threshold
    - Should be as close to 100 as possible without getting OOM
    - Default is zero – no threshold
  - Set the eviction-heap-percentage
    - Lower than critical
    - 50 is a good starting point
  - Choose Regions that will participate by setting their eviction attributes to lru-heap-percentage
- Set the JVM GC tuning parameters in relation to the LRU
- Monitor, tune, test!!!!

```
<cache>
    <resource-manager critical-heap-percentage="75"
                      eviction-heap-percentage="50"/>
    ...
    <region-attributes refid="REPLICATE_HEAP_LRU"/>
</cache>
```

# Resource Manager - Java usage

```
Cache cache = CacheFactory.create();

ResourceManager rm = cache.getResourceManager();
rm.setCriticalHeapPercentage(85);
rm.setEvictionHeapPercentage(75);

RegionFactory rf = cache.createRegionFactory
    (RegionShortcut.PARTITION_HEAP_LRU);

Region region = rf.create("bigDataStore");
```

# Controlling Heap Use With the Resource Manager

The GemFire resource manager:

- Enables controlling the JVM heap usage, and protects the VM from hangs and crashes due to memory overload by implementing garbage collections (GC) to evict old data
- **Uses** `EvictionHeapPercentage` **and** `CriticalEvictionHeapPercentage` **thresholds**

Configures Java VM for heap LRU eviction

```
java -Xms512M -Xmx512M -XX:+UseConcMarkSweepGC  
      -XX:CMSInitiatingOccupancyFraction=N
```

where `N < EvictionHeapPercentage`

- Monitor performance via VSD

# Heap LRU Eviction

```
<cache>
    <resource-manager critical-heap-percentage="85"
                      eviction-heap-percentage="70" />
    <region name="Customer">
        <region-attributes refid="REPLICATE_HEAP_LRU">
            <eviction-attributes>
                <lru-heap-percentage action="local-destroy" />
            </eviction-attributes>
        </region-attributes>
    </region>
</cache>
```

`cache.xml`

# Lab

**In this lab, you will**

- 1.**Configure cache expiration
- 2.**Configure cache eviction

# Review of Learner Objectives

**You should be able to do the following:**

- Describe how disk stores are used in GemFire
- Understand and configure data expiration in GemFire
- Understand and configure data eviction in GemFire

# Pivotal

BUILT FOR THE SPEED OF BUSINESS

# Configuring Server-side Event Handling

# Learner Objectives

**After this lesson, you should be able to do the following:**

- Describe the GemFire event framework
- Describe the types of event in the framework
- Implement cache event handlers:
  - Cache listener
  - Cache writer
  - Cache loader
  - AsyncEventListener

# Lesson Road Map

- **The GemFire Event Framework**
- Types of Event
- Implementing Cache Loader
- Implementing Cache Writer
- Implementing Cache Listener
- Implementing AsyncEventListener



# What are GemFire Events?

There are two Categories of events:

- Cache events:

- Provides detail-level notification for changes to data
- Are processed by event handlers that are running in the cache and its Regions

- Administrative events:

- Provides notification on the health and status of the members, caches, and regions
- Are processed with event handlers that are running in any administrative application, which is not a part of cache



You cannot handle both cache and administrative events in a single member

# The GemFire Event Framework

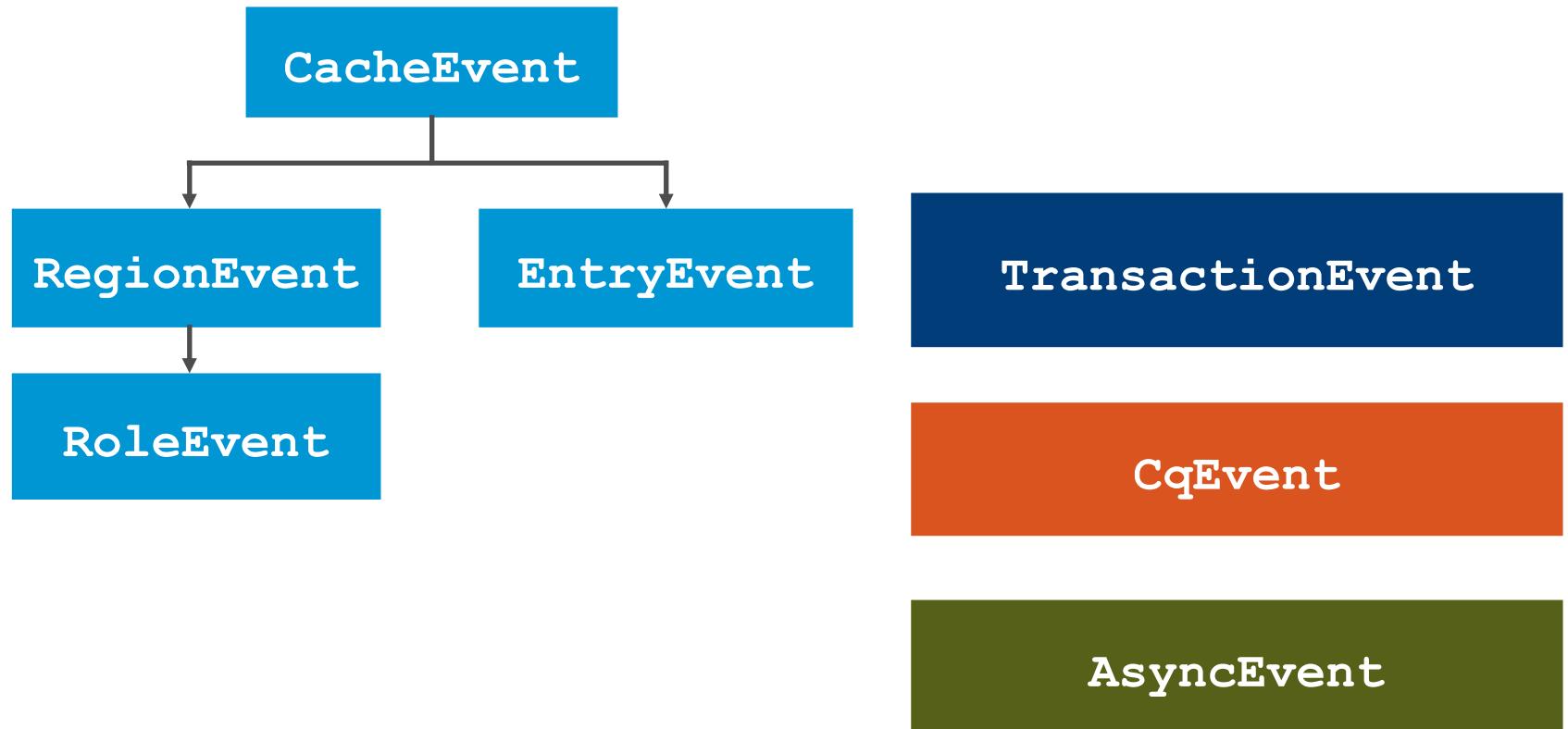
- GemFire operations, such as data entry or cache close, generate events:
  - An Operation object
  - An event object
- The event handlers are invoked, based on the type of events fired
- The event handler APIs are defined in  
`com.gemstone.gemfire.cache` package

# Lesson Road Map

- The GemFire Event Framework
- **Types of Event**
- Implementing Cache Listener
- Implementing Cache Writer
- Implementing Cache Loader
- Implementing AsyncEventListener



# Types of Event



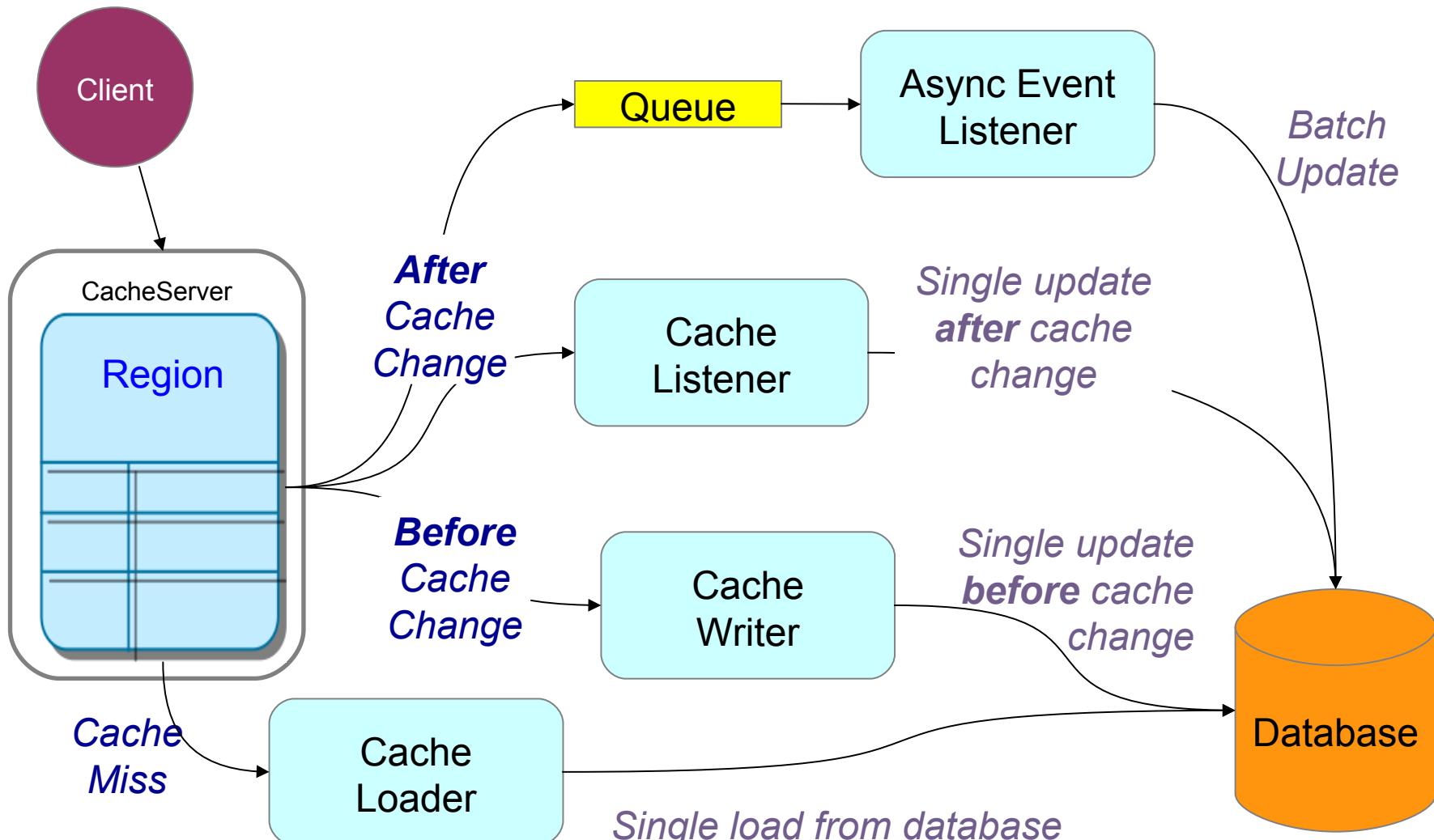
# Event API: CacheEvent

- The CacheEvent interface contains information about events affecting a region, or entry-related event affecting the cache
- Important methods:
  - getRegion()
  - getOperation()
  - getCallbackArgument()
  - getDistributedMember()

# Event API: EntryEvent

- The `EntryEvent` interface contains information about an event affecting an entry, including its identity and the circumstances of the event
- Important methods:
  - `getKey()`
  - `getNewValue()`
  - `getOldValue()`

# Gemfire Event Handlers Overview



# Which Event Listeners Do What?

- CacheWriter
  - This is intended to be where you place any data validation logic such as edit-checks etc
  - It is called before create, before update, before destroy, before region destroy or before region clear
  - If it throws a CacheWriterException the operation in progress will be aborted and no change made to the cache
  - Only one instance of a CacheWriter for region within the distributed system will be called
    - Replicated Region: only member contacted by client
    - Partitioned Region: member having primary bucket

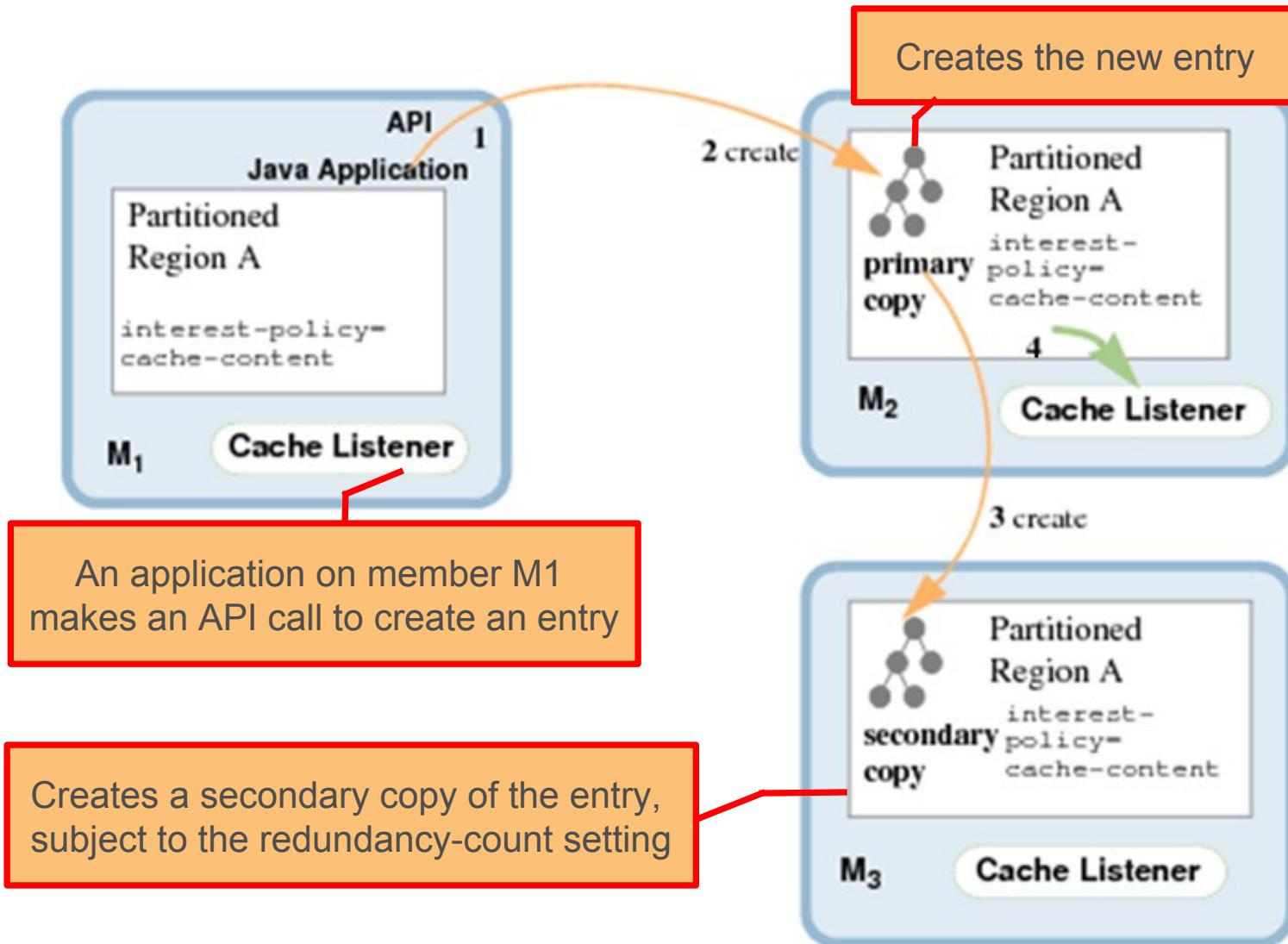
# Which Event Listeners Do What?

- CacheLoader
  - This is called on a cache-miss
  - Its purpose is to perform whatever action is required to fetch the missing entry from a backing store or wherever
  - A local CacheLoader will always be invoked if one exists  
Otherwise one remote loader is invoked
  - If more than one CacheLoader invocation occurs for the same key the latter ones will be held back until the first one completes. This way the backing store will not be hit multiple times since the first one will satisfy all of them

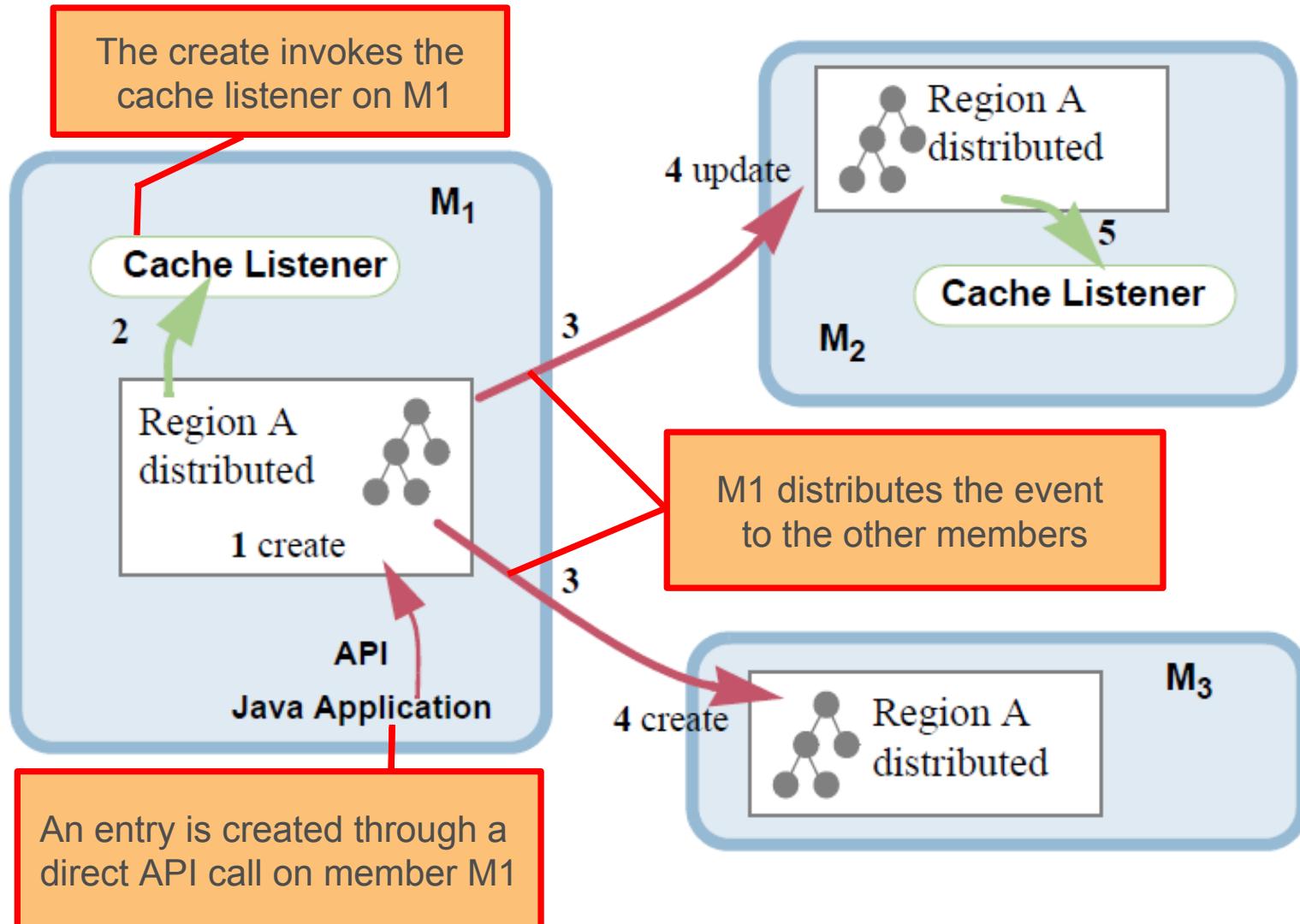
# Which Event Listeners Do What?

- CacheListener
  - This is called after a Create, Update, Destroy or Invalidate has occurred for an Entry
  - This is also called after a RegionClear, RegionCreate, RegionDestroy, RegionInvalidate, RegionLive occurs
  - Its main purpose is to provide notification that something has occurred so that the application code may perform post processing
  - This Listener will be called on every member who has registered interest on the Region involved

# Events in Partitioned Regions



# Events in Replicated Regions



# Implementing Cache Event Handlers

1. Identify the events your application needs to recognize.  
For example, EntryEvent event for handling a new entry
2. Identify the event handlers to handle the event  
For example:

```
public class MyListener implements CacheListener {  
    public void afterCreate(EntryEvent e) {  
        } ...  
}
```

1. Implement the event handler by overwriting the afterCreate(EntryEvent e) method

```
public void afterCreate(EntryEvent e) {  
    //code to handles the event of the new key added to region  
    } ...  
}
```

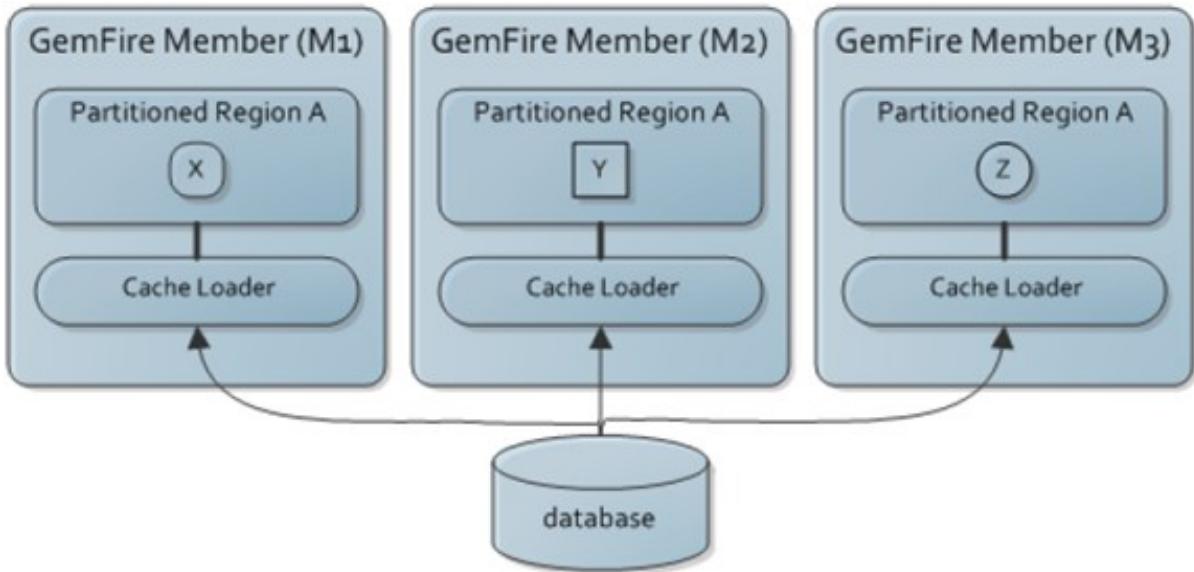
# Lesson Road Map

- The GemFire Event Framework
- Types of Event
- **Implementing Cache Loader**
- Implementing Cache Writer
- Implementing Cache Listener
- Implementing AsyncEventListener

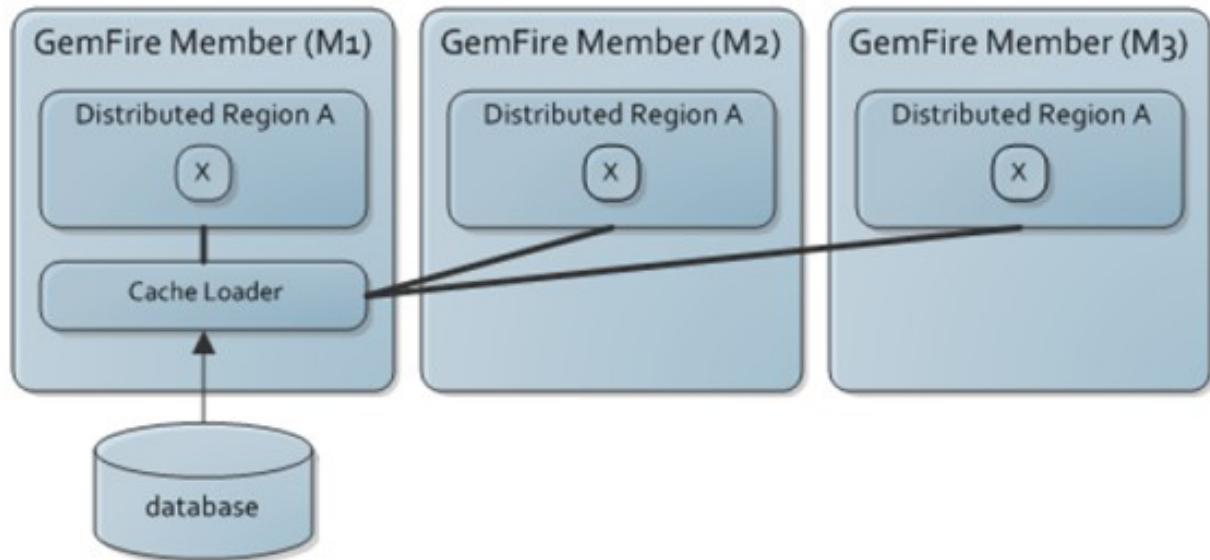


# Overview of Cache Loader

- Data loading in partitioned regions:

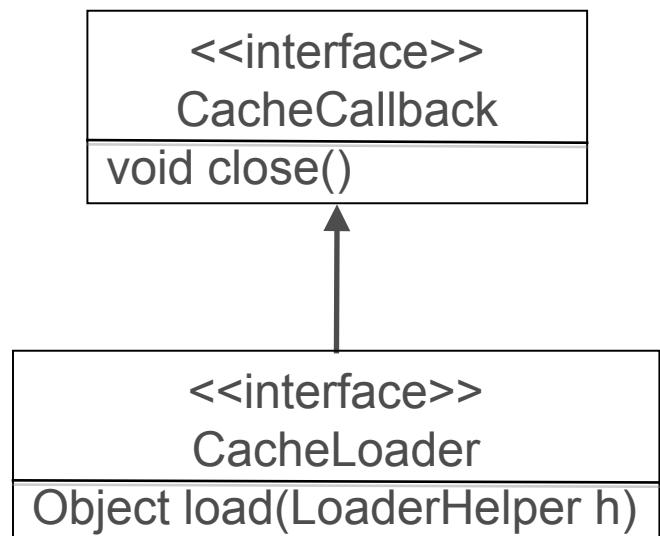


- Data loading in replicated regions:



# Event Handler API: CacheLoader

- The CacheLoader event handler:
  - Provides capabilities to load data into the region on cache misses
  - Contains a single `load` method that you implement for retrieving the data from outside the cache
  - **Synchronous**



# LoaderHelper API

Method	Description
getArgument()	Returns the callback argument that may have been passed as part of the method call that triggered the event.
getKey()	Returns the key associated to the entry being loaded
getRegion()	Returns the region where the entry belongs

*Not a complete list*

## Using Callback Arguments

- Can be used in combination with the key to load the data
- Could be discrete item or collection, etc
- Up to caller and CacheLoader load() implementation to agree

```
// Second argument is the callbackArg  
Customer c = customersRegion.get(123, "additional info");
```

# Implementing a Cache Loader

```
package io.pivotal.demo;

import com.gemstone.gemfire.cache.CacheLoader;
import com.gemstone.gemfire.cache.LoaderHelper;

public class SampleLoader implements CacheLoader {
    public Object load(LoaderHelper helper) throws
        CacheLoaderException {
        System.out.println("Loading " + helper.getKey());
        //code to load data
        return entryValue;
    }
    public void close() {}
}
```

# Installing a Cache Loader

```
<cache>
  <region name="Customer">
    <region-attributes>
      <cache-loader>
        <class-name>
          io.pivotal.demo.SampleLoader
        </class-name>
      </cache-loader>
    </region-attributes>
  </region>
</cache>
```

serverCache.xml

# Passing Information To Handlers

GemFire provides two general purpose mechanisms for passing configuration information to Event Handlers and other pluggable components

- XML Parameters

- Any pluggable component registered in XML MUST implement Declarable
- Use <parameter> XML elements to configure instance
- One-time configuration

- Callback Arguments

- Additional information passed in from clients
- Many cache & region calls have overloaded method
- Ex: put(key, value, callbackArgument)

# Parameterizing an Event Handler

```
<region name="Customer">  
    ...  
    <cache-loader>  
        <class-name>io.pivotal.demo.SampleLoader</class-name>  
        <parameter name="url">  
            <string>https://myservices.com/myservice</string>  
        </parameter>  
        <parameter name="username">  
            <string>gfadmin</string>  
        </parameter>  
        <parameter name="password">  
            <string>admin1</string>  
        </parameter>  
    </cache-loader>  
    ...
```

serverCache.xml

Pivotal™

# Declarable Interface

- Required to be implemented on anything that will be specified in the `serverCache.xml` file.
- `init(Properties p)` method allows you to initialize attributes of the class.

# Using the parameters

```
public class SampleLoader implements CacheLoader, Declarable {  
  
    public Object load(LoaderHelper helper) throws  
        CacheLoaderException {  
  
        . . .  
        // Initialize the Messaging service/Web service using input  
        // parameters  
  
        WebService.getConnection(url, username, password);  
  
        . . .  
    }  
    . . .  
  
    public void init(Properties props) {  
        this.url = props.getProperty("url");  
        this.username = props.getProperty("username");  
        this.password = props.getProperty("password");  
    }  
}
```

# Using CallbackArgument

```
public class SampleLoader implements CacheLoader, Declarable {  
  
    public Object load(LoaderHelper helper) throws  
        CacheLoaderException {  
  
        . . .  
        // Initialize a parameter used to help with fetch -  
        // provided as additional argument in get() from client.  
        String arg = (String) helper.getArgument();  
  
        . . .  
    }  
    . . .  
}
```

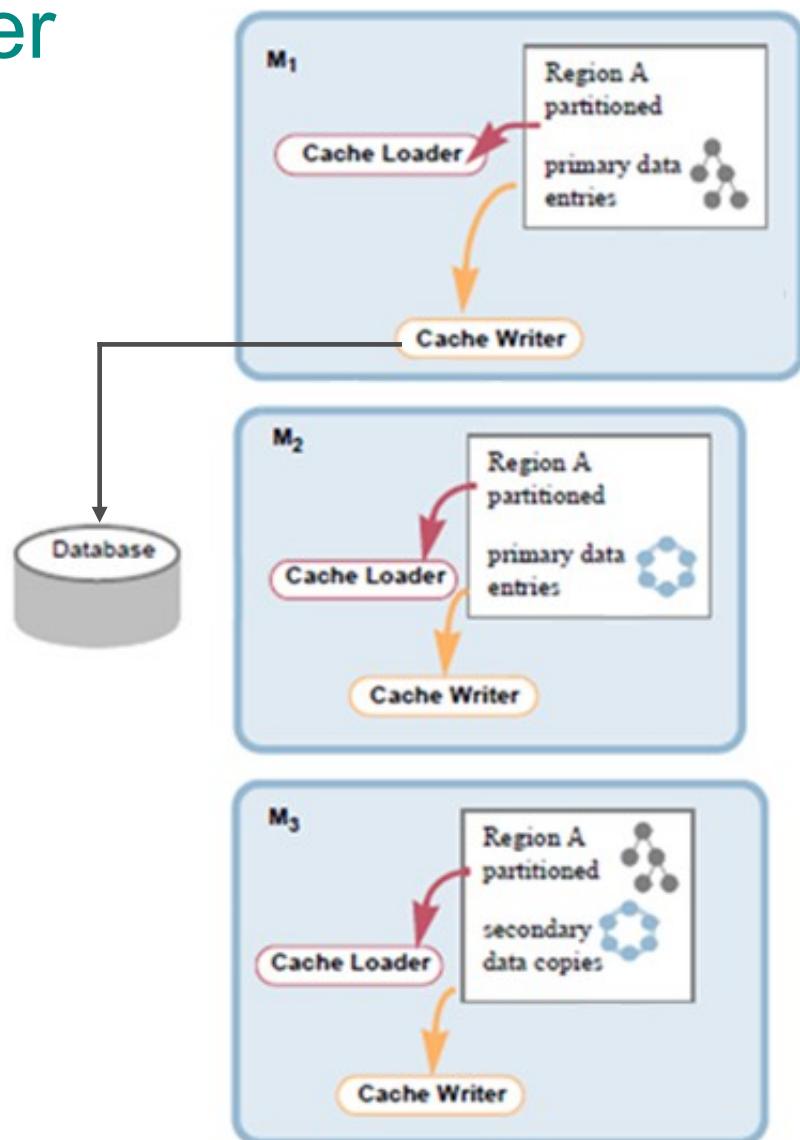
# Lesson Road Map

- The GemFire Event Framework
- Types of Event
- Implementing Cache Loader
- **Implementing Cache Writer**
- Implementing Cache Listener
- Implementing AsyncEventListener



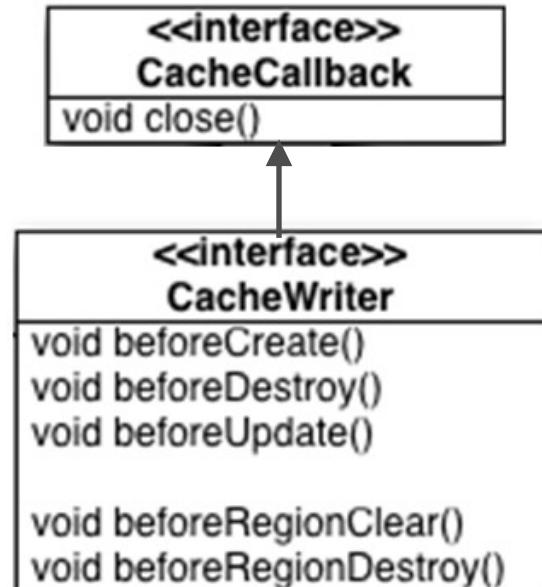
# Overview of Cache Writer

- Receives events for *pending* changes to the region and its data entries in this member or one of its peers
- Has the ability to abort the operations in question
- In distributed regions, a cache writer may be invoked remotely from other members that have the region defined
- In partitioned regions, the system only uses the cache writer in the primary data host for the event
- **Synchronous**



# Event Handler API: CacheWriter

- The CacheWriter event handler
  - Provides write-through caching capabilities with an external data source
  - Is implemented in the CacheWriterAdapter class
- Install writers where you need them:
  - For partitioned regions, the writer installed in the primary data host is called
  - For replicated regions, the closest available writer in the distributed system is called
  - For local regions, the cache writer in the local cache is called



# Event Handler API: CacheWriterAdapter

```
Package io.pivotal.demo;

import com.gemstone.gemfire.cache.*;
import com.gemstone.gemfire.cache.util.*;

public class SampleWriter extends CacheWriterAdapter {

    @Override
    public void beforeCreate(EntryEvent event) throws CacheWriterException {
        // Perform some validation - throw CacheWriterException if validation fails
    }

    @Override
    public void beforeUpdate(EntryEvent event) throws ... {
        System.out.println("Updating " + event.getOldValue() + " to " +
                           event.getNewValue());
    }

    // Other methods could be overridden as well
}
```

# Installing a Cache Writer

```
<cache>
  <region name="Customer">
    <region-attributes>
      <cache-writer>
        <class-name>
          io.pivotal.demo.SampleWriter
        </class-name>
      </cache-writer>
    </region-attributes>
  </region>
</cache>
```

serverCache.xml

# Installing a Cache Writer: Java Code

```
Package io.pivotal.demo;

import com.gemstone.gemfire.cache.*;
import com.gemstone.demo.SampleListener;

public class createRegion ...{

    public static void main(String args[]){

        ...

        RegionFactory rf =
            cache.createRegionFactory(RegionShortcut.PARTITION);
        rf.setCacheWriter(new SampleWriter());
        custRegion = rf.create("Customer");
        ...

    }

}
```

# Lesson Road Map

- The GemFire Event Framework
- Types of Event
- Implementing Cache Loader
- Implementing Cache Writer
- **Implementing Cache Listener**
- **Implementing AsyncEventListener**

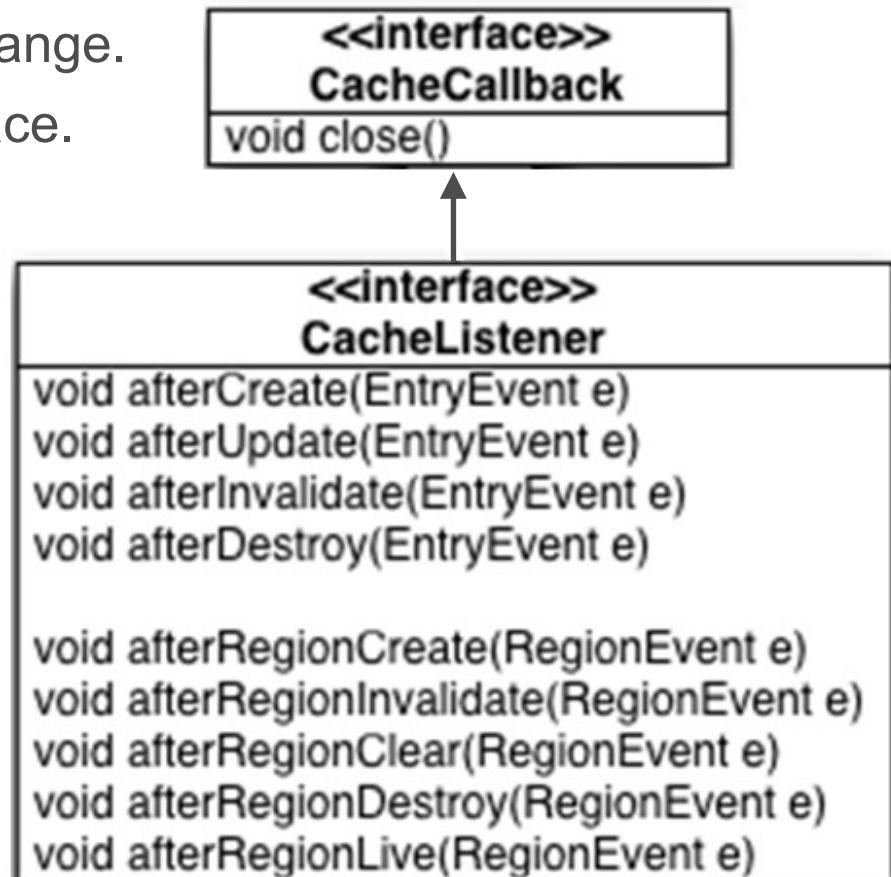


# Event Handler API: CacheListener

The CacheListener event handler:

- Tracks changes to the region and its data entries.
- Responds **synchronously** to any change.
- Extends the CacheCallback interface.
- Is installed on the region.
- Is implemented in the

CacheListenerAdapter class.



# Event Handler API: CacheListenerAdapter

```
Package io.pivotal.demo;

import com.gemstone.gemfire.cache.*;
import com.gemstone.gemfire.cache.util.*;

public class SampleListener extends CacheListenerAdapter {

    @Override
    public void afterCreate(EntryEvent e)
        {System.out.println("Created " + e.getNewValue());
    }

    @Override
    public void afterUpdate(EntryEvent e)
        {System.out.println("Updated " + e.getOldValue());
    }
    // Other methods could be overridden as well
}
```

# Installing a Cache Listener

```
<cache>
  <region name="Customer">
    <region-attributes>
      <cache-listener>
        <class-name>
          io.pivotal.demo.SampleListener
        </class-name>
      </cache-listener>
    </region-attributes>
  </region>
</cache>
```

serverCache.xml

# Installing a Cache Listener: Java Code

```
package io.pivotal.demo;

import com.gemstone.gemfire.cache.*;
import io.pivotal.demo.SampleListener;

public class createRegion ...{

    public static void main(String args[]) {

        ...
        RegionFactory rf =
            cache.createRegionFactory(RegionShortcut.PARTITION);
        rf.addCacheListener(new SampleListener());
        custRegion = rf.create("Customer");
        ...
    }
}
```

# Lesson Road Map

- The GemFire Event Framework
- Types of Event
- Implementing Cache Loader
- Implementing Cache Writer
- Implementing Cache Listener
- **Implementing AsyncEventListener**



# Event Handler API: AsyncEventListener

The `AsyncEventListener` event handler:

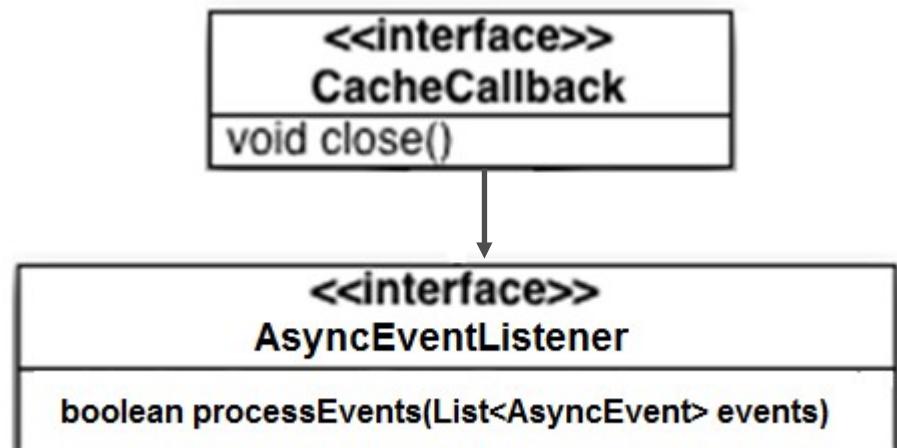
- Tracks changes to the cache, its regions, and its data entries
- Good for ‘Write-Behind’
- Responds asynchronously to changes in batch
- Parallel or Serial
- Extends the `CacheCallback` interface
- Is installed on the cache
- Is tied to one or more regions

```
public boolean processEvents(List events);
```

# Event Handler API: AsyncEventListener

The AsyncEventListener event handler:

- Provides a write-behind cache capabilities to synchronize region updates with a database
- Responds asynchronously to any change
- Extends the CacheCallback interface
- Is installed on an AsyncEventQueue
  - Serial
  - Parallel
- Queue associated to a region



# Event Handler API: AsyncEventListener

```
class MyAsyncEventListener implements AsyncEventListener {  
    public void processEvents(List<AsyncEvent> events) {  
        // Process each AsyncEvent  
        ...  
        for(AsyncEvent event: events) {  
            // Write the event to a database  
        }  
    }  
}
```

# AsyncEvent API

- Very similar to Cache Event
  - Has getCallbackArgument, getKey, getOperation, etc.
  - Adds additional methods

Method	Description
getEventSequenceId	Gets a wrapper class containing information on the event sequence
getPossibleDuplicate	Is this a possible duplicate (ex HA Function)?

- From GatewayQueueEvent base class

Method	Description
getDeserializedValue	Returns the deserialized value
getSerializedValue	Returns the serialized value as a byte array

# Configuring an AsyncEventListener

```
<cache>

    <async-event-queue id="sampleQueue" persistent="true"
        disk-store-name="exampleStore" parallel="false">
        <async-event-listener>
            <class-name>io.pivotal.MyAsyncEventListener</class-name>
        </async-event-listener>
    </async-event-queue>

    <region name="Customer">
        <region-attributes async-event-queue-ids="sampleQueue"/>
    </region>
    ...
</cache>
```

serverCache.xml

Pivotal™

# Lab

**In this lab, you will**

1. Configure a cache writer to perform validation
2. Configure a cache loader to load data in cache.

# Review of Learner Objectives

You should be able to do the following:

- Describe the GemFire event framework.
- Describe the types of event in the framework.
- Implement cache event handlers:
  - Cache listener
  - Cache writer
  - Cache loader
  - AsyncEventListener

# Pivotal

BUILT FOR THE SPEED OF BUSINESS

# Client-side Event Handling

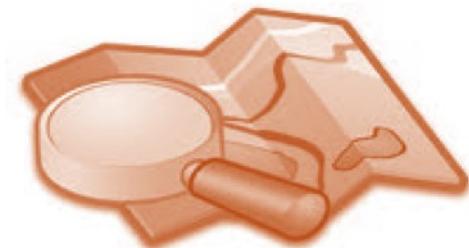
# Learner Objectives

**After this lesson, you should be able to do the following:**

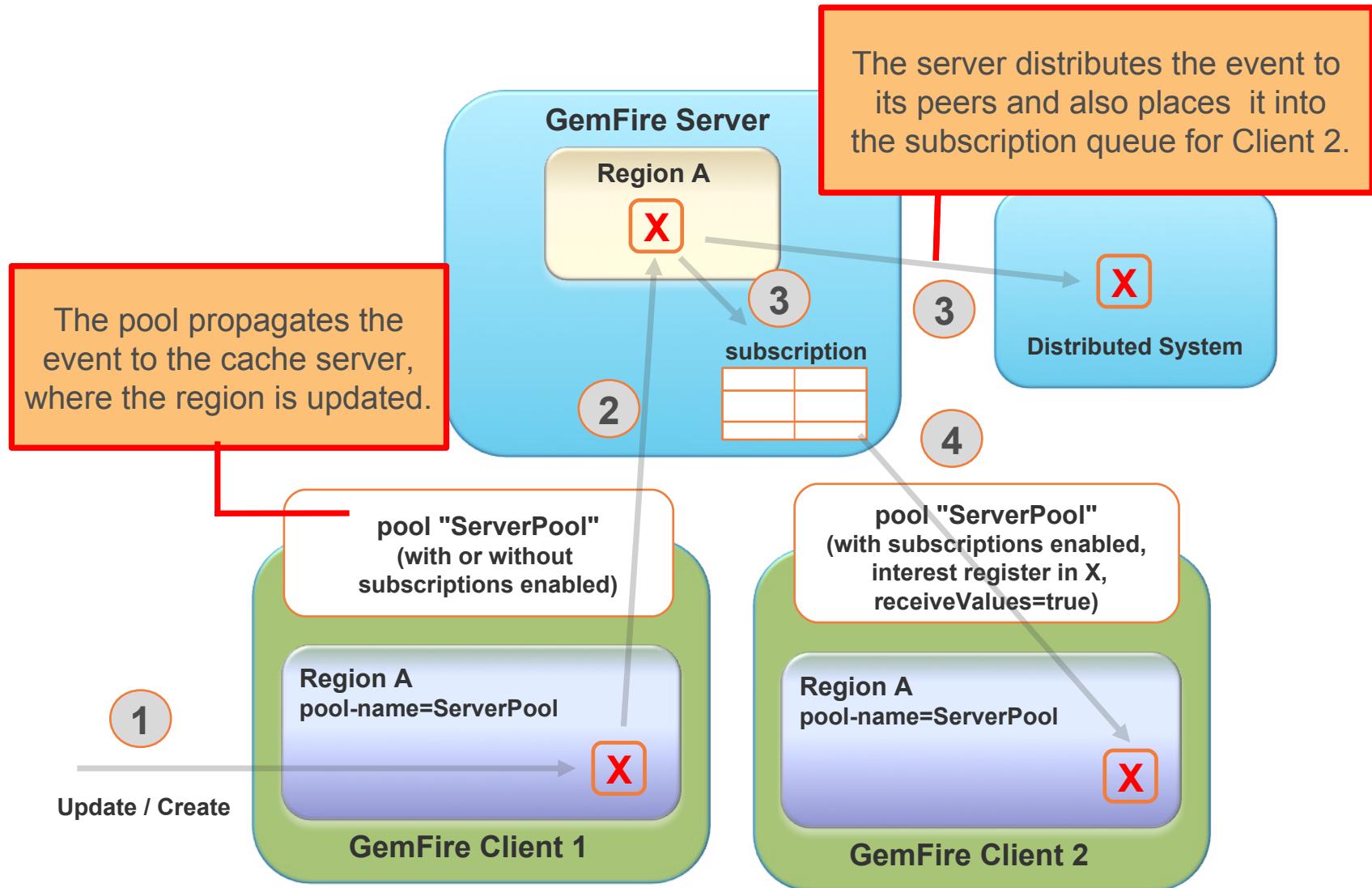
- Describe the types of client-side events in the framework
- Implement a CacheListener
- Register interest in receiving values & events
  - Register a simple Continuous Query

# Lesson Road Map

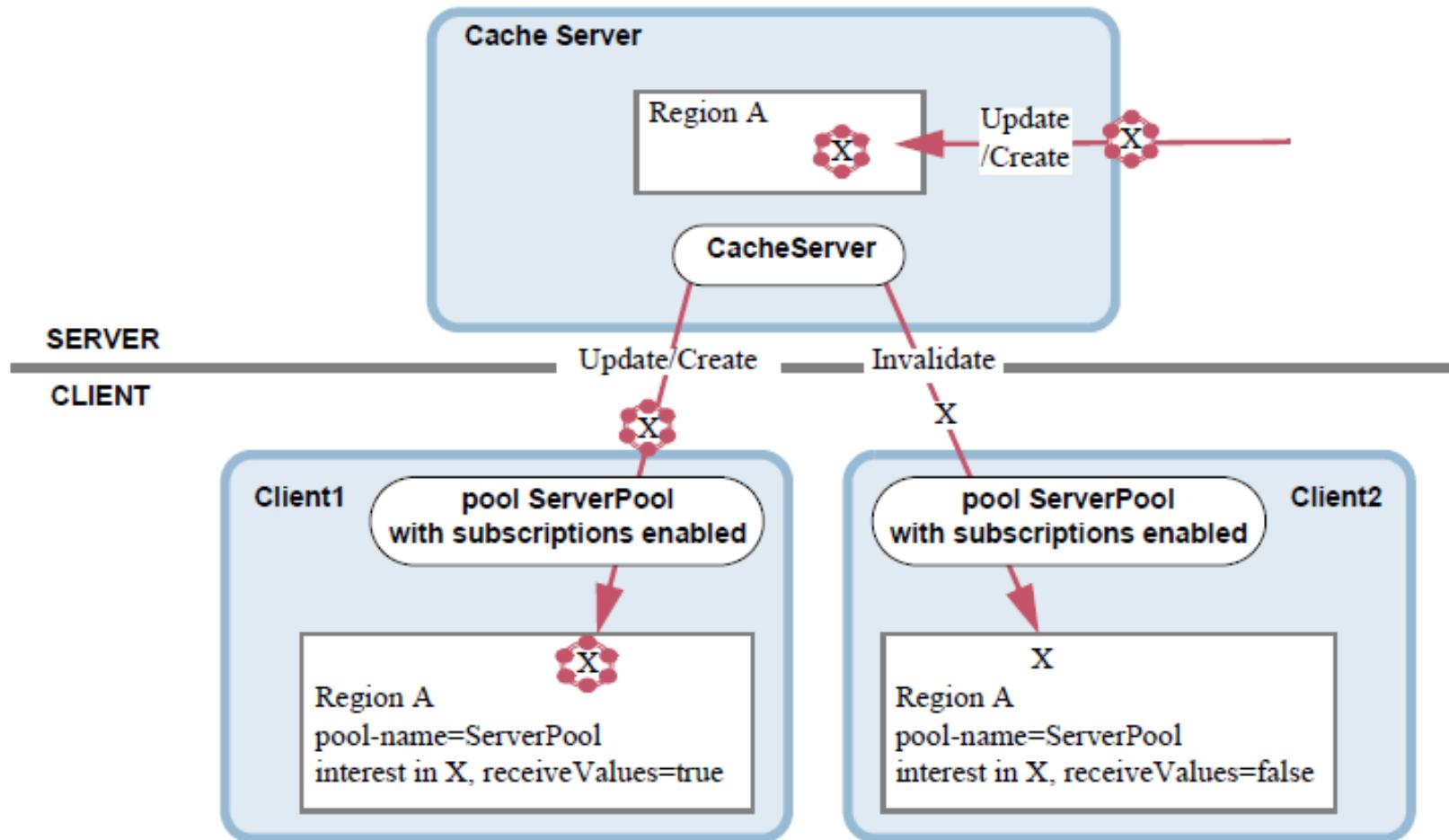
- **GemFire Client Events**
- Registering Interest
- Implementing an Event Listener
- Continuous Queries



# Client/Server Event Distribution

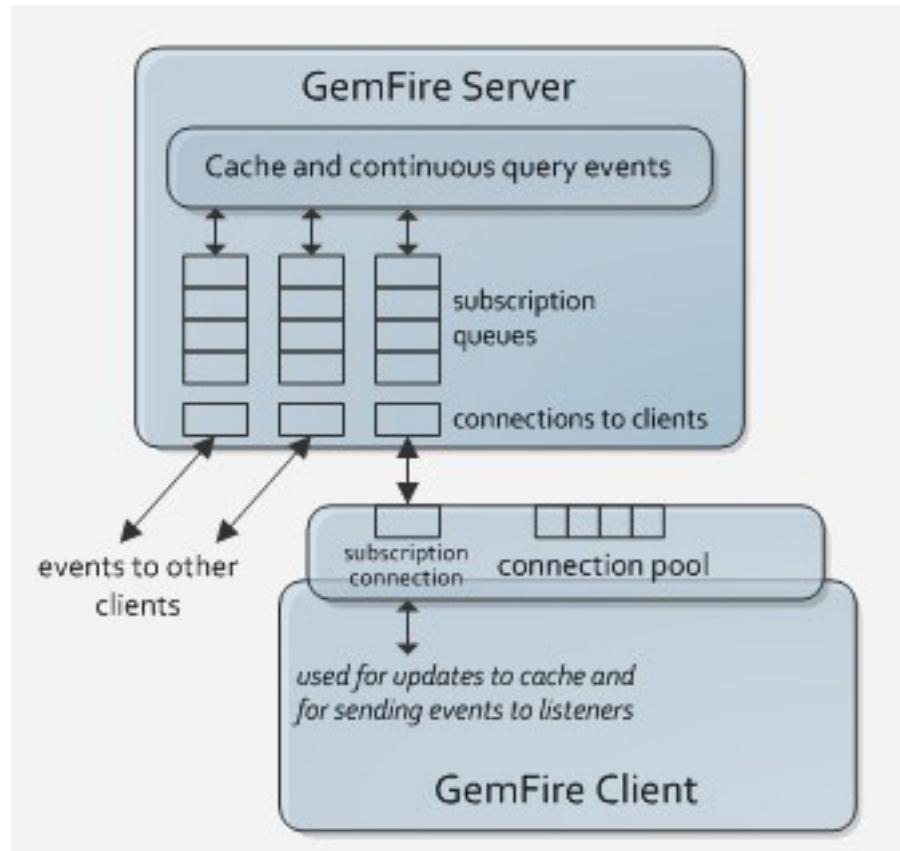


# Server-Initiated Data Flow



# Client Subscriptions

- Used to stream events from server back to client
  - Requires pool definition set subscription-enabled property to true
  - Establishes a subscription queue per client connection
  - Events sent back to client asynchronously – sharing the same client connection
    - Java Event Framework
    - Not polling



# Subscribing to Server Events

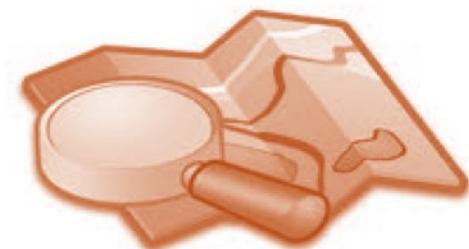
Setting subscriptions-enabled = true establishes client subscription queue for the connection

```
<client-cache>
    <pool name="ServerPool" subscription-enabled="true">
        <locator host="host3" port="41111">
    </pool>
    <region name="Customer">
        <region-attributes refid="PROXY"
                           pool-name="ServerPool">
        </region-attributes>
    </region>
<client-cache>
```

clientCache.xml file

# Lesson Road Map

- GemFire Client Events
- **Registering Interest**
- Implementing an Event Listener
- Continuous Queries



# Configuring Client to Receive Notifications

1. Set subscriptions-enabled attribute in pool definition

```
<client-cache>

    <pool name="ServerPool" subscription-enabled="true">
        <locator host="host3" port="41111">
    </pool>

    ...
<client-cache>
```

2. Write client to register interest

```
public class ClientInterestRegistrar {
    public void registerInterest() {
        // Get the Region reference
        Region customers = ...
        // Register interest in 'Key1'
        customers.registerInterest("Key1");
    }
}
```

# Registering Interest

- Two key aspects to registering interest
  - What key(s) are you interested in?
  - Interest Policy using InterestResultPolicy enumeration
    - NONE: Fire event but don't send key or value
    - KEYS: Initializes local cache with matching keys
    - KEYS\_VALUES: Initializes local cache with keys and values
    - Default if not specified is KEYS\_VALUES
- Initial call to registerInterest() causes initial initialization of cache (usually) as well as updates on matching keys
  - Updates on matching keys also sent to local cache per interest policy

# Ways to Register Interest

- Individual entries

```
// Register interest in 'Key1'  
customers.registerInterest("Key1");  
  
// Register interest in a list of keys  
// If Keys is a list, all keys in the list are sent  
List keys = new ArrayList();  
keys.add("Key1");  
keys.add("Key2");  
customers.registerInterest(keys, InterestResultPolicy.KEYS);
```

- All entries for all keys

```
// Register interest all keys - interest policy can be included  
customers.registerInterest("ALL_KEYS");
```

- Entries matching a regular expression

```
// Register interest in keys matching pattern  
customers.registerInterestRegEx("[a-zA-Z]+_[0-9]+");
```

Can only use with keys of type java.lang.String

# Unregistering Interest

- You unregister in much the same way you can register
- Unregister key or keys

```
// Unregister interest in 'Key1'  
customers.unregisterInterest("Key1");  
  
// Unregister interest in a list of keys  
List keys = new ArrayList();  
keys.add("Key1");  
keys.add("Key2");  
customers.unregisterInterest(keys);  
  
// Unregister interest in all keys  
customers.unregisterInterest("ALL_KEYS");
```

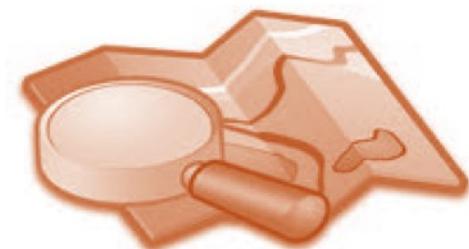
- Unregister matching pattern

```
// Unregister interest in keys matching pattern  
customers.unregisterInterestRegEx("[a-zA-Z]+_[0-9]+");
```

Can only use with keys  
of type java.lang.String

# Lesson Road Map

- GemFire Client Events
- Registering Interest
- **Implementing an Event Listener**
- Continuous Queries



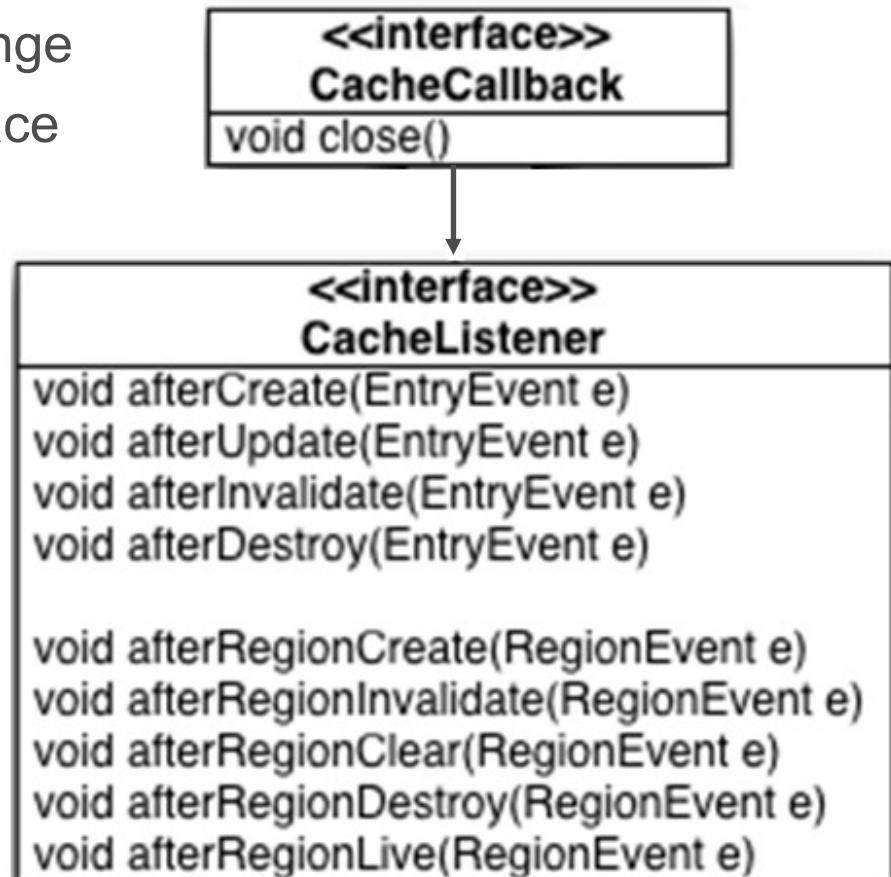
# The CacheListener

- Called after a Create, Update, Destroy or Invalidate has occurred for an Entry
- Also called after a RegionClear, RegionCreate, RegionDestroy, RegionInvalidate, RegionLive occurs
- Main purpose is to provide notification that something has occurred so that the application code may perform post processing
- Will be called on every client who has registered interest on the Region involved AND registered CacheListener

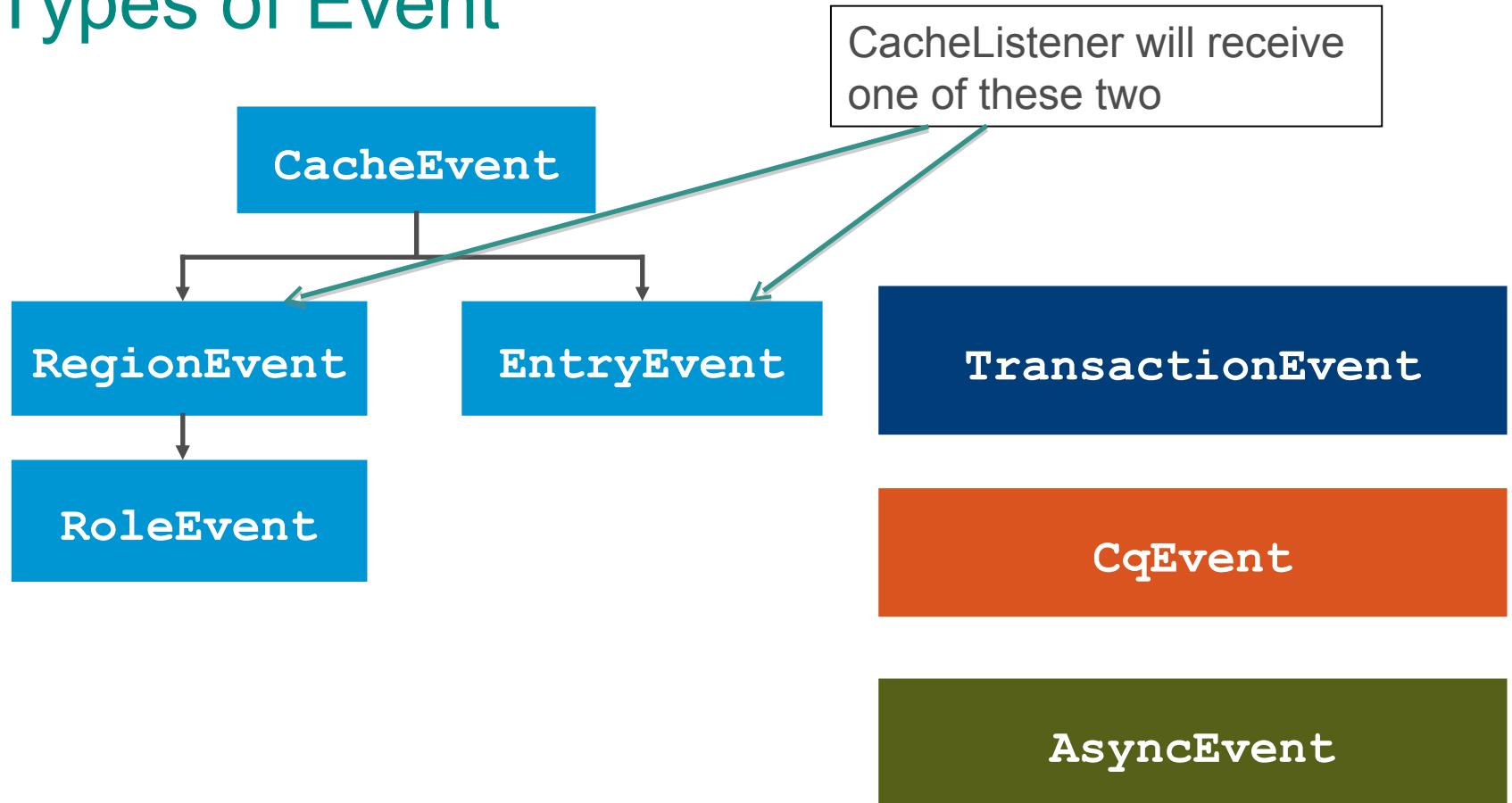
# Event Handler API: CacheListener

The CacheListener event handler:

- Tracks changes to the region and its data entries
- Responds synchronously to any change
- Extends the CacheCallback interface
- Is installed on the region
- Is implemented in the CacheListenerAdapter class



# Types of Event



# Event API: RegionEvent

- The RegionEvent interface contains information about events affecting a region, or entry-related event affecting the cache
- Important methods:
  - getRegion
  - getOperation
  - getCallbackArgument
  - getDistributedMember

# Event API: EntryEvent

- The `EntryEvent` interface contains information about an event affecting an entry, including its identity and the circumstances of the event
- Important methods:
  - `getKey`
  - `getNewValue`
  - `getOldValue`

# Event Handler API: CacheListenerAdapter

```
package io.pivotal.demo;

import com.gemstone.gemfire.cache.*;
import com.gemstone.gemfire.cache.util.*;

public class SampleListener extends CacheListenerAdapter {

    @Override
    public void afterCreate(EntryEvent e)
        // Automatically register interest on any entries created locally
        e.getRegion().registerInterest(e.getKey());
    }

    @Override
    public void afterUpdate(EntryEvent e)
        {System.out.println("Updated " + e.getOldValue());
    }
    // Other methods could be overridden as well
}
```

# Installing a Cache Listener

```
<client-cache>

    <pool name="ServerPool" subscription-enabled="true">
        <locator host="host3" port="41111">
    </pool>
    <region name="Customer">

        <region-attributes refid="CACHING_PROXY">
            <cache-listener>
                <class-name>
                    io.pivotal.demo.SampleListener
                </class-name>
            </cache-listener>
        </region-attributes>
    </region>
</client-cache>
```

clientCache.xml

# Installing a Cache Listener: Java Code

```
import com.gemstone.gemfire.cache.*;
import io.pivotal.demo.SampleListener;

public class createRegion ...{
    public static void main(String args[]) {
        ...
        RegionFactory rf =
            cache.createRegionFactory(RegionShortcut.PARTITION);
        rf.addCacheListener(new SampleListener());
        custRegion = rf.create("Customer");
        ...
    }
}
```

# Parameterizing a Cache Listener

```
<region name="Customer">  
    ...  
    <cache-listener>  
        <class-name>io.pivotal.demo.MyCacheListener</class-name>  
        <parameter name="url">  
            <string>https://myservices.com/myservice</string>  
        </parameter>  
    </cache-listener>  
    ...
```

Cache.xml

# Using the parameters

```
public class MyCacheListener extends CacheListenerAdapter
    implements Declarable {

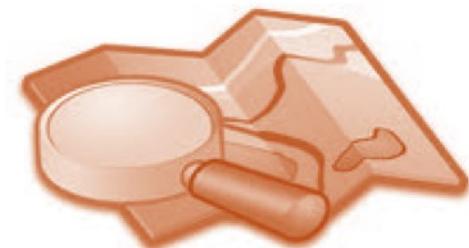
    public void afterCreate(EntryEvent e) {
        . . .
        // Initialize the Messaging service/Web service using input
        // parameters

        WebService.getConnection(url, username, password);
        . . .
    }
    . . .

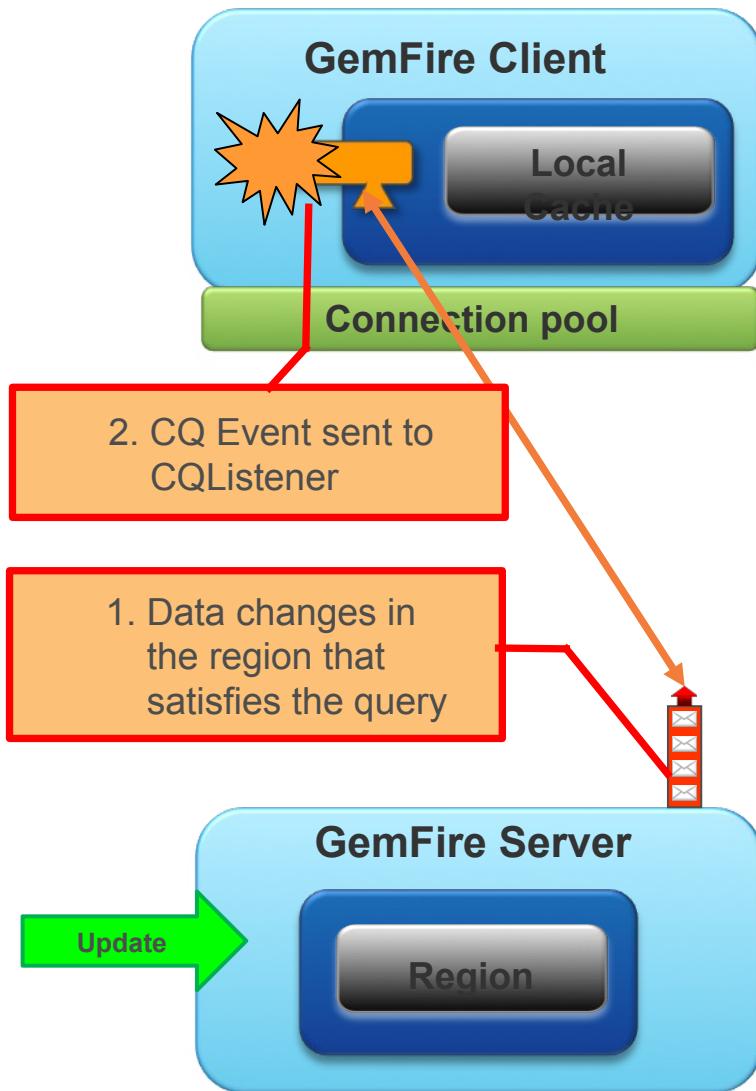
    public void init(Properties props) {
        this.url = props.getProperty("url");
    }
}
```

# Lesson Road Map

- GemFire Client Events
- Registering Interest
- Implementing an Event Listener
- **Continuous Queries**



# Continuous Queries



- Allow you to subscribe to server-side events using SQL-type query filtering
- The query is sent to the server side for execution and receives results that satisfy the criteria
- Stays on the server, but executes only when a change of data happens in the specified region
- Attached to a region – no joins in CQ
- Use judiciously – can consume lots of CPU
- Client cache is not automatically updated. Client event listener can decide what to do with the event

# CQ Event and CQ Listener

```
public class SimpleCQListener implements CqListener {  
    public void close() {}  
    public void onError(CqEvent event) {}  
    public void onEvent(CqEvent event) {}  
}
```

```
public interface CqEvent{  
    public Operation getBaseOperation();  
    public CqQuery getCq();  
    public byte[] getDeltaValue();  
    public Object getKey();  
    public Object getNewValue();  
    public Operation getQueryOperation();  
    public Throwable getThrowable();  
}
```

# Registering a Continuous Query

```
// Get a reference to the pool
Pool myPool = PoolManager.find("client");

// Get the query service for the Pool
QueryService queryService = myPool.getQueryService();

// Create CQ Attributes
CqAttributesFactory cqAf = new CqAttributesFactory();
cqAf.addCqListener(new SimpleCQListener());
CqAttributes cqa = cqAf.create();

// Construct a new CQ
String query = "IMPORT com.bookshop.domain.BookOrder; " +
               "SELECT * FROM /BookOrders b " +
               "WHERE b.totalPrice > 100.00";
CqQuery myCq = queryService.newCq("myCQ", query, cqa);

// Execute the Cq. This registers the cq on the server. Either
// executeWithInitialResults() or just execute() can be used
SelectResults sResults = myCq.executeWithInitialResults();
myCq.execute();
```

# Processing Initial Results

- Results returned as a collection of Struct representing key & value pairs

```
// Initial setup as before

// Execute the Cq. This registers the cq on the server. Either
// executeWithInitialResults() or just execute() can be used
SelectResults sResults = myCq.executeWithInitialResults();

for (Object obj : sResults) {
    Struct s = (Struct) obj;
    MyValueObj mvo = (MyValueObject) s.get("value");
    String k = (String) s.get("key");
    // Do any other processing on result
}
```

# Lab

**In this lab, you will**

- 1.**Configure a cache listener to log data in a file
- 2.**Implement a Continuous Query

# Pivotal

BUILT FOR THE SPEED OF BUSINESS

# Data Serialization

# Learner Objectives

**After this lesson, you should be able to do the following:**

- Describe Data Serialization in GemFire
- Describe the Serialization options available from GemFire
- Understand the key strengths of PDX Serialization

# Lesson Road Map

- **Java Serialization**
- Using GemFire Serialization APIs
- PDX
- DataSerializable



# Serialization in a Nutshell

*The process of converting a data structure or object into a format that can be stored and resurrected later in the same or different environment*

- Often used for
  - Network transmission of data
  - File storage
  - In memory storage
- Serialized object can be de-serialized resulting in an identical clone of original
- Also called deflating or marshaling an object

# Java Serialization - The Basics

- Minimum requirement for objects used in GemFire
- Requires marking classes as implementing `java.io.Serializable`
- All encapsulated objects must also be marked as `Serializable`
- Most Java types are `Serializable`

```
public class MyDomainObject implements java.io.Serializable { ... }
```

# Java Serialization – Technical Details

- Uses simple translation of fields into a byte stream
- Primitives as well as non-transient and non-static referenced objects are encoded
- Assuming the above:
  - Each contained object must also be serialized
  - If referenced objects aren't Serializable, the serialization fails
  - Supports circular object graphs
  - Developer influence behavior (ex marking objects as transient, etc)

# Implementing Externalizable

- Extends Serializable
- Offers developer more control over serialization
- Used when specific control is needed during serialization
- Frequent changes can make this mechanism brittle

```
public class MyDomainObject implements java.io.Externalizable {  
    public void readExternal(ObjectInput in)  
        throws ClassNotFoundException, IOException { }  
    public void writeExternal(ObjectOutput out)  
        throws ClassNotFoundException, IOException { }  
}
```

# GemFire De-serialization Scenarios

- GemFire will de-serialize region data on the server any time one of the following occurs
  - When a CacheListener/Writer calls getNew/OldValue
  - When a Function access region data
  - If a region has an index
  - If a query is done on a region
- Exceptions and caveats
  - Once a value is de-serialized, the object form is kept until it is updated again
  - **PDX makes it possible to avoid de-serialization in the case of index and queries on regions**

# Lesson Road Map

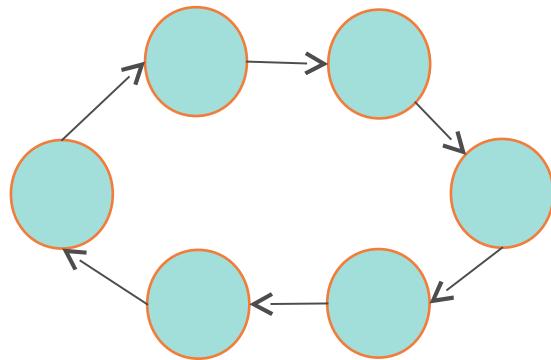
- Java Serialization
- **Using GemFire Serialization APIs**
- PDX
- DataSerializable



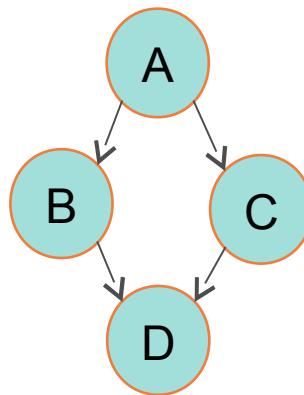
# Implications of Circular Object Graphs

- GemFire serialization does NOT support circular object graphs whereas Java serialization does
  - If the same object is referenced more than once in an object graph, the object is serialized for each reference
  - De-serialization produces multiple copies of the object
- Java Serialization serializes the object once
  - It produces one instance of the object with multiple references.

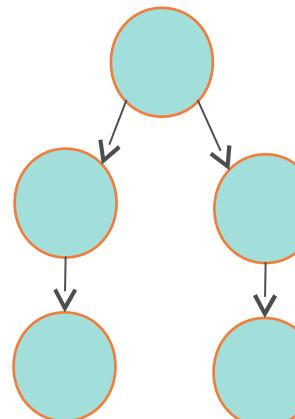
# GemFire Serialization – Referential Problems



Will cause StackOverflowError



Becomes.....  
Referential  
integrity not  
maintained



# Comparing GemFire Serialization APIs

Capability	GemFire Data Serialization	PDX Serialization
Implements java.io.Serializable	✓	
Provides compatibility across multiple versions of an Object (as long as the changes are just additions or subtractions of fields)		✓
Provides single field access of data without full deserialization (OQL as well)		✓
Automatically ported to other languages		✓
.NET	✓	✓
C++	✓	✓ (GemFire 8)
Works with Delta Propagation	✓	✓ ✈️ ✈️ (with caveats)

# Why Use GemFire Serialization

- Java Serialization is NOT portable between languages so only useful for Java clients
- Usually faster and more compact than standard Java serialization
- More importantly: PDX Serialization allows access to data without de-serialization
- Lowered memory impact using PDX Serialization

# Lesson Road Map

- Java Serialization
- Using GemFire Serialization APIs
- **PDX**
- DataSerializable



# PDX Serialization Overview

- Portable Data eXchange
- Cross language data format – can reduce the cost of distributing objects
- Data stored in named fields
- Support for all clients (as of GemFire 8.0)
- Benefits
  - **#1: Ease of use**
  - Non-Java clients don't need a special Java class
  - Reduced memory consumption on servers
  - Supports domain class versioning
  - No need for domain classes on server classpath\*  
*if read-serialized flag is set to true*

# Versioning with PDX

- Support for the evolution of objects
  - You can add or remove fields in a given version of an object
  - You may NOT change the type of a field
- Object field meta-data is stored in a central repository
  - Meta-data is passed along with object
  - If referenced field does not exist for given version, a default value (null or 0) is returned

# Portability with PDX

- Central registry is key to portability
  - Used to store and distribute the meta-data describing each type that has been serialized by PDX
  - A unique type code is generated automatically when a new type is found during serialization
  - The distributed-system-id byte is added to each generated PDX type code for Multi-site configurations

# Reduced De-serialization Requirements

- Object stored in serialized form
  - Only need to de-serialize the fields desired
  - Supports versioning – all described fields maintained even though a “version” of the object may not reference it
- Caveats
  - Serialization is 25% slower than DataSerializable

# Options for PDX Serialization

- Reflection-based (automatic) Serialization
  - Advantage: No extra coding to your domain object(s)
  - Disadvantage: Must register `ReflectionBasedAutoSerializer` with cache and register each domain object with serializer but can use wildcards
- Implementing `PdxSerializable`
  - Advantage: One place (the domain object) to define serialization
  - Disadvantage: Mixing of code concerns
- Using `PdxSerializer`
  - Advantage: No extra coding to your domain object(s)
  - Disadvantage: You must write and register your own serializer class

# 5 Steps to PDX Serialization

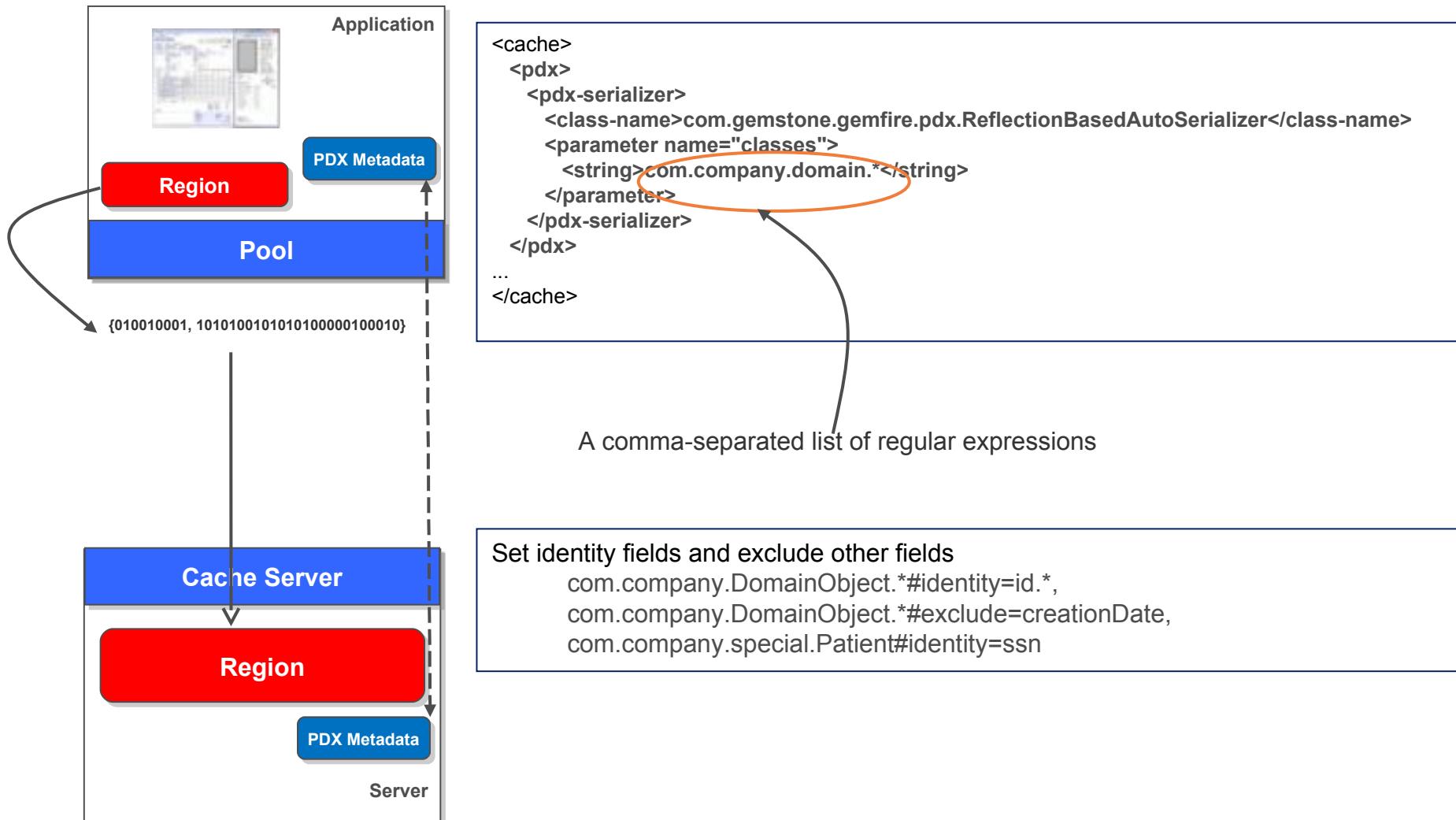
1. Pick a serialization approach
2. Configure appropriately – including read-serialized=true
3. Optionally configure persistence with PDX serialization (if persisting to disk)
4. Optionally set distributed-system-id in each member gemfire.properties (for Multi-site configurations)
5. Optionally write application code against PdxInstance – an advanced option

# Lesson Road Map

- Java Serialization
- Using GemFire Serialization APIs
- PDX
  - Auto Serialization
  - Implementing PdxSerializable
  - Implementing PdxSerializer
  - Using PdxInstance
- DataSerializable



# PDX Reflection-based Serialization



# Enabling Auto Serialization

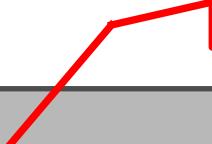
1. Configure pdx-serializer class to be ReflectionBasedAutoSerializer
  - Each JVM that will serialize/de-serialize requires config
2. Add 'classes' pattern
  - Failure to register classes means no classes serialized
  - Can use Java RegularExpressions
3. Add exclude and identity assignments as needed
  - Add #exclude=pattern or #identity=pattern to RegEx above
4. Ensure domain classes AND all contained classes have a default or no-argument constructor
  - A warning will be logged & Java Serialization attempted

# 1. Configure PDX Serializer

- Set PDX Serializer to use ReflectionBasedAutoSerializer
  - Cache.xml
  - Java API

## cache.xml

```
<cache>
    <pdx read-serialized="true">
        <pdx-serializer>
            <class-name>
                com.gemstone.gemfire.pdx.ReflectionBasedAutoSerializer
            </class-name>
            ...
        </pdx-serializer>
    </pdx>
...
</cache>
```



Optional flag that tells  
cache instance NOT to  
de-serialize entry on fetch

## 2. Customize Serializer Behavior

- Use classes pattern to specify classes to serialize
- Can be done in Cache.xml or Java

### cache.xml

```
<cache>
    <pdx>
        <pdx-serializer>
            <class-name>
                com.gemstone.gemfire.pdx.ReflectionBasedAutoSerializer
            </class-name>
            <parameter name="classes">
                <string>com.company.domain.DomainObject</string>
            </parameter>
        </pdx-serializer>
    </pdx>
...
</cache>
```

### 3. Optionally use exclude and identity

- Class pattern strings
  - Standard regular expressions
  - Select all classes from com.company.domain
    - com.company.domain.\*
  - Exclude fields from serialization
    - com.company.domain.\*#identity=id.\*#exclude=creationDate

```
<cache>
  <pdx>
    <pdx-serializer>
      <class-name>com.gemstone.gemfire.pdx.ReflectionBasedAutoSerializer
      </class-name>
      <parameter name="classes">
        <string>com.company.domain.*#identity=id.*#exclude=creationDate
        </string>
      </parameter>
    </pdx-serializer>
  </pdx>
...
</cache>
```

# A Word on Identity Fields

- Fields used by PDX to identify your object (for queries)
- Used to compare objects like distinct queries
- Used when a PdxInstance computes its hashCode() and equals() behavior
  - If not specified, all PDX fields are used
  - Identified fields should match fields used in hashCode() and equals()

## Syntax of the pattern String

```
<class pattern> [# (identity|exclude) = <field pattern>] ... [,  
<class pattern>...]
```

# 4. Ensure PDX Enabled for Target Classes

- Configure Check Portability
  - Cache.xml or Java API
  - Use to find out which classes are still using standard Java Serialization

cache.xml

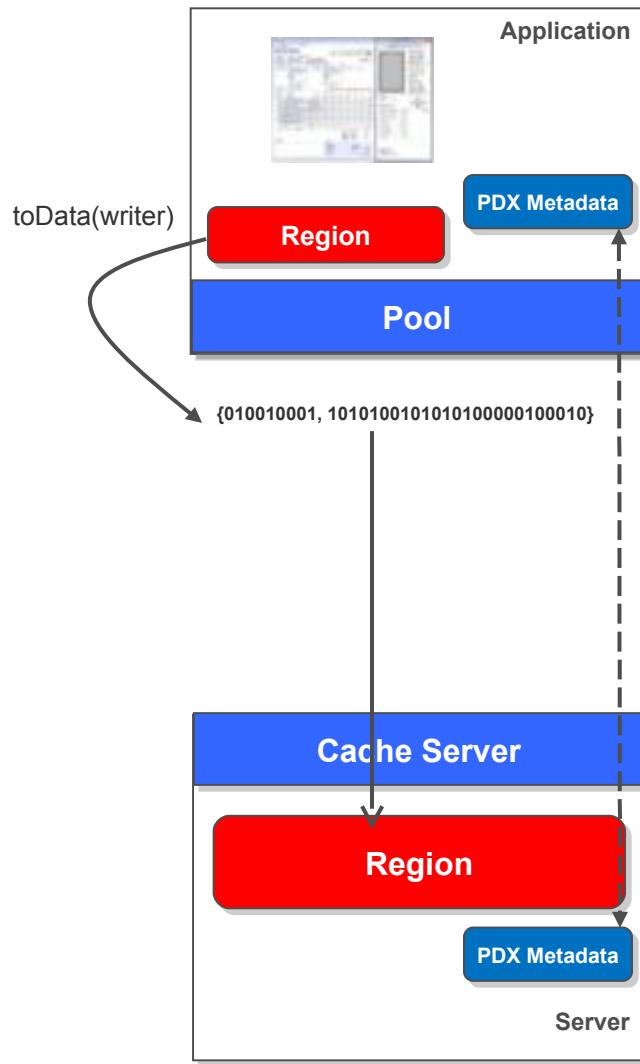
```
<cache>
  <pdx>
    <pdx-serializer>
      <class-name>
        com.gemstone.gemfire.pdx.ReflectionBasedAutoSerializer
      </class-name>
      ...
      <parameter name="check-portability"> <string>true</string>
    </parameter>
    </pdx-serializer>
  </pdx>
...
</cache>
```

# Lesson Road Map

- Java Serialization
- Using GemFire Serialization APIs
- PDX
  - Auto Serialization
  - **Implementing PdxSerializable**
  - Implementing PdxSerializer
  - Using PdxInstance
- DataSerializable



# Implementing PdxSerializable



```
<cache>
  <TypeMetaDataStore disk-store-name="default"/>
</cache>
```

```
public interface PdxSerializable {
    public void toData(PdxWriter writer);
    public void fromData(PdxReader reader);
}
```

```
//Get a reference to the pool
public class Person implements PdxSerializable {
    private int id;
    private String name;
    private List addressList;

    // Setters and getters method implementation.

    // PdxSerializable implementation.
    public void toData(PdxWriter writer) {
        writer.writeInt("id", id);
        writer.markIdentityField("id");      // Effectively modifies equals() and hashCode()
        writer.writeString("name", name);
        writer.writeCollection("addressList", addressList);
    }
    public void fromData(PdxReader reader) {
        id = reader.readInt("id");
        name = reader.readString("name");
        addressList = (List)reader.readCollection("addressList");
    }
}
```

Set property  
'gemfire.validatePdxWriters'  
for extra debug messages

# Implementing PdxSerializable

- Implementing the interface

```
import com.gemstone.gemfire.pdx.PdxReader;
import com.gemstone.gemfire.pdx.PdxSerializable;
import com.gemstone.gemfire.pdx.PdxWriter;

public class PortfolioPdx implements PdxSerializable
{ ... }
```

- Zero arg constructor – explicit or default is required

```
public PortfolioPdx()
{
}
```

# Implementing toData()

- toData()

```
public void toData(PdxWriter writer) {  
    writer.writeInt("id", id)  
    .markIdentityField("id")  
    .writeDate("creationDate", creationDate)  
    .writeString("pkid", pkid)  
    .writeObject("positions", positions)  
    .writeString("type", type)  
    .writeString("status", status)  
    .writeStringArray("names", names)  
    .writeByteArray("newVal", newVal);  
}
```

- The Fields

```
// PortfolioPdx fields  
private int id;  
private String pkid;  
private Map<String, PositionPdx> positions;  
private String type;  
private String status;  
private String[] names;  
private byte[] newVal;  
private Date creationDate; ...
```

# Implementing `fromData()`

- `fromData()`

```
public void fromData(PdxReader reader)
{
    id = reader.readInt("id");
    creationDate = reader.readDate("creationDate");
    pkid = reader.readString("pkid");
    position1 = (PositionPdx) reader.readObject("position1");
    position2 = (PositionPdx) reader.readObject("position2");
    positions = (Map<String, PositionPdx>)
        reader.readObject("positions");
    type = reader.readString("type");
    status = reader.readString("status");
    names = reader.readStringArray("names");
    newVal = reader.readByteArray("newVal");
    arrayNull = reader.readByteArray("arrayNull");
    arrayZeroSize = reader.readByteArray("arrayZeroSize");
}
```

# Lesson Road Map

- Java Serialization
- Using GemFire Serialization APIs
- **PDX**
  - Auto Serialization
  - Implementing PdxSerializable
  - **Implementing PdxSerializer**
  - Using PdxInstance
- DataSerializable



# Implementing PdxSerializer

- Similar to PdxSerializable
- Implement toData() and fromData()
- The fromData() method responsible for creating & returning instance of class
  - Allows using constructor of your choice (i.e. no absolute requirement for default constructor)
- Ability to add PDX support without modifying the class
- Use one PdxSerializer implementation for entire cache

# Declaring PdxSerializer

- Implementing the interface

```
import com.gemstone.gemfire.cache.Declarable;
import com.gemstone.gemfire.pdx.PdxReader;
import com.gemstone.gemfire.pdx.PdxSerializer;
import com.gemstone.gemfire.pdx.PdxWriter;

public class ExamplePdxSerializer implements PdxSerializer,
Declarable
{ ... }
```

- Cache.xml

```
<cache>
  <pdx> <pdx-serializer>
    <class-name>com.company.ExamplePdxSerializer</class-name>
  </pdx-serializer> </pdx>
...
</cache>
```

# Implementing `toData()`

- Write each standard Java field using `PdxWriter` to write metadata
- Use `markIdentifyingField()` to flag identity fields to GemFire
- In the same version of the class, write order and naming must remain the same
- For best performance, write fixed width fields first , then variable width
- `writeObject()`, `writeObjectArray()` and `writeField()` support a `checkPortability` argument to check portability at runtime

# Implementing `fromData()`

- Create an instance
- Read your data fields using `PdxReader`
- Return the created object
- Be sure to use the same names as `toData()`

# Lesson Road Map

- Java Serialization
- Using GemFire Serialization APIs
- **PDX**
  - Auto Serialization
  - Implementing PdxSerializable
  - Implementing PdxSerializer
  - **Using PdxInstance**
- DataSerializable



# PdxInstance

- Allows access and modification to PDX data without de-serializing
  - Ask for values of a field by name
  - Ask if field exists
  - Ask for all field names
- The `toString()` method prints out all field names and values
- Never stored in the cache – it's just a container for serialized data
- Overrides objects `equals()` and `hashCode()` methods

# Using PdxInstance

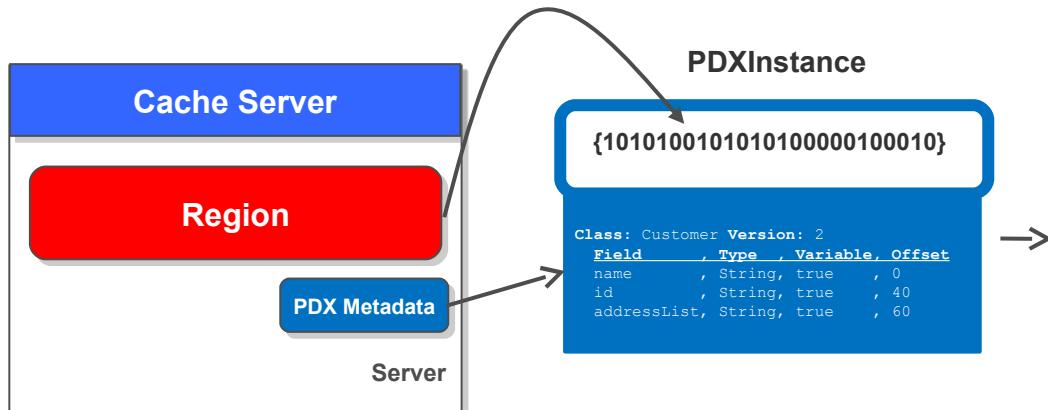
PDXInstance is immutable !

Use myPDXInstance.createWriter() to update

put(key, updatedPDX) to region to update cache

```
//Configure the Cache to return PDX wrappers instead of deserialized objects  
CacheFactory.setPdxReadSerialized(true);  
  
// Get the customer with a specific id and get the value of the name field  
String customerId = "010934303939";  
WritablePdxInstance wpdxCust = customerRegion.get(customerId).createWriter();  
wpdxCust.setField("name", "Roger Rabbit");  
// Update the value in the cache  
customerRegion.put(customerId,wpdxCust);
```

```
// Getting the PDX object without de-serializing  
PdxInstance pdxCust = (PdxInstance)event.getNewValue();  
// Getting field values, without constructing the domain object  
String name = (String)pdxCust.getField("name");
```



```
public class PDXInstance {  
    public boolean hasField(String fieldName);  
    public Object getField(String fieldName);  
    public List<String> getFieldNames();  
    public boolean isIdentityField(String fieldName);
```

```
Object getObject();  
WritablePdxInstance createWriter();
```

```
public boolean equals(Object obj);  
public int hashCode();  
public String toString();
```

```
}
```

```
public interface WritablePDXInstance {  
    public void setField(String fieldName, Object value);  
}
```

# Steps to Using PdxInstance

1. Set read-serialized to true where data will be read

```
<cache>
    <pdx read-serialized="true" />
    ...
</cache>
```

2. Write code to fetch entries and use PdxInstance

```
// get checks Object type and handles each appropriately
Object myObject = myRegion.get(myKey);
if (myObject instanceof PdxInstance)
{
    // get returned PdxInstance instead of domain object
    PdxInstance myPdxInstance = (PdxInstance)myObject;

    // PdxInstance.getField deserializes the field, but not the object
    String fieldValue = myPdxInstance.getField("stringFieldName");
    ...
}
```

# Updating Values in PdxInstance

```
// put/get code with serialized read behavior
// put is done as normal
myRegion.put(myKey, myPdxSerializableObject);

// get checks Object type and handles each appropriately
Object myObject = myRegion.get(myKey);
if (myObject instanceof PdxInstance)
{
    // get returned PdxInstance instead of domain object
    PdxInstance myPdxInstance = (PdxInstance)myObject;

    // PdxInstance.getField deserializes the field, but not the object
    String fieldValue = myPdxInstance.getField("stringFieldName");

    // Update a field and put it back into the cache
    // without deserializing the entire object
    WritablePdxInstance myWritablePdxI = myPdxInstance.createWriter();
    myWritablePdxI.setField("fieldName", fieldValue);
    region.put(key, myWritablePdxI);

    // Deserialize the entire object if needed, from the PdxInstance
    DomainClass myPdxObject = (DomainClass)myPdxInstance;
} else if (myObject instanceof DomainClass) {
    // get returned instance of domain object
} ...
```

# Using PdxInstanceFactory

- Can be used when you don't have access to domain class
- Similar to PdxWriter

```
PdxInstance pi =  
    cache.createPdxInstanceFactory("com.company.DomainObject")  
    .writeInt("id", 37)  
    .markIdentityField("id")  
    .writeString("name", "Mike Smith"))  
    .create();
```

# Persistent PDX

- Required for caches that use PDX with persistent regions
- Required for caches that use PDX and WAN
  - You must set the distributed-system-id GemFire property

```
<pdx read-serialized="true"  
      persistent="true" disk-store-name="SerializationDiskStore">  
    <pdx-serializer>  
      <class-name>pdxSerialization.defaultSerializer</class-name>  
    </pdx-serializer>  
</pdx>  
<region ...>
```

# PDX for Keys

- Highly discouraged
- Changes hashCode behavior
- If you MUST
  - Do not set read-serialized to true
  - Use a different disk store than regions not using PDX keys

# Lesson Road Map

- Java Serialization
- Using GemFire Serialization APIs
- PDX
- **DataSerializable**



# Using DataSerializable

- The fastest and most compact representation – but:
  - Requires lots of serialization and de-serialization on access
  - Doesn't work well with complex object graphs
- Includes utility methods to assist in Serialization
- Avoids overhead of reflection by using an Instantiator object
  - Alternatively, a no-arg constructor can be used (preferred)
- Cannot use with 3<sup>rd</sup> party classes

# Implementing DataSerializable

- There are two methods very similar to PDX
  - `toData()`
  - `fromData()`

```
public void fromData(DataInput in) {}

public void toData(DataOutput out) {}
```

- Alternatively, use DataSerializer
  - Serialize domain objects without modifying the domain class itself

# Using DataSerializer

```
public boolean toData(Object o, DataOutput out) throws IOException
{
    if (o instanceof Company)
    {
        Company company = (Company) o;
        out.writeUTF(company.getName());
        Address address = company.getAddress();
        DataSerializer.writeObject(address, out);
        return true;
    } else { return false; }
}

public Object fromData(DataInput in) throws IOException,
ClassNotFoundException
{
    String name = in.readUTF();
    Address address = (Address) DataSerializer.readObject(in);
    return new Company(name, address);
}
} //end of class
```

# Mixing Serialization Strategies

- Mixing DataSerializable with Serializable or PdxSerializable a bad idea
  - Increased memory use
  - Lower throughput
- Especially true if using Collections
  - The bigger the collection, the lower the throughput
  - Metadata for collection entries not shared

# Lab

**In this lab, you will**

1. Configure PdxSerialization using Auto Serializer
2. Load data and verify correct serialization
3. Modify domain object and verify multiple versions still work

# Pivotal

BUILT FOR THE SPEED OF BUSINESS

# Transaction Management

# Learner Objectives

**After this lesson, you should be able to do the following:**

- Describe GemFire transaction features
- Describe transactions in a:
  - Partitioned region
  - Replicated region
- Describe GemFire client transactions and APIs
- Understand how events work in a transaction

# Lesson Road Map

- **GemFire Transaction Features**
  - Transaction – Partitioned Region
  - Transaction – Replicated Region
  - Client Transactions and APIs
  - Transaction Event Handling
- 



# GemFire Transaction Features

- Basic transaction properties: atomicity, consistency, isolation, and durability.
- Rollback and commit operations along with standard GemFire cache operations
- High concurrency and high performance
- Transaction statistics gathering and archiving
- Compatible with Java Transaction API (JTA) transactions, using either the GemFire JTA or a third-party implementation.

# GemFire Transaction Support

GemFire provides support for two kinds of transactions:

- Cache transactions

- Native GemFire specific transaction manager
- Used when transactions exist exclusively within GemFire distributed system
- The begin, commit and rollback actions are directly controlled by the application

- JTA global transactions

- Java Transaction API – part of the Enterprise Java spec
- Useful when you want to coordinate transactions between GemFire and traditional JDBC type resources

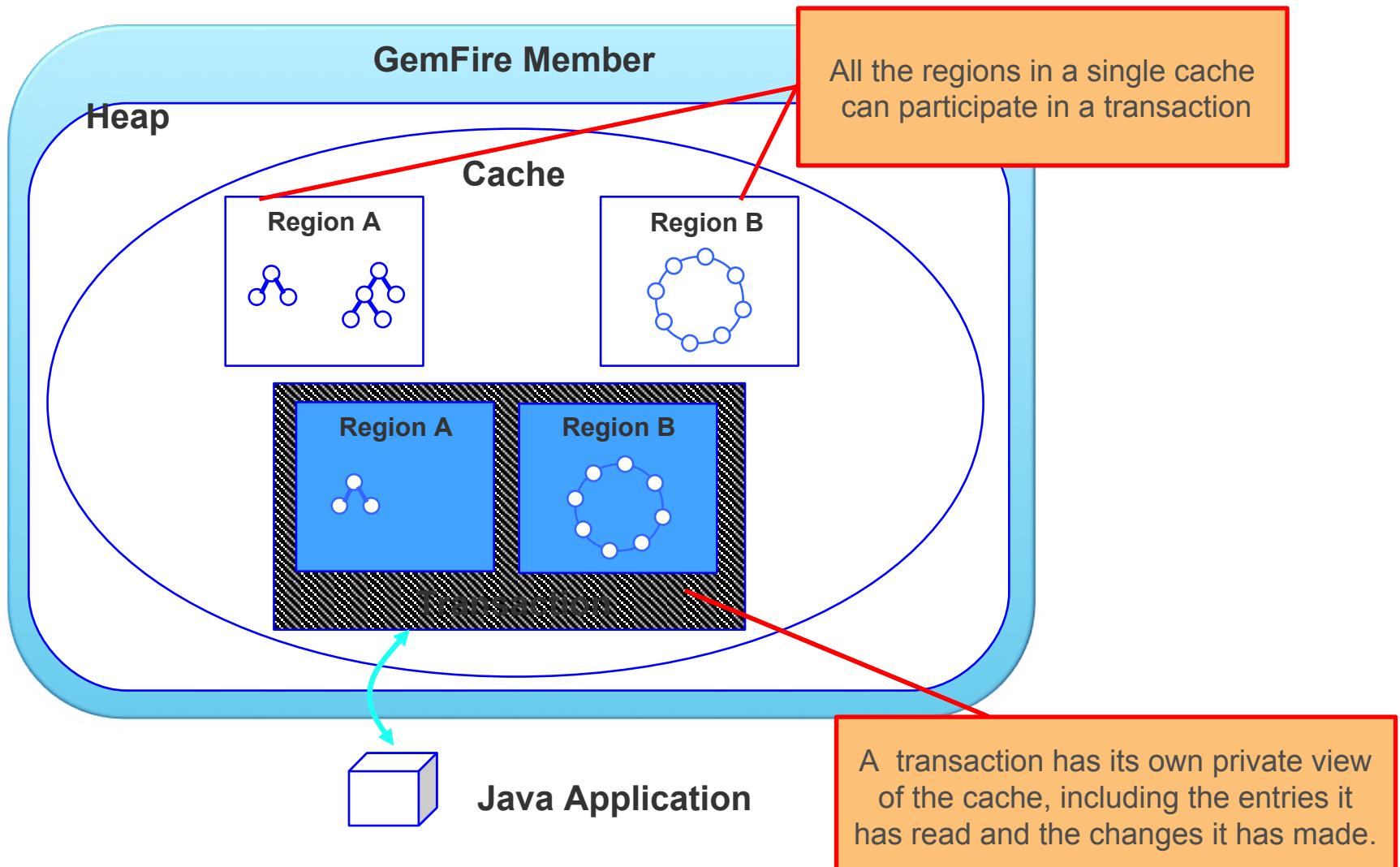
# GemFire Cache Transactions

- GemFire cache transactions enables:
  - Grouping multiple cache operations, and handle them as a single unit.
  - Either independent transaction execution under application control, or a part of global JTA transactions
- The begin, commit, and rollback actions are directly controlled by the application, unless the application executes within a J2EE container
- Applications manage transactions on a per-cache basis

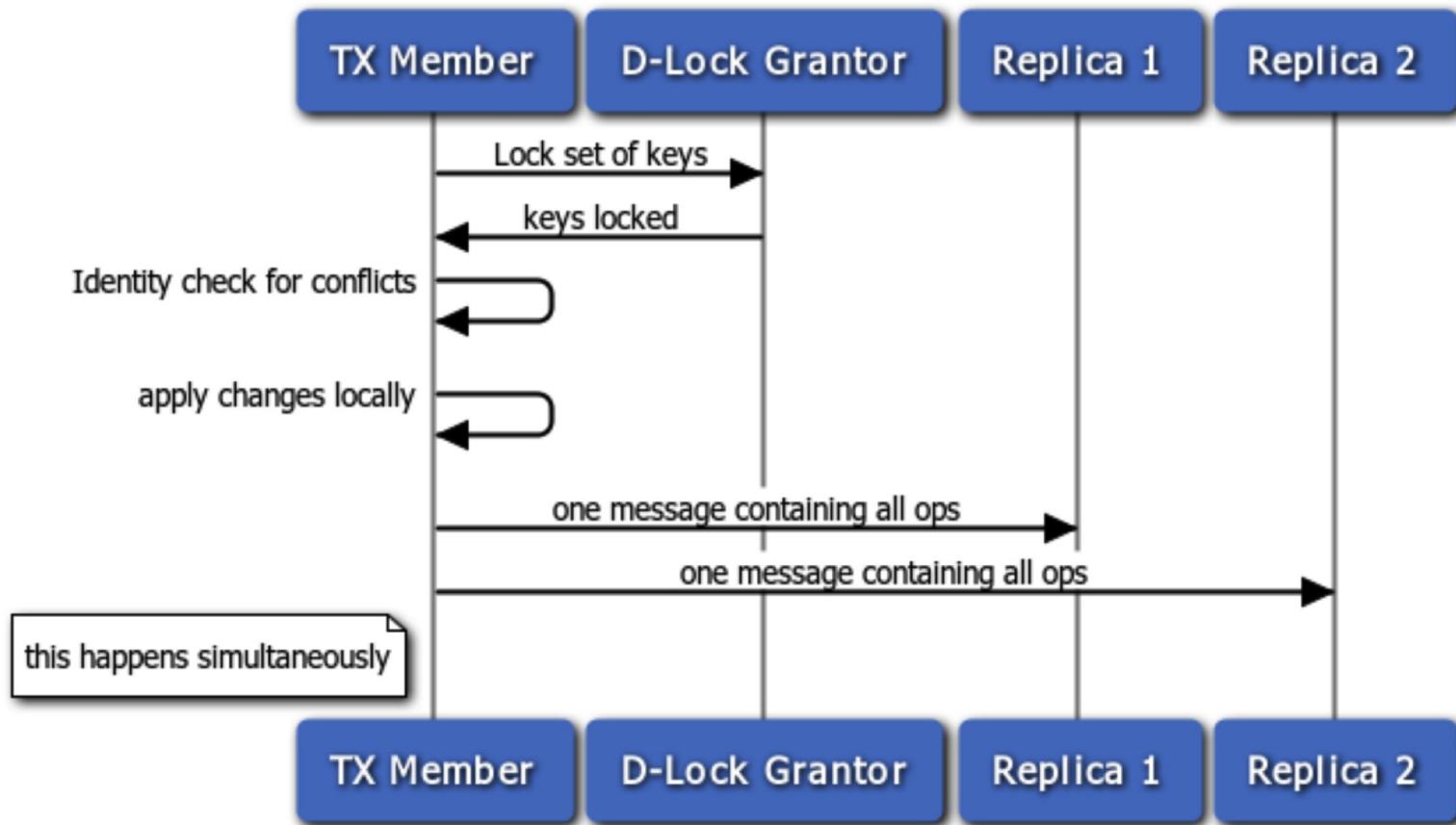
GemFire transactions cannot span JVMs.

This is the primary reason for using Co-location combined with functions  
More on this later

# How Cache Transactions Work



# What Happens on Commit



# Transactions and copy-on-read

- By default, object returned by `get()` is a direct reference to the entry
  - Can occur when working in the server (ex functions)
  - Results in poor isolation in transactions
- Using copy-on-read forces a copy of the entry to be returned

```
<cache copy-on-read="true">
    <!-- region definitions, etc -->
</cache>
```

- This is especially important when making entry changes within a transaction

# Lesson Road Map

- GemFire Transaction Features
  - **Transaction – Partitioned Region**
  - Transaction – Replicated Region
  - Client Transactions and APIs
  - Transaction Event Handling
- 

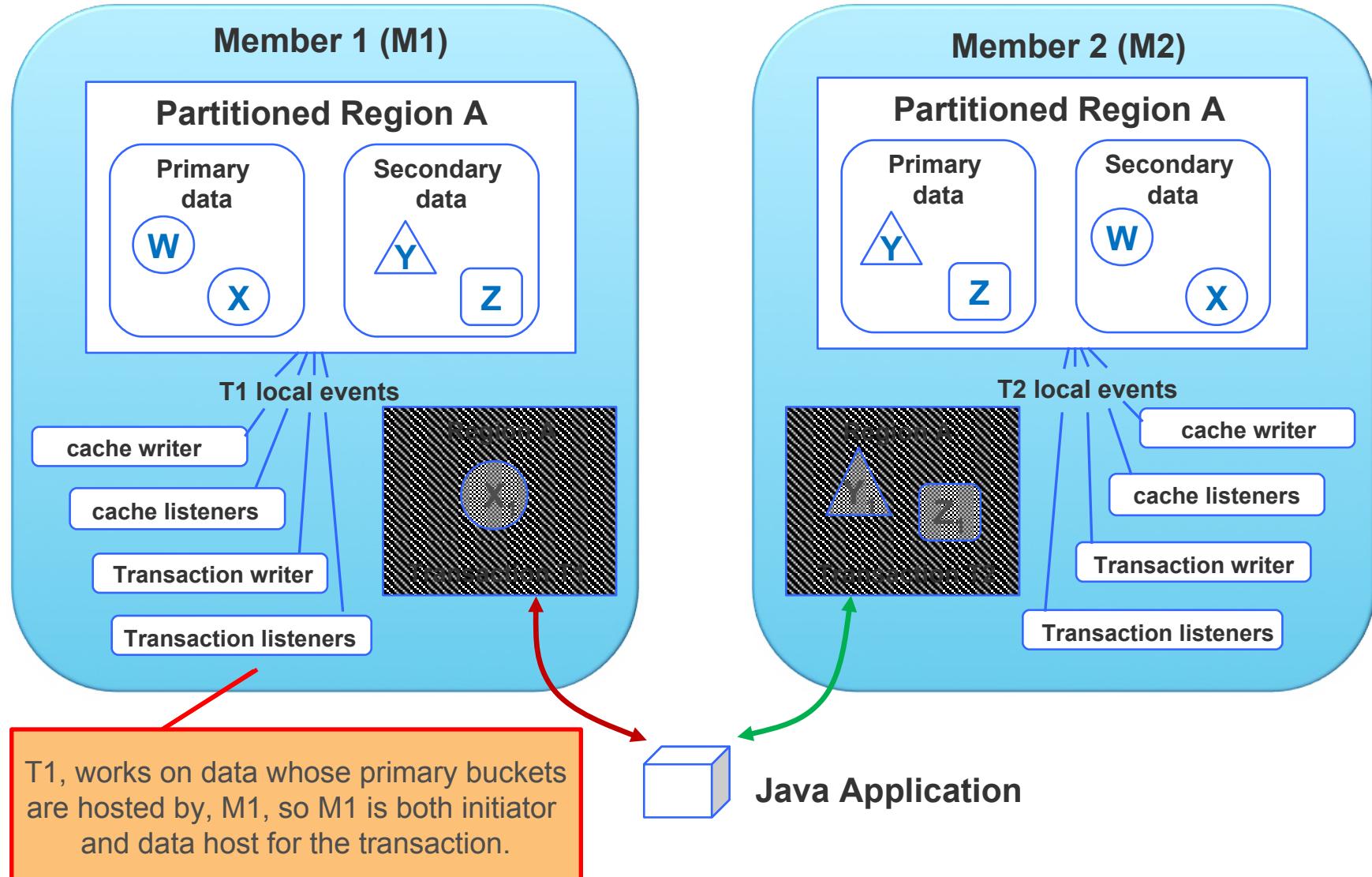


# Transactions in Partitioned Regions

## In partitioned regions:

- Transaction operations are done first on the primary data host then distributed to other members
- The member running the transaction code is called the transaction initiator
- The member that hosts the data, and the transaction is called the transactional data host
- Local locks are used – changes are sent to redundant data at commit time
- No global locks
- Currently, a transaction must happen on a single member. This will change in the future. – colocation
- Any member with the Region defined can initiate the transaction

# Transactions in Partitioned Regions



# Lesson Road Map

- GemFire Transaction Features
  - Transaction – Partitioned Region
  - **Transaction – Replicated Region**
  - Client Transactions and APIs
  - Transaction Event Handling
- 



# Transactions in Replicated Regions

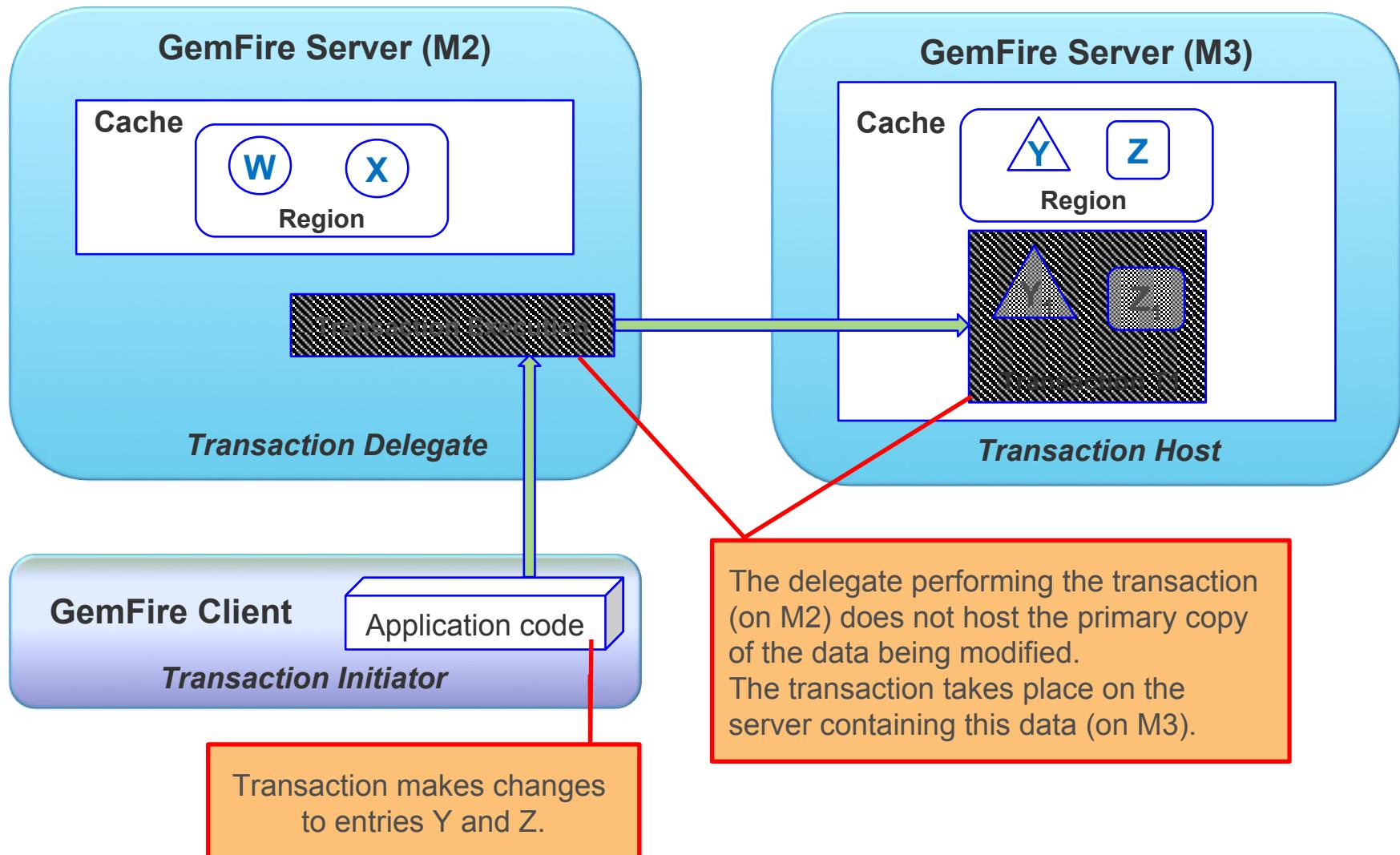
- The transaction and its operations are applied to the local member, and the resulting transaction state is distributed to other members
- The region's scope attribute specifies how data is distributed during the commit phase:
  - distributed-no-ack
  - distributed-ack
  - local

# Lesson Road Map

- GemFire Transaction Features
  - Transaction – Partitioned Region
  - Transaction – Replicated Region
  - **Client Transactions and APIs**
  - Transaction Event Handling
- 



# Client Transactions



# Transaction Manager API

- GemFire cache transaction manager:

```
CacheTransactionManager txManager =  
    custRegion.getCache().getCacheTransactionManager();
```

- JTA transaction manager:

```
Context ctx = cache.getJNDIContext();  
UserTransaction txManager =  
    (UserTransaction)ctx.lookup("java:/UserTransaction");
```

- Program to run transaction:

```
try {  
    txManager.begin();  
    // ... do work  
    txManager.commit();  
}  
catch (CommitConflictException conflict) {  
    txManager.rollback();  
}
```

# Transaction Coding Example

```
public static class DiscountManager {  
  
    public void discount(CustomerId cid, OrderId oid,  
                         BigDecimal discount) {  
        // Assumes customerRegion and orderRegion set somewhere  
        CacheTransactionManager mgr =  
            ((Cache) customerRegion.getRegionService())  
                .getCacheTransactionManager();  
        Customer custToUpdate = customerRegion.get(cid);  
        Order orderToUpdate = orderRegion.get(oid);  
  
        mgr.begin();  
  
        custToUpdate.setDiscount(discount);  
        orderToUpdate.applyDiscount(discount);  
        customerRegion.put(cid, custToUpdate);  
        orderRegion.put(oid, orderToUpdate);  
  
        mgr.commit();  
    }  
}
```

Work done within the all-or-nothing transactional scope

# Transaction Caveats

- Remember, transactions can only operate within a single cache (JVM) instance
- If using transactions on persistent regions you need to explicitly set a system flag to allow (off by default)
  - Set GemFire property  
`ALLOW_PERSISTENT_TRANSACTIONS=true`

```
gfsh> gfsh start server --name=server1 --cache-xml-file=cache.xml  
      --J=-Dgemfire.ALLOW_PERSISTENT_TRANSACTIONS=true  
...
```

# Transaction Exceptions

- Exceptions can occur for a number of reasons on the client side
  - TransactionDataNodeHasDepartedException – A host participating in a transaction has departed unexpectedly
  - TransactionDataRebalanceException – A rebalance happened during transaction causing data to have moved systems
  - TransactionDataNotColocatedException – Data in a transaction is not co-located on the same host
- All should be caught
  - GemfireException and all subclasses are RuntimeException (unchecked) so immediate try/catch not required
  - Some can and should be retried (ex TransactionDataRebalanceException)
  - Some cannot be retried (ex TransactionDataNotColocatedException)
  - How & whether to retry often application specific though

# Transaction Exceptions

Exception	Description
TransactionDataNodeHasDepartedException	Tx host has shut down or no longer has the data. Only when tx is hosted on a member that is not the initiator
TransactionDataNodeColocatedException	Tx modified data that was not located on the same host. Only when tx is hosted on a member that is not the initiator
TransactionDataRebalancedException	Only on partitioned regions. Can also occur when data is not co-located
TransactionInDoubtException	Thrown in the presence of node failures, when certainty of the transaction is unknown
CommitConflictException	Thrown when a commit fails due to a write conflict
CommitDistributionException	Attempt to notify required participants of a tx involving one or more regions may have failed. Commit completed, but all members may not have been notified.
CommitIncompleteException	Commit fails due to errors

# Exception Handling Example

```
public class TransactionalDemo {  
    ...  
    public void runTransaction() {  
        ...  
        CacheTransactionManager mgr =  
            clientCache.getCacheTransactionManager();  
        // Set up code  
        try {  
            mgr.begin()  
            // ... do business logic  
            mgr.commit();  
        } catch (TransactionException tex) {  
            if (tex instanceof TransactionDataRebalancedException) {  
                // wait & retry  
            } else {  
                // Handle other exceptions  
                mgr.rollback()  
            }  
        }  
    }  
    ...  
}
```

# Lesson Road Map

- GemFire Transaction Features
  - Transaction – Partitioned Region
  - Transaction – Replicated Region
  - Client Transactions and APIs
  - **Transaction Event Handling**
- 



# Transactions and Events

- All event handlers (plugins) work with transactions on the server
- Transaction events don't directly propagate to clients
- Clients can register interest and receive events on changes once transaction commits

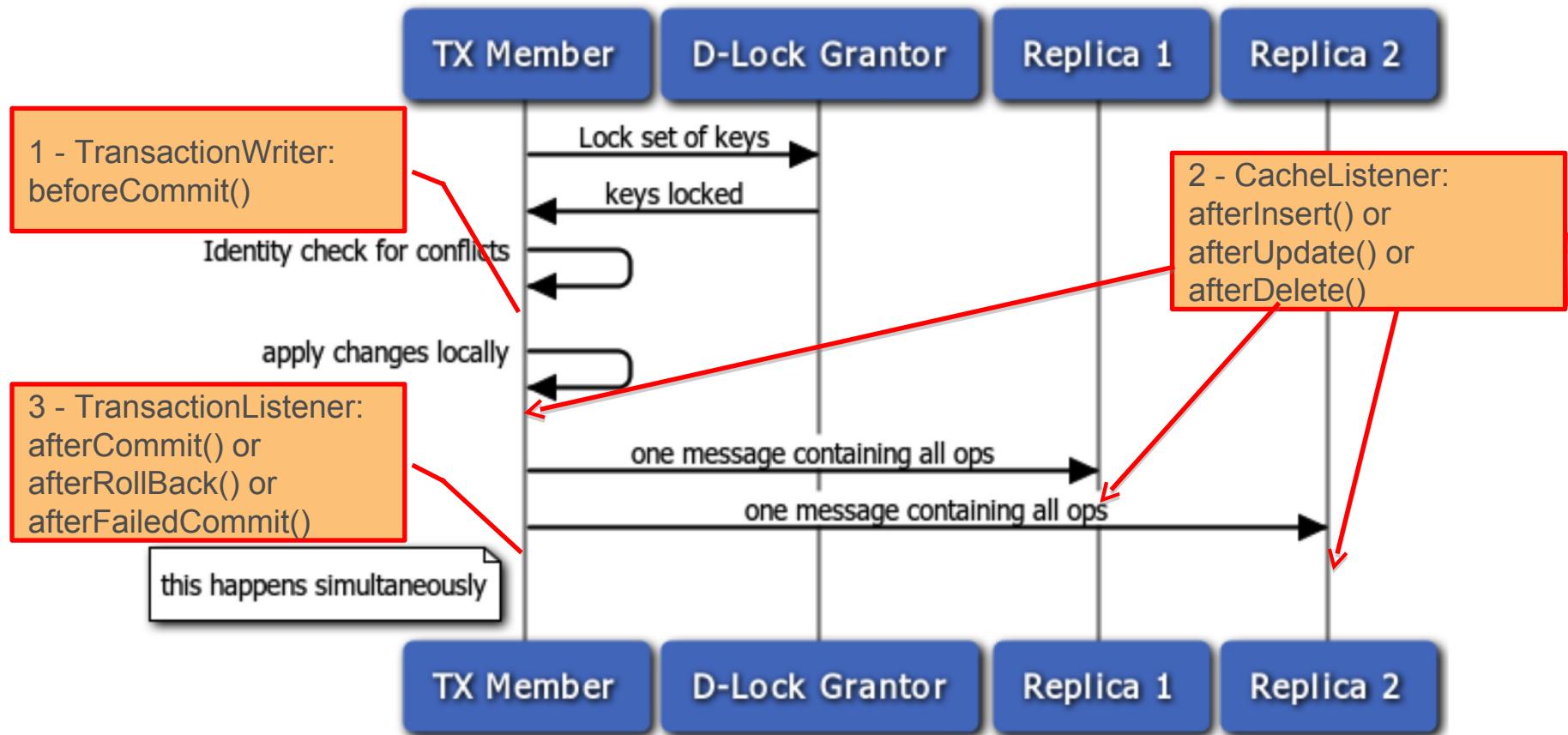
# Impact of Transactions on Cache Plugins

- CacheLoader
  - Values loaded by the cache loader may cause a write conflict at write time
- CacheWriter
  - Only executed in the member where transactional data resides
  - Can abort the operation
  - Works in the transactional view
- CacheListener
  - Only called after transaction commits
  - Only receives a single conflated event

# Transaction Event Handlers

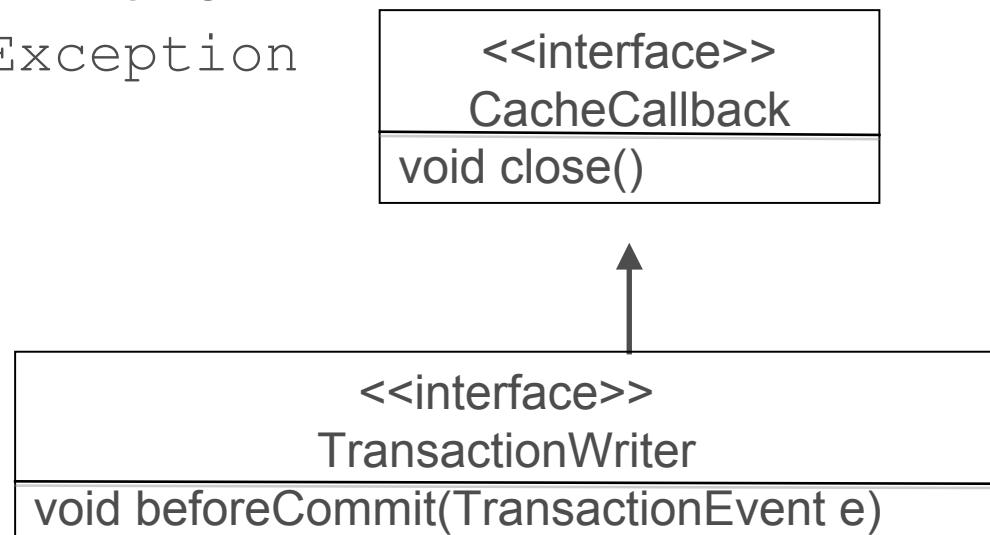
- Additional handlers available to work within transactional context
- TransactionWriter
  - Called after commit is called but before action taken
  - Can abort transaction by throwing exception
  - Only install one per cache server
- TransactionListener
  - Allows for follow up on successful commit, failed commit and rollbacks
  - Could start a new transaction
  - Can install more than one per cache server

# Commit with Event Handlers



# Event Handler API: TransactionWriter

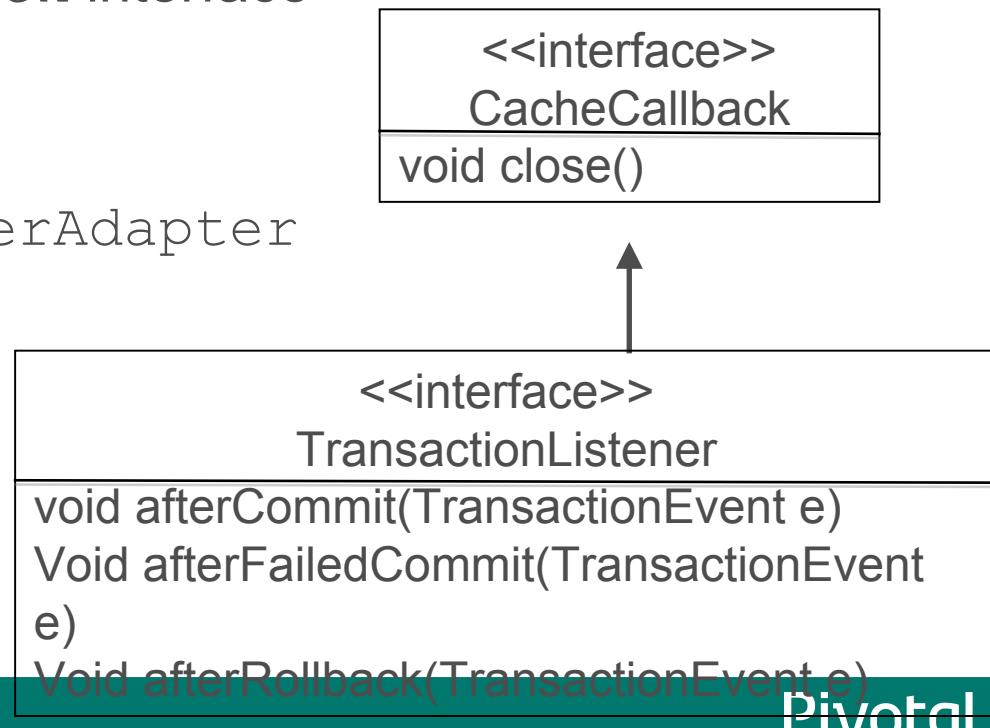
- The TransactionWriter event handler:
  - Called BEFORE commit occurs (but after conflict checking)
  - **Synchronous**
  - Can be used to abort commit via TransactionWriterException



# Event Handler API:

## TransactionListener

- The TransactionListener event handler:
  - Captures the key events of a transaction (ex commit, rollback, etc)
  - Extends CacheCallback interface
  - **Synchronous**
  - Implemented in the TransactionListenerAdapter class



# TransactionEvent API

Method	Description
getCache()	Returns the associated Cache object for the transaction event
getTransactionId()	Returns the TransactionId object for this transaction event
getEvents()	Returns an ordered list of every CacheEvent associated to this transaction event. Elements may be an instance of EntryEvent

Other methods still remain but are deprecated as of GemFire 5.0

# Registering Transaction Event Handlers

```
<cache
    xmlns="http://schema.pivotal.io/gemfire/cache"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://schema.pivotal.io/gemfire/cache
                        http://schema.pivotal.io/gemfire/cache-cache-8.1.xsd"
    version="8.1"
    copy-on-read="true">

    <cache-transaction-manager>
        <transaction-listener>
            <class-name>myPackage.MyTransactionListener</class-name>
        </transaction-listener>
        <transaction-writer>
            <class-name>myPackage.MyTransactionWriter</class-name>
        </transaction-writer>
    </cache-transaction-manager>
    ...
</cache>
```



Used solely to register transaction writer and listeners

# Lab

## In this lab, you will

1. Co-locate Order and Customer Regions
2. Implement transactions using  
CacheTransactionManager
3. Implement and register a TransactionListener

# Review of Learner Objectives

You should be able to do the following:

- Describe GemFire transaction features
- Describe transactions in a:
  - Partitioned region
  - Replicated region
- Describe GemFire client transactions and APIs
- Understand how events work with transactions

# Pivotal

BUILT FOR THE SPEED OF BUSINESS

# Writing and Registering Functions

# Learner Objectives

**After this lesson, you should be able to do the following:**

- Describe difference between data dependent and server functions
- Write a function – both data dependent and data independent
- Register server side function

# Lesson Road Map

- **Function Service Overview**
  - Implementing Functions
  - Registering Functions
  - Executing Functions
- 



# Overview of Function Execution Service

- The GemFire functions are:
  - Similar to database stored procedures, written by programmers
  - Data dependent or independent
  - Concurrently executed, map like process
  - Highly Available
- Results are collected from all participating cache members
  - Reduce like process
  - Results returned to a single member to aggregate results
- Caller retrieves the final result from a ResultCollector
  - Can be blocking/non-blocking

# Function Execution Service Use Cases

You can apply function execution service for applications to:

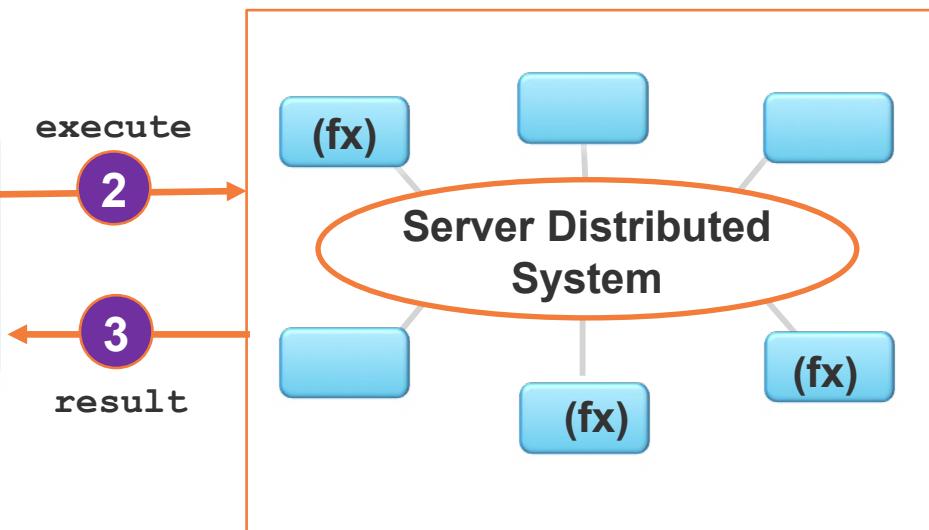
- Execute a server-side transaction, or data updates using the GemFire distributed locking service
- Initialize application components on multiple servers
- Initialize third-party services
- Iterate over values on local data sets
- Perform complex analytics
- Aggregation operations

# Where Functions are Executed

- Generally determined by
  - `on*` methods – for clients, sent to the server system
  - Region configuration
  - Filters
- Member Functions
  - A single member system
  - A subset of members (in parallel) in a distributed system
  - All the members of a distributed system (in parallel)
  - On another server system the member is a client of
- Data Aware Functions
  - Specific region and set of keys on which to run the function upon
  - Uses `onRegion` and `withFilter(keys)` to route function request
  - executed against a Partitioned Region on servers where the data is hosted for that region

# Basic Function Execution

The `FunctionService.on*` method determines the location where the function (`fx`) will be executed.



## Client Code:

```
...
ResultCollector rc = execution.execute("fx");
List result = (List) rc.getResult();
...
```

# Steps to Implement and Use Functions

1. Write the function code
2. Register the function
3. Write application code to run the function and handle any results
4. For special results handling, code a custom `ResultCollector` for your function

# Lesson Road Map

- Function Service overview and use cases
  - **Implementing Functions**
  - Registering Functions
  - Executing Functions
- 



# Interface Function

## Methods to implement:

- execute()
  - The method which contain the logic to be executed.
- getId()
  - Return a unique function identifier, called by the registration process.
  - User to locate the function via the FunctionService
- hasResult()
  - Indicates whether the function returns results
  - false for “fire and forget”
  - Whether there is one or many results is up to the code in the execute method.
- optimizeForWrite() – only valid via onRegion() function calls
  - true if your method updates data
  - false if it only reads, used by default

# Writing the Function Code

```
package org.myproject;
import com.gemstone.gemfire.cache.*;

public class MyFunction implements Function {
    public String getId() {
        return "fx";
    }

    public boolean hasResult() {
        return true;
    }

    public boolean optimizeForWrite() {
        return false;
    }

    public boolean isHA() {
        return false;
    }

    public void execute(FunctionContext fc) {
        // function logic to be written here.
    }
}
```

Set to true if you only want the function to execute against primary buckets

Code the execute () method to perform the work of the function.

# Extending FunctionAdapter

- Alternative to implementing Function
- Implements almost all methods of Function interface
- Requires implementation of execute() and getId() methods
  - Best practice: Use @Override annotation
- Defaults of remaining methods
  - hasResult() returns **true**
  - optimizeForWrite() returns **false**
  - isHA() returns **true**

# Coding the execute method

- Needs to be thread safe
  - May be invoked simultaneously
- If HA is required, consider this in the function code
  - Consider using:  
`FunctionContext.isPossibleDuplicate()`
- Use the `FunctionContext` API to get information about:
  - The function ID and the region
  - The `RegionFunctionContext` API that holds the `Region` object, and the Set of key filters
  - The `PartitionRegionHelper` API that provides access to additional information for a partition region



To query a specific node in a partitioned region, use  
`Query.execute(RegionFunctionContext context)` API

# FunctionContext API

Method	Description
getArguments()	Retrieve optional arguments sent by function caller
getFunctionId()	Returns the id of the function
getResultSender()	ResultSender is used to send results. Function writer will use ResultSender.sendResult() and ResultSender.lastResult()
isPossibleDuplicate()	Set by GemFire if re-executing an HA function. Function can use to prevent duplicate operations.

# Returning Results from Function

- Ask the FunctionContext for the result caller via getResultSender()
- Send each result via sendResult(Object)
- Send lastResult(Object) to signify that the execution is complete

```
public void execute(FunctionContext fc) {  
  
    List<Customer> allCustomerList // ArrayList of customers and their details  
  
    int result // Number of results to be sent  
  
    // Perform task to populate allCustomerList  
  
    for (int j = 0; j < (result - 1); j++) {  
        fc.getResultSender().sendResult(allCustomerList.get(j));  
    }  
    fc.getResultSender().lastResult(allCustomerList.get(result-1));  
}
```

# Sending Results

- Many choices are available
  - One item at a time
    - Send all but last item using ResultSender.sendResult()
    - Send last item using ResultSender.lastResult()
  - All results with empty last result
    - Send entire result using ResultSender.sendResult()
    - Invoke ResultSender.lastResult(null)
  - All results using lastResult()
- Note: Default ResultCollector assumes one item at a time and collects items in ArrayList<Object>
- Changing default requires agreement between function and callers when registering ResultCollector

# Data Aware Functions and RegionFunctionContext

- Determined by caller by using  
`FunctionService.onRegion(Region r)`
- Target region must be a Partitioned Region
- The object passed to the `execute()` method is actually a `RegionFunctionContext` object
- Subclass of `FunctionContext`

Method	Description
<code>getDataSet()</code>	Returns the region on which the function is being executed
<code>getFilter()</code>	Get the keys sent as a filter on the function call. These would have been sent using <code>withFilter()</code> from the caller.

# Implementing Data Aware Functions

- Cast FunctionContext to access RegionFunctionContext methods
- You can use instanceof test to assert the FunctionContext is actually RegionFunctionContext

```
public void execute(FunctionContext fc) {  
    if (fc instanceof RegionFunctionContext) {  
        RegionFunctionContext rfc = (RegionFunctionContext) fc;  
        // Do stuff with rfc  
    }  
    else {  
        throw new FunctionException("Must be called as onRegion function");  
    }  
}
```

# PartitionRegionHelper overview

Method	Description
getLocalData()	Return a region containing only the local data for a given PartitionedRegion. Designed for efficient local reads.
getLocalDataForContext()	Return a region having read access based on the RegionFunctionContext. If function has optimizeForWrite true, only primary copy of the data returned.
getColocatedRegions()	Returns a map of regions (name and region reference) for all co-located regions to given region
assignBucketsToPartitions()	Forces all buckets to be assigned even if they won't have data
isPartitionedRegion()	Is the given region a partitioned region?
getPartitionRegionInfo()	Returns a set of details related to all partitioned regions in the local cache or region (either argument can be supplied)

Note: Not an exhaustive list of capabilities. See JavaDoc for more details

# Lesson Road Map

- Function Service Overview
  - Function API's
  - **Registering Functions**
  - Executing Functions
- 



# Function Registration

There are three basic ways to register a function

1. Register in XML
2. Deploy JAR file containing Function using gfsh
3. Register programmatically

# Registering Function using XML

- Function is registered in the Cache.xml file on the cache (not any particular region)
- Function implementation MUST implement Declarable
- Function code must be on the classpath of server
- Optional arguments can then be provided
  - These are global configuration parameters NOT arguments to the function

```
<cache>
    <region name="Customer" refid="PARTITION"/>
    <function-service>
        <function>
            <class-name>org.myproject.MyFunction</class-name>
        </function>
    </function-service>
</cache>
```

# Registering Function using gfsh Deploy

- Use gfsh to deploy JAR file containing function
- Function is automatically detected and registered
- Cannot pass configuration parameters

```
gfsh>deploy --jar=FunctionTest.jar
Member | Deployed JAR | Deployed JAR Location
----- | ----- | -----
server1 | FunctionTest.jar | <server path>/server1/vf.gf#FunctionTest.jar#1
server2 | FunctionTest.jar | <server path>/server2/vf.gf#FunctionTest.jar#1
server3 | FunctionTest.jar | <server path>/server3/vf.gf#FunctionTest.jar#1

gfsh>list functions
Member | Function
----- | -----
server1 | io.pivotal.bookshop.buslogic.GenericSumFunction
server2 | io.pivotal.bookshop.buslogic.GenericSumFunction
server3 | io.pivotal.bookshop.buslogic.GenericSumFunction
```

# Registering Function using Java API

- Performed on the Server Cache
- Could be done in the following ways
  - As part of custom GemFire server application
  - From a server-side event handler
  - From another function
- Can initialize/configure function in a variety of ways

```
public class FunctionRegistrationApplication {  
    ...  
    public void registerFunction() {  
        ...  
        MyFunction myFunction = new MyFunction();  
        FunctionService.registerFunction(myFunction);  
    }  
}
```

# Lesson Road Map

- Function Service Overview
  - Function API's
  - Registering Functions
  - **Executing Functions**
- 



# Function Execution Overview

- Functions can be executed from client or from another member of distributed system
- Examples of server-side execution:
  - Region event handler calls function as part of event handling
  - One function calls another function
- Beware of latency that may be introduced
  - Especially important when functions called from Event handlers
  - Consider non-blocking function calls (i.e. don't call `ResultCollector.getResults()`)



More detailed examples of client calling server functions covered in the next module

# Function Service APIs: Class FunctionService

- The `FunctionService` class provides the entry point into execution of user defined functions.
- Return an `Execution` object for `on*` methods
- Important methods:
  - `onRegion(Region r)`
    - Execute function on one Region
    - Execute function using data affinity information (routing keys)
  - `onMember(DistributedSystem ds)`
    - Execute function on specific peer member (plural also)
  - `onServer(Pool p)`
    - Execute function on specific server from a client (plural also)
  - `registerFunction(Function f)`
    - Registers the given Function with the FunctionService

# Function Execution Strategies

- Data independent strategy
  - In local distributed system
    - onMember ()
    - onMembers ()
  - In server's distributed system
    - onServer ()
    - onServers ()
- Data dependent strategy
  - In local and server's distributed system
    - onRegion ()

# Executing the Function

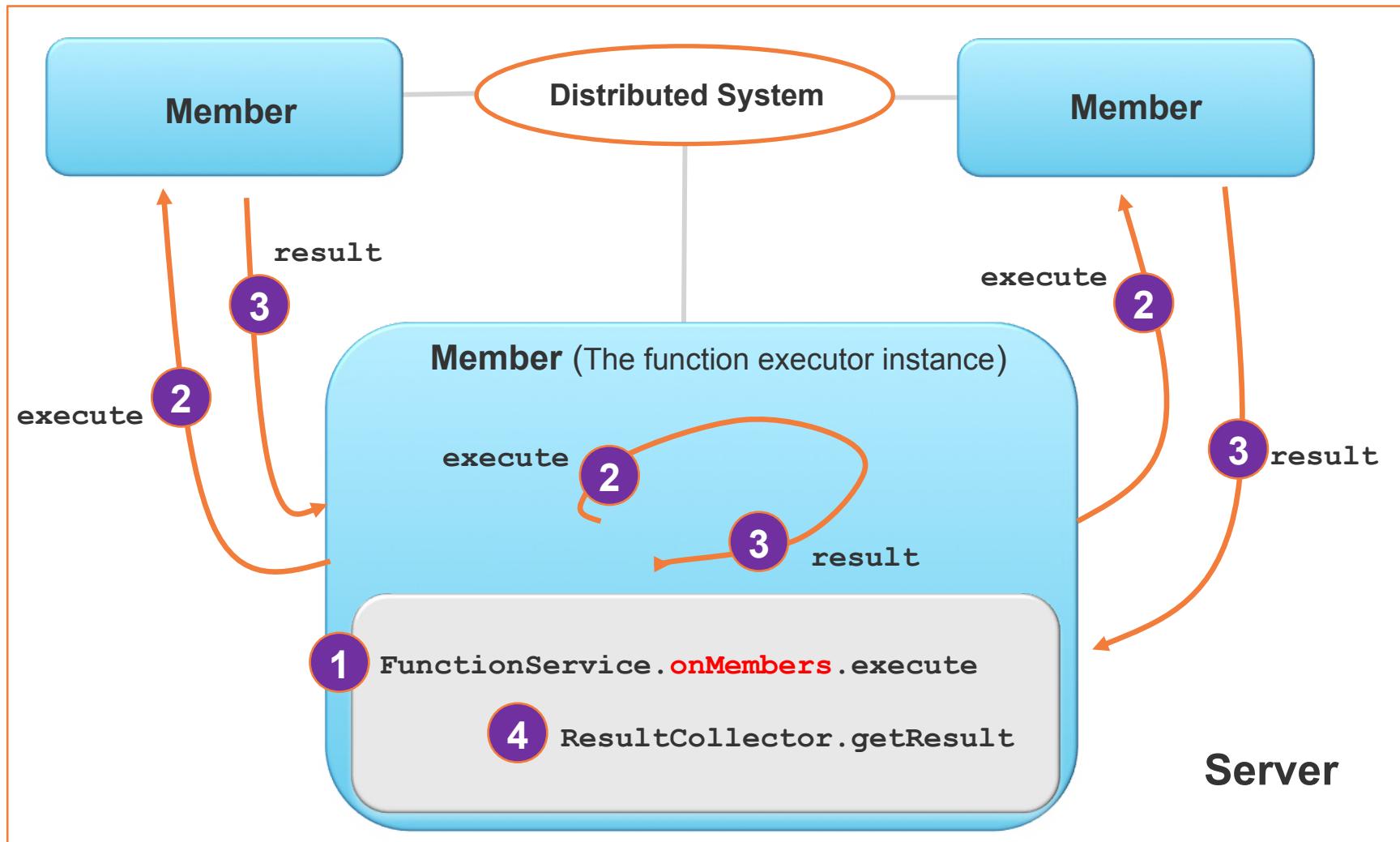
- Example of function 1 calling function 2

```
public class Function1 extends FunctionAdapter {  
    ...  
    public void execute (FunctionContext fc) {  
  
        // do stuff ...  
  
        Execution execution = FunctionService.onRegion(Customer));  
        ResultCollector rc = execution.execute("Function2") ;  
  
        // do other stuff ...  
  
        List result = (List) rc.getResult();  
    }  
}
```

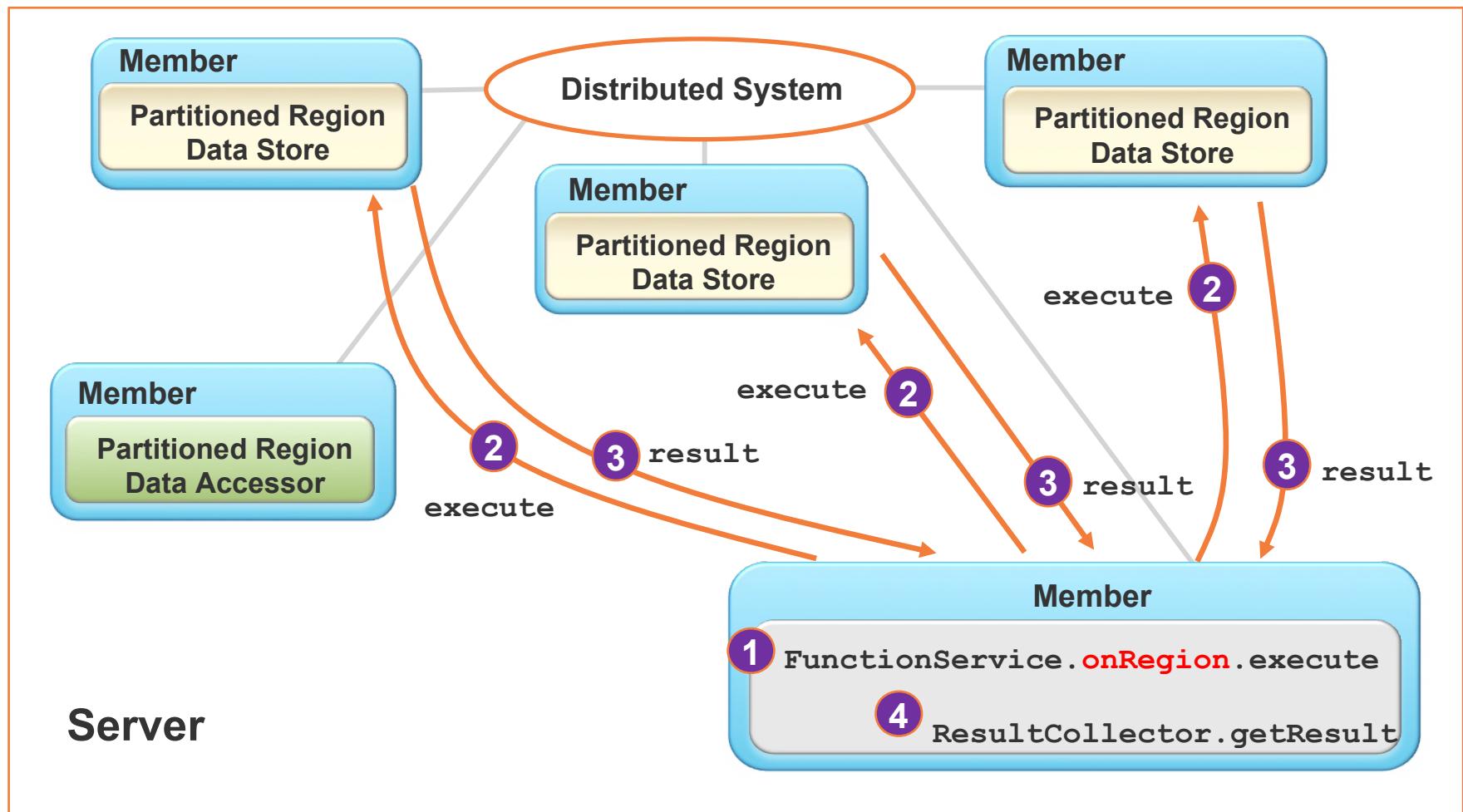
# Getting Results

- ResultCollector blocks on getResult () method until all results are received and the entire result set is returned
- Default ResultCollector returns an ArrayList
- Can extend ResultCollector to change this behavior or cover other special cases
  - Send partial results
  - Sorting of results
  - Use withCollector (ResultCollector) on the Execution object
- Additional details in the next module

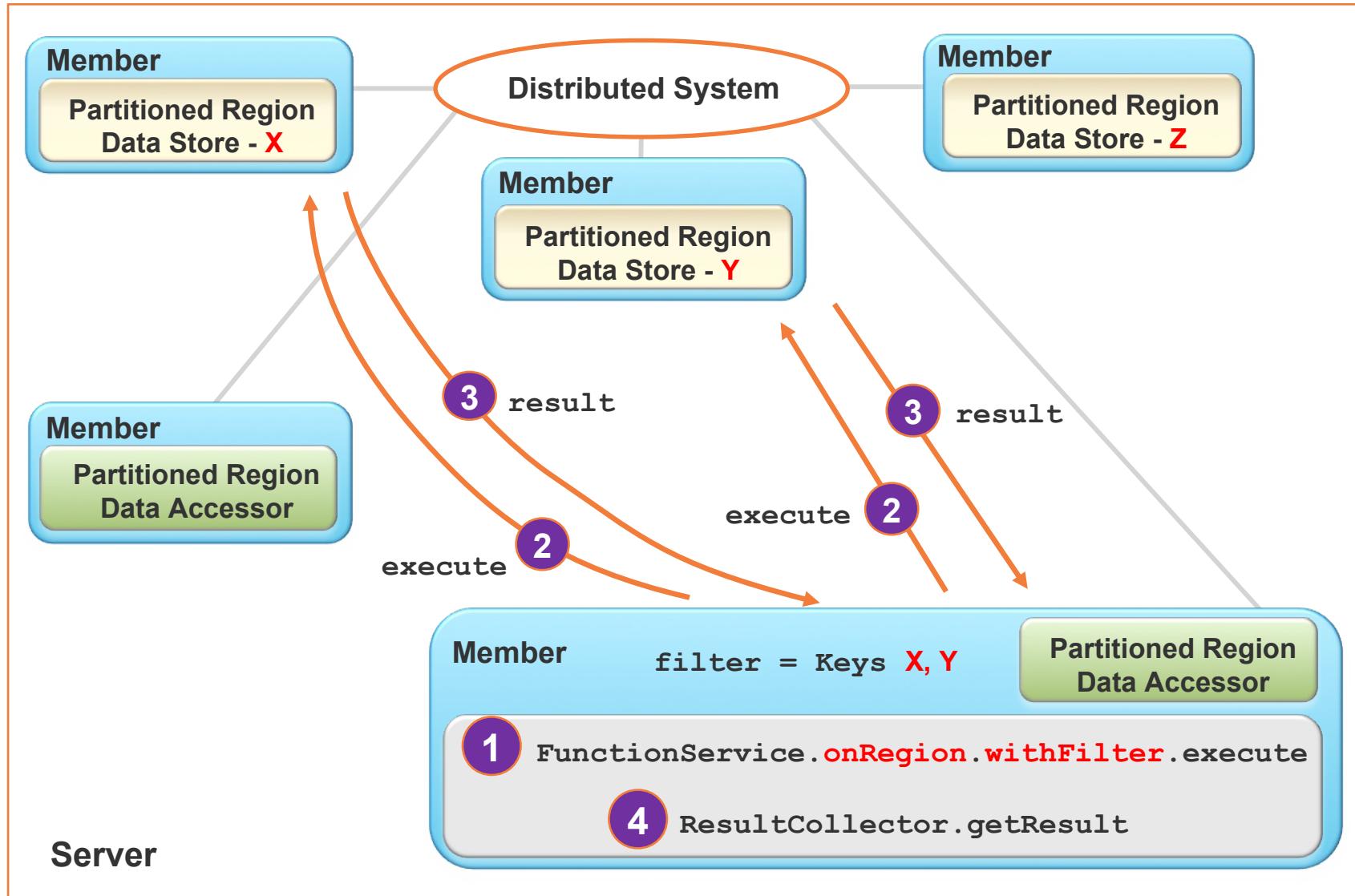
# Data-Independent Function Execution on All Members



# Data-Dependent Function Execution on All Members



# Data-Dependent Function Execution with Filter from a Peer



# Lab

## In this lab, you will

1. Develop a GemFire function
2. Register functions by using XML configuration
3. Execute the function on multiple servers
4. Invoke the function from a client application

# Review of Learner Objectives

**You should be able to do the following:**

- Describe difference between data dependent and server functions
- Write a function – both data dependent and data independent
- Register server side function

# Pivotal

BUILT FOR THE SPEED OF BUSINESS

# Function Execution

Initiating functions from the Client

# Learner Objectives

**After this lesson, you should be able to do the following:**

- Describe difference between data dependent and server functions
- Write a custom result collector
- Execute a data aware function from client code
- Understand patterns for using functions

# Lesson Road Map

- **Functions Overview**
- Distributed Execution
- Function Execution Patterns
- Customizing the ResultCollector
- HA Functions

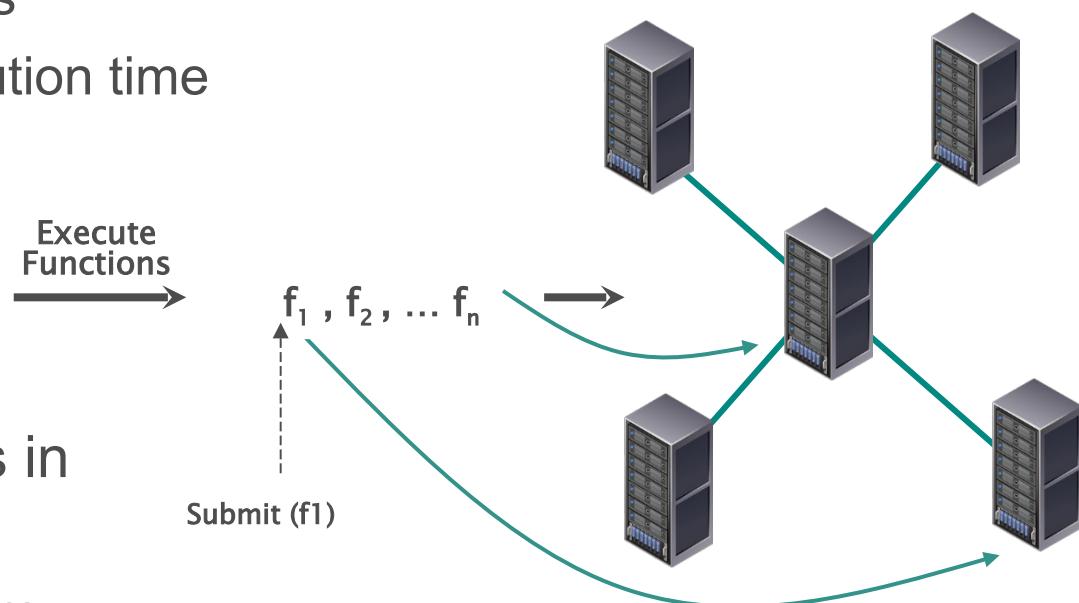


# Reminder: Where Functions are Executed

- Always run on the server
  - Which servers depends on
    - Region configuration, use of `on*` methods,filters
- Member Functions
  - A single member system
  - A subset of members (in parallel) in a distributed system
  - All the members of a distributed system (in parallel)
  - On another server system the member is a client of
- Data Aware Functions
  - Specific region and set of keys on which to run the function upon
  - Executed against a Partitioned Region on servers where the data is hosted for that region

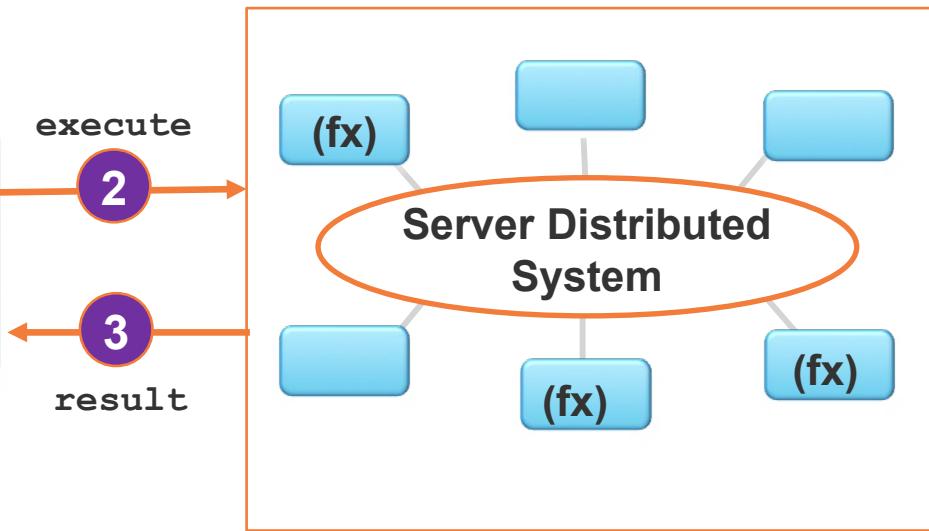
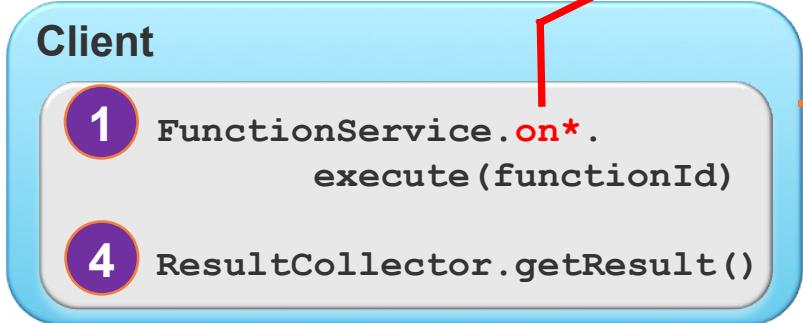
# Function Execution Approach – Data Aware

- GemFire routes the function execution to where the data resides (instead of fetching data to the executing node)
  - Reduces I/O wait times
  - Reduces overall execution time
- GemFire routes functions to the primary node hosting the data
- Implement the functions in Java
- Invoke the functions from Java, C++ or .NET or via REST



# Basic Function Execution

The `FunctionService.on*` method determines the location where the function (`fx`) will be executed.

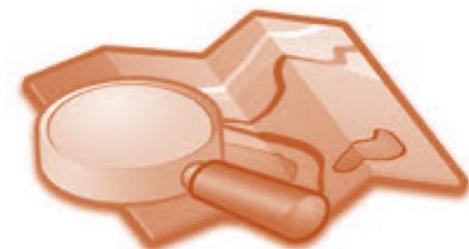


## Client Code:

```
...
Region customers = clientCache.getRegion("Customers");
Execution e = FunctionService.onRegion(customers);
ResultCollector rc = e.execute("fx");
List result = (List) rc.getResult();
...
```

# Lesson Road Map

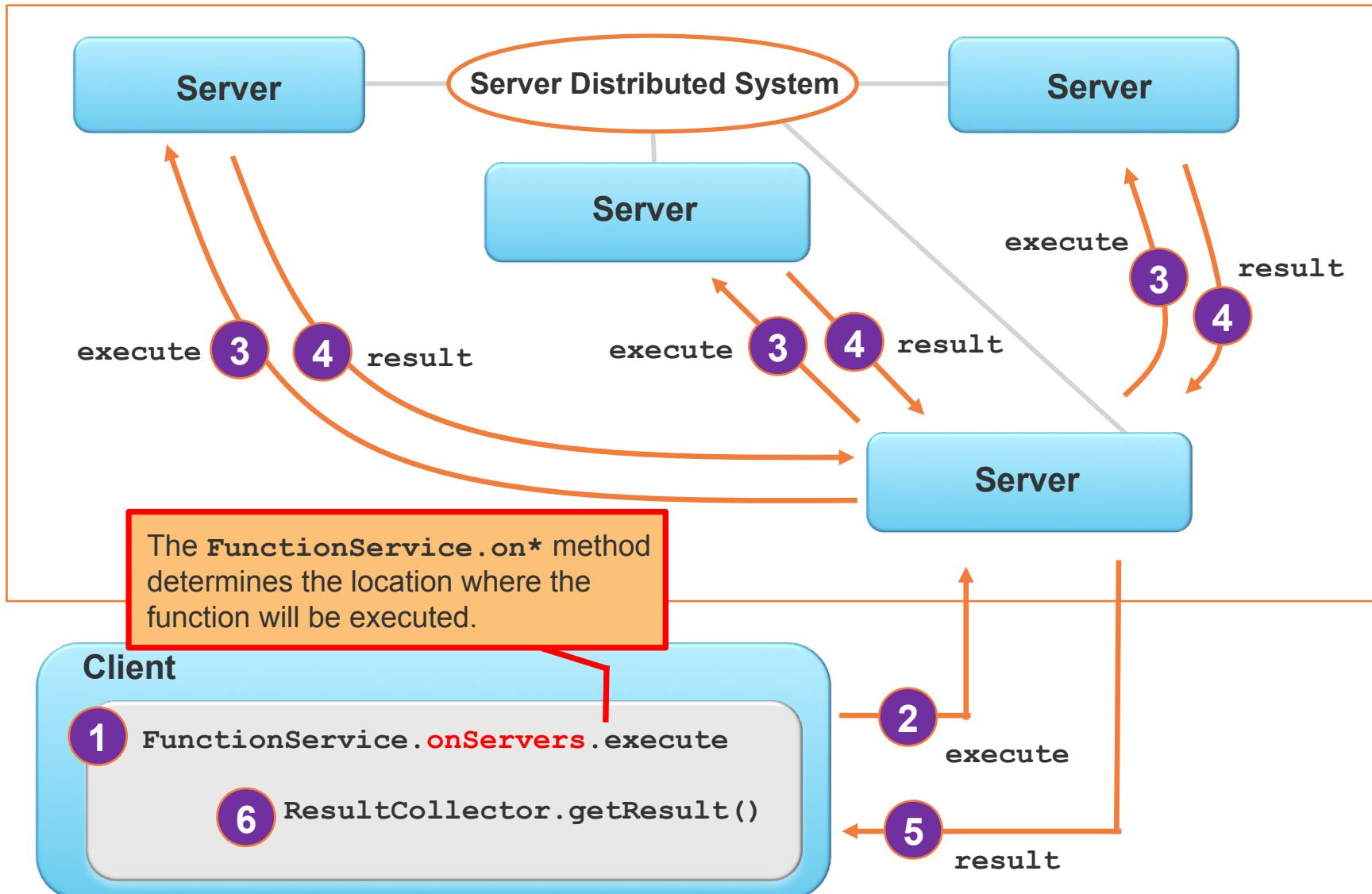
- Functions Overview
- **Distributed Execution**
- Function Execution Patterns
- Customizing the ResultCollector
- HA Functions



# Function Execution Strategies

- Data independent strategy
  - In server's distributed system
    - `onServer()` - On A server in the distributed system
    - `onServers()` - On ALL servers in the distributed system
- Data dependent strategy
  - In local and server's system having the specified region
    - `onRegion()`

# Data-Independent Function Execution from a Client



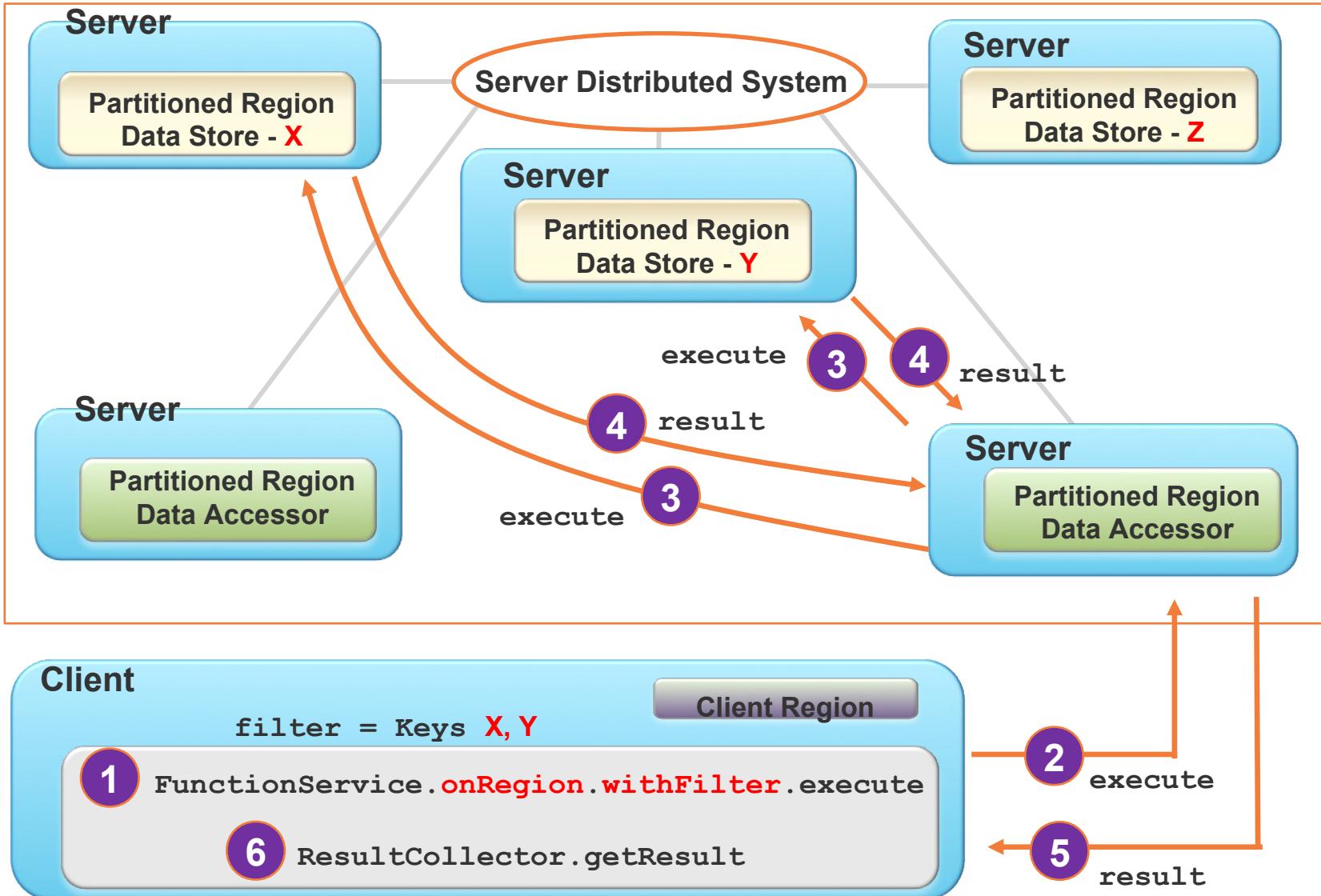
# A Full Client Side Example

```
...
// look up pool defined in client-cache.xml
Pool pool = PoolManager.find("myPool");
// Execute function on servers in the pool
Execution e = FunctionService.onServers(pool);
// Set up arguments
String arg = "A string arg";
// Pass arguments to function
ResultCollector rc = e.withArgs(arg).execute("fx");
// Wait for function results to return from all servers
List result = (List) rc.getResult();
...
...
```

- Note method chaining
- Many methods on Execution object support this

- A note on arguments...
  - Argument can be anything as long as there's agreement between function and executor of function
  - Ex String, List, Set, etc

# Data-Dependent Function Execution with Filter from a Client

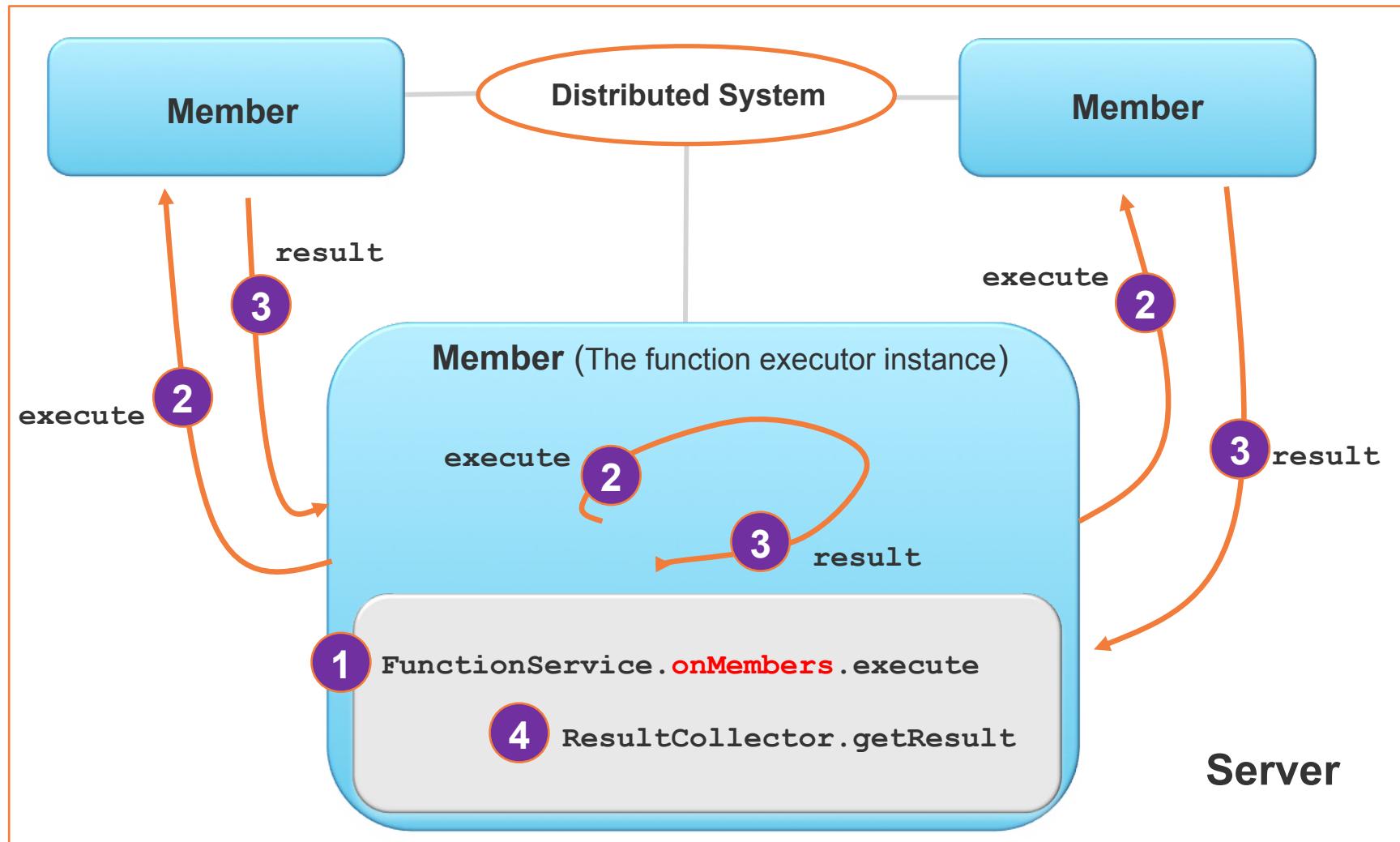


# Data Aware Example

```
...
// Get the region
Region customers = cache.getRegion("Customers");
// Execute function on servers in the pool
Execution e = FunctionService.onRegion(customers);
// Set up filter
Set keys = new HashSet();
keys.add("Key1");
keys.add("Key2");
// execute with filter
ResultCollector rc = e.withFilter(keys).execute("fx");
// Wait for function results to return from all servers
List result = (List) rc.getResult();
...
```

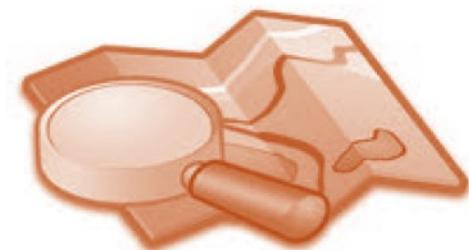
withArgs() can also  
be used here in the chain

# Data-Independent Function Execution on All Members



# Lesson Road Map

- Functions Overview
- Distributed Execution
- **Function Execution Patterns**
  - Server-side Transactions
  - Server-side Queries
  - Aggregation
- Customizing the ResultCollector
- HA Functions



# Writing a Server-side Transaction

- Ability to perform transactions on data spanning co-located regions
  - Generally makes sense when performing operation on a specific set of co-located data that has a one-to-many relationship
  - Can be better performance than invoking from the client as the actual data remains local to the member
  - Use a combination of withFilter() and withArgs() to pass required data to function
    - Use withFilter() to specify the key for the ‘one’ side of the one-to-many relationship
    - Use withArgs() to pass the rest of the data

# Calling Transactional Function

- Using the FunctionService to run a function against a region ...

```
public class ClientApplication {  
    public static void main(String [] args){  
        // Set up client cache and region  
        Region custRegion = ...  
  
        // Set up calling arguments  
        Integer custToUpdate = new Integer(1234) ;  
        OrderId orderToUpdate = new OrderId(555,custToUpdate) ;  
        Set filter = new HashSet() ;  
        filter.add(custToUpdate) ;  
  
        // Make function call  
        FunctionService.onRegion(custRegion).withFilter(filter)  
            .withArgs(orderToUpdate).execute("CustTxFunction") ;  
    }  
}
```

# Transactional Function Example

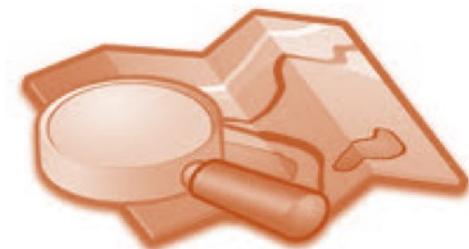
Server

```
public class CustomerTransactionFunction implements Function {  
    ...  
    public void execute(ExecutionContext fc){  
        if (fc instanceof RegionFunctionContext) {  
            RegionFunctionContext rfc = (RegionFunctionContext) fc;  
            Region<Integer, Customer> custRegion = rfc.getDataSet();  
            Region<OrderId, Order> orderRegion =  
                custRegion.getRegionService().getRegion("Order");  
            CacheTransactionManager txMgr =  
                CacheFactory.getAnyInstance().getCacheTransactionManager();  
            Integer custToUpdate = (Integer) rfc.getFilter().iterator().next();  
            OrderId orderToUpdate = (OrderId) rfc.getArguments();  
            txMgr.begin();  
            try {  
                // Update Customer and Order, then ...  
                txMgr.commit();  
            } catch (TransactionException tex) {  
                txMgr.rollback();  
            }  
        } else {  
            throw new FunctionException("Must be called as onRegion function");  
        }  
    }  
}
```

Pivotal™

# Lesson Road Map

- Functions Overview
- Distributed Execution
- **Function Execution Patterns**
  - Server-side Transactions
  - **Server-side Queries**
  - Aggregation
- Customizing the ResultCollector
- HA Functions



# Server-side Queries

- Joins across partitioned regions not supported from client
- Executing query from a server member allows equi-joins on co-located regions

# Calling Query Function

```
public class ClientApplication {
    public static void main(String [] args) {
        // Set up client cache and region
        Region custRegion = ...

        // Set up equi-join query on co-located partitioned regions
        String queryString = "SELECT o FROM /Customer c, /Order o" +
            " WHERE c.state = 'CO' AND c.customerId = o.customerId";

        // Make function call
        ResultCollector results = FunctionService.onRegion(custRegion) .
            .withArgs(queryString).execute("CustomerQueryFunction");
        // Blocking call to wait for servers to return
        ArrayList resultList = (ArrayList) results.getResult();
        List queryResults = new ArrayList();
        for (Object obj : resultList) {
            if (obj != null) { // Server may return null - next slide
                queryResults.addAll((ArrayList) obj);
            }
        }

        // Do something with the contents of queryResults
    }
}
```

# Query Function Example

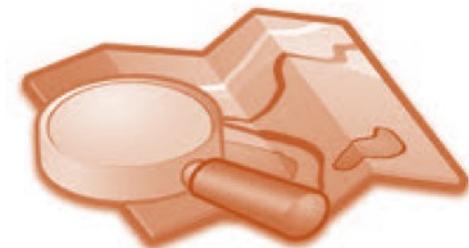
Server

```
public class CustomerQueryFunction implements Function {  
    ...  
    public void execute(ExecutionContext fc){  
        if (fc instanceof RegionFunctionContext) {  
            RegionFunctionContext rfc = (RegionFunctionContext) fc;  
            Cache cache = CacheFactory.getAnyInstance();  
            QueryService queryService = cache.getQueryService();  
            String queryString = (String) rfc.getArguments();  
            try {  
                Query query = queryService.newQuery(queryString);  
                // Only execute the query on the data set for local partitioned data  
                SelectResults results = (SelectResults) query.execute(rfc);  
                // Send all results at once  
                rfc.getResultSender().sendResult((ArrayList) (results).asList());  
                rfc.getResultSender().lastResult(null);  
            } catch (QueryException qex) {  
                throw new FunctionException("Error processing query: " + qex);  
            }  
        }  
    }  
}
```

Causes query to be run on region specified by RegionFunctionContext

# Lesson Road Map

- Functions Overview
- Distributed Execution
- **Function Execution Patterns**
  - Server-side Transactions
  - Server-side Queries
  - **Aggregation**
- Customizing the ResultCollector
- HA Functions



# Aggregation Operations

- Functions can be used for a variety of aggregation and calculation operations
  - Simple summation
  - Average
  - More complex analytics of data where the data lives

# Example Generic Sum Function

```
public class ClientApplication {  
    public static void main(String [] args){  
        // Set up client cache and region  
        Region ordersRegion = ...  
  
        // Make function call  
        FunctionService.onRegion(ordersRegion).withArgs("totalPrice")  
            .execute("GenericSumFunction");  
        // Blocking call to wait for servers to return  
        ArrayList resultList = (ArrayList) results.getResult();  
  
        // Process results  
    }  
}
```

# Example Generic Sum Function

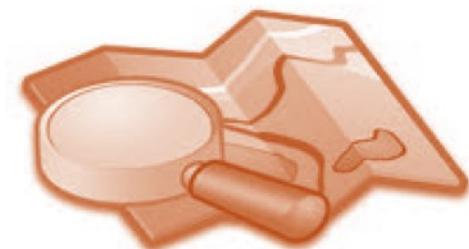
Server

```
public class GenericSumFunction implements Function {  
    ...  
    public void execute(FunctionContext fc){  
        if (fc instanceof RegionFunctionContext) {  
            RegionFunctionContext rfc = (RegionFunctionContext) fc;  
            String sumField= (String) rfc.getArguments();  
            Region<Object, PdxInstance> localData =  
                PartitionRegionHelper.getLocalDataForContext(rfc);  
  
            // Assume field is decimal (Double) type  
            BigDecimal sum = BigDecimal.ZERO;  
            for (PdxInstance item : localData.values()) {  
                Object field = item.getField(sumField);  
                if (field instanceof Double) {  
                    sum = sum.add(BigDecimal.valueOf((Double) field));  
                }  
            }  
            rfc.getResultSender().lastResult(sum);  
        } else {  
            throw new FunctionException("Must be called as onRegion function");  
        }  
    }  
}
```

Pivotal™

# Lesson Road Map

- Functions Overview
- Distributed Execution
- Data Aware Functions
- **Customizing the ResultCollector**
- HA Functions



# ResultCollector Behavior

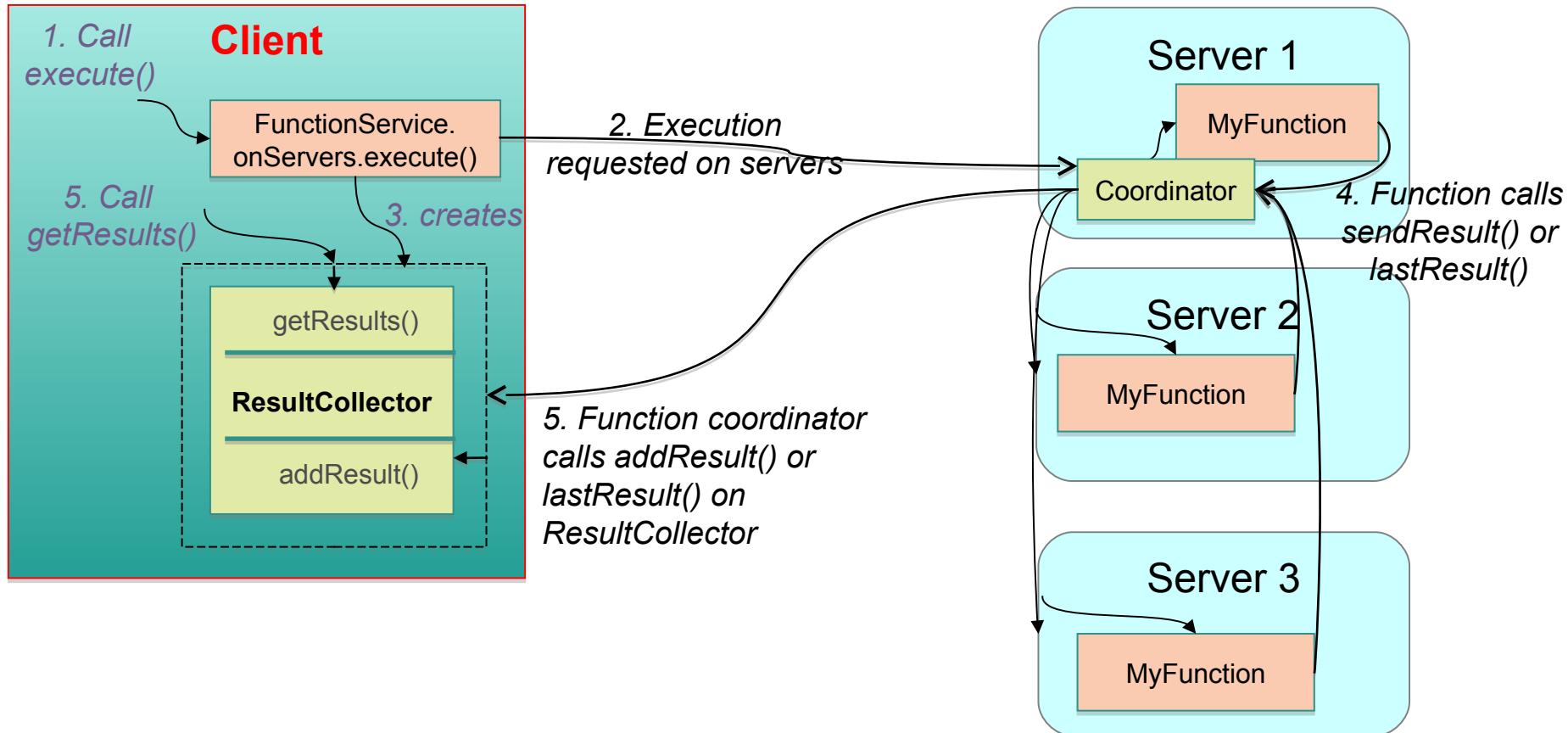
Client

- Behind the scenes – collects results returned by functions
- getResult() returns ArrayList by default
  - Blocks until all results are returned
  - Entries in ArrayList sorted by order of insertion

Name of function as registered on servers

```
...
ResultCollector rc = execution.execute("fx");
List result = (List) rc.getResult();
...
```

# GemFire Function Execution and ResultCollector



# Custom ResultCollector in 3 Steps

1. Write a class implementing ResultCollector interface
  - implement addResult() and endResult(), which are called by GemFire when function sends results back
  - implement getResult() to define how to return results collected
  - Note: GemFire will manage concurrency & collection of results from servers executing the function
2. *Optionally* define HA behavior: implement clearResults()
  - Called by GemFire before re-execution on failure
3. Register custom ResultCollector

# Custom ResultCollector Example

```
public class MyCollector
    implements ResultCollector<Serializable, Serializable> {
    ArrayList<Serializable> result = new ArrayList<Serializable>();

    public void addResult
        (DistributedMember memberId, Serializable resultOfSingleExecution) {
        // Populate result with resultOfSingleExecution or write conditional code
        // to perform filtering
    }

    public Serializable getResult() throws FunctionException {
        // Code to return result - or you could perform a sort before returning
    }

    // Used in HA functions - called by GemFire before re-executing functions
    public void clearResult() { result.clear(); }

    // Optional: perform post processing after function execution is complete
    public void endResults() {}
}
```

Additional overloaded method: `getResults(long timeout TimeUnit units)`

# Custom ResultCollector Example

Client

- Now tell the Execution to use your custom collector:

```
...
Pool pool = PoolManager.find("myPool");
Execution e = FunctionService.onServers(pool);

// ... other necessary setup

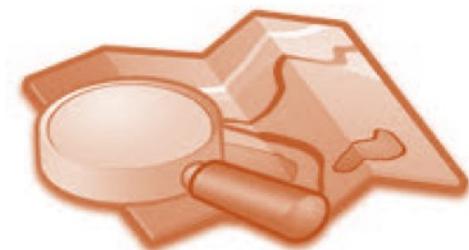
ResultCollector rc = e.withCollector(new MyCollector())
    .execute("fx");
List result = (List) rc.getResult();
...
```



client application

# Lesson Road Map

- Functions Overview
- Distributed Execution
- Data Aware Functions
- Customizing the ResultCollector
- **HA Functions**



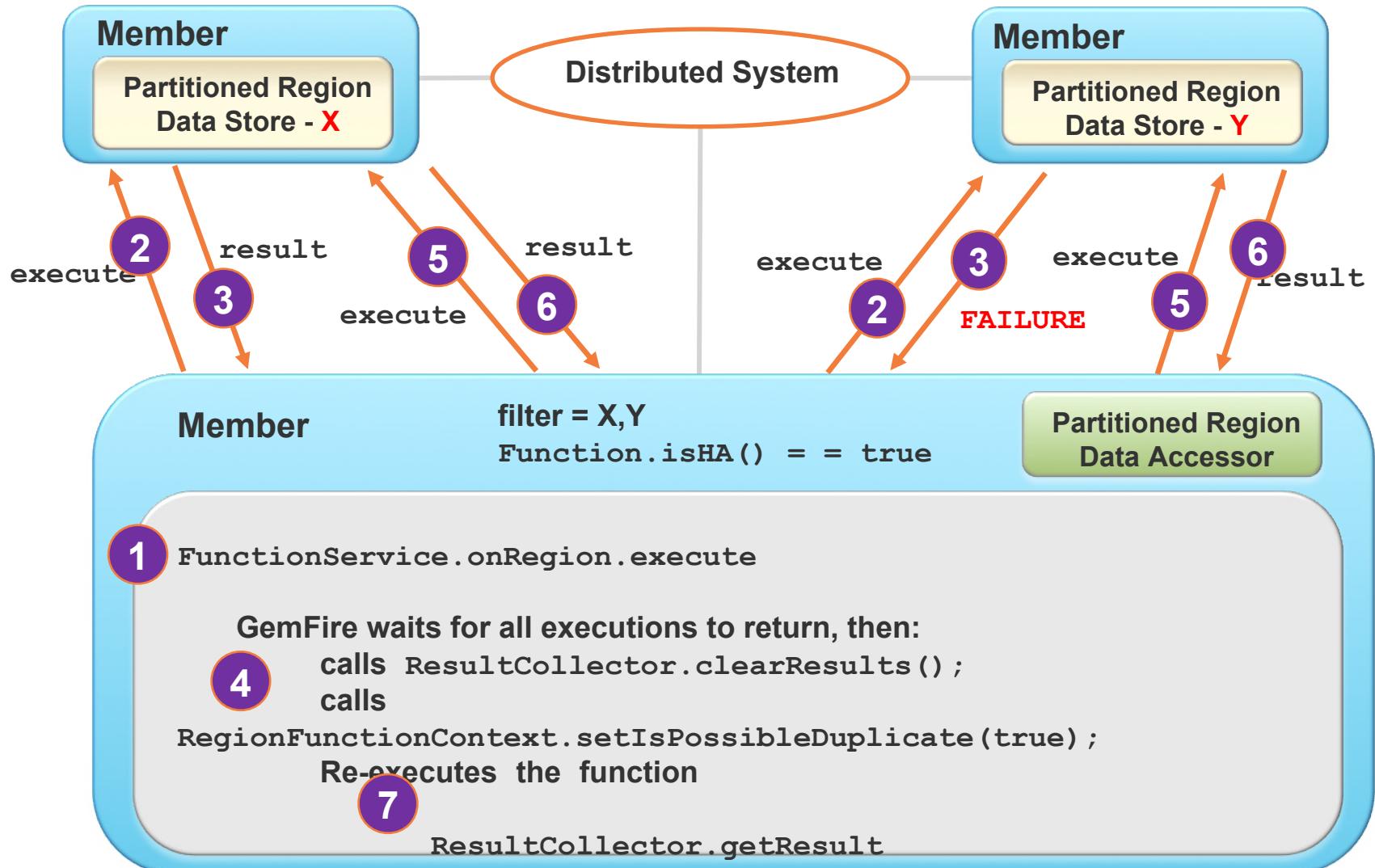
# Highly Available Functions

- During an execution error, or member crash while executing functions, the system responds as follows:
  - Waits for all calls to return
  - Sets a boolean indicating a re-execution is being done
  - Calls the result collector's `clearResults` method
  - Executes the function
- Function must be coded and configured for this, and client must invoke the `getResults` method.
  - For clients, retries set by Pool `retryAttempts`
  - Members – retry until success or no data – use with caution!



You can specify whether the function is eligible for re-execution (in case of failure) by implementing the `Function.isHA()` method.

# Highly Available Data-Dependent Function



# Lab

**In this lab, you will:**

1. Execute a function on multiple servers
2. Invoke the function from a client application
3. Develop a custom function result collector

# Pivotal

BUILT FOR THE SPEED OF BUSINESS

# Spring Introduction

Spring Configuration, Dependency Injection  
and Bean Creation

# Topics in this session

- **Why Spring?**
- Configuration using Spring
- Bean Creation

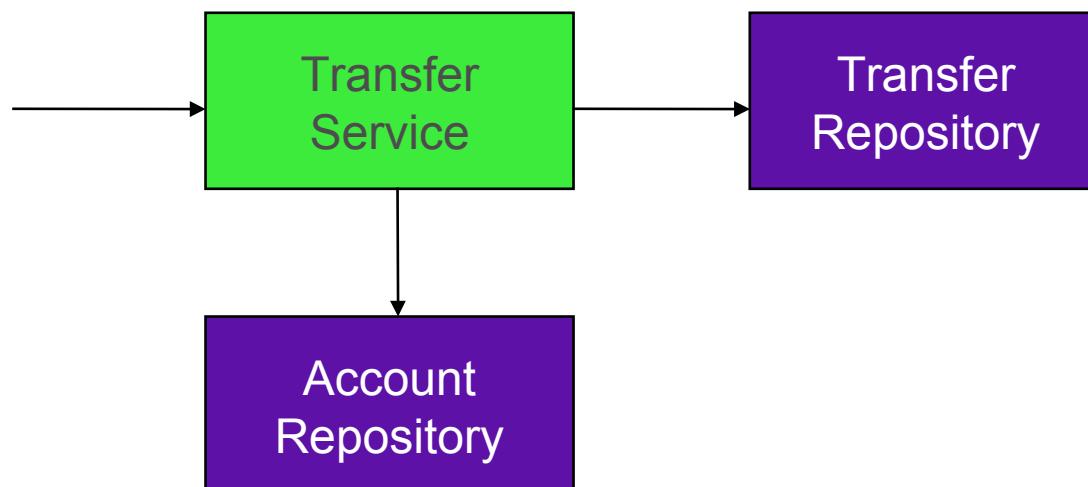
# Goal of the Spring Framework

- Provide comprehensive infrastructural support for developing enterprise Java™ applications
  - Spring deals with the plumbing
  - So you can focus on solving the domain problem
- Key Principles
  - DRY – Don't Repeat Yourself
  - SoCs – Separation of Concerns

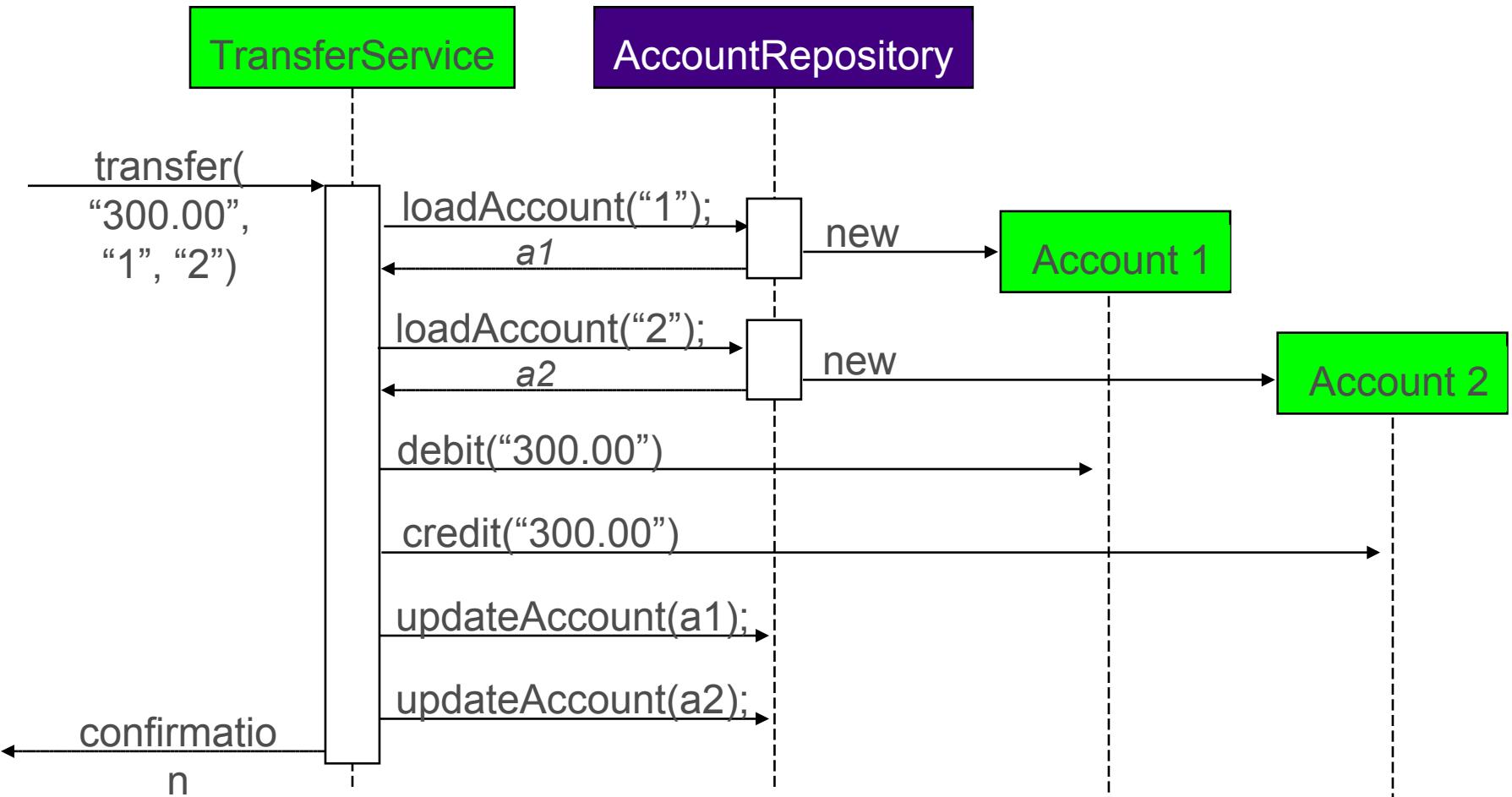


# Application Configuration

- A typical application system consists of several parts working together to carry out a use case



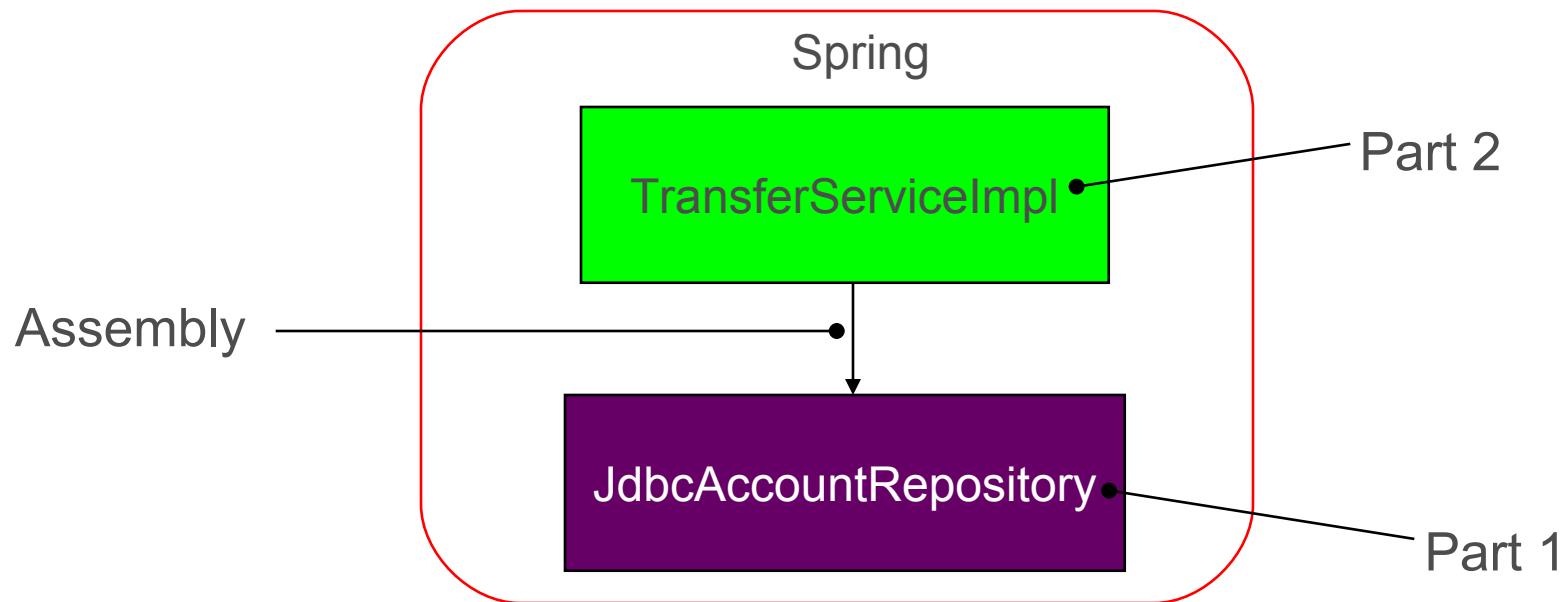
# Example: Money Transfer System



# Spring's Configuration Support

- Spring provides support for assembling such an application system from its parts
  - Parts do not worry about finding each other
  - Any part can easily be swapped out

# Money Transfer System Assembly



```
(1) repository = new JdbcAccountRepository(...);  
(2) service = new TransferServiceImpl();  
(3) service.setAccountRepository(repository);
```

# Parts are Just Plain Old Java Objects

```
public class JdbcAccountRepository implements  
    AccountRepository {  
    ...  
}
```

Implements a service/business interface

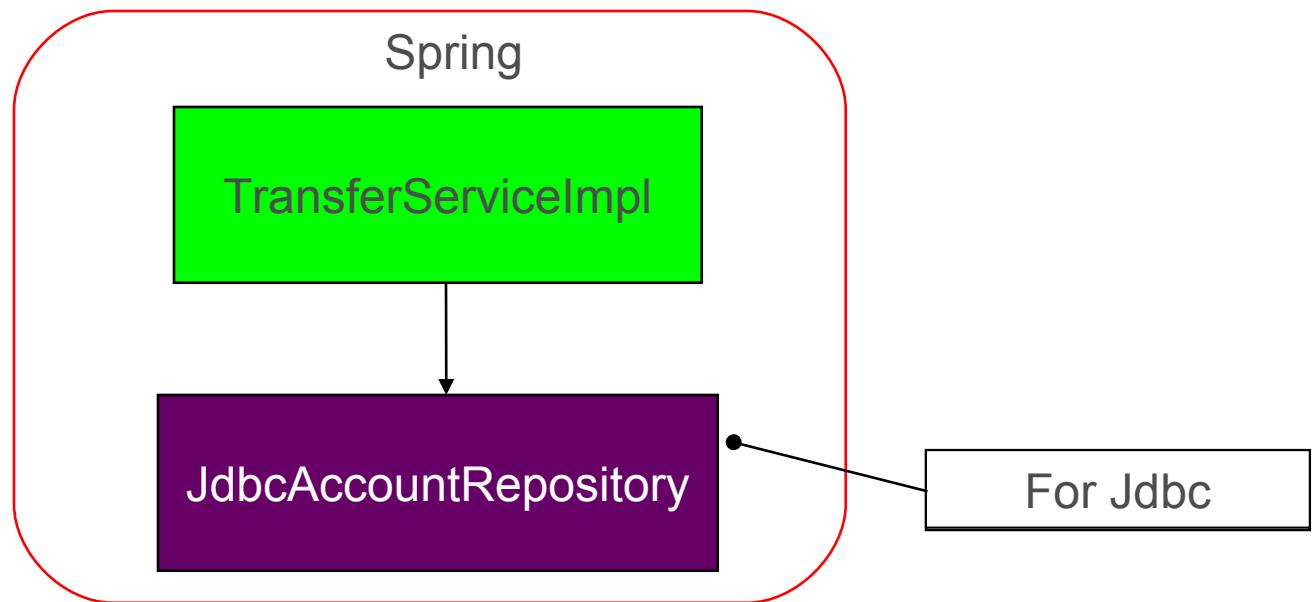
Part 1

```
public class TransferServiceImpl implements TransferService {  
    private AccountRepository accountRepository;  
  
    public void setAccountRepository(AccountRepository ar) {  
        accountRepository = ar;  
    }  
    ...  
}
```

Depends on *interface*;  
conceals complexity of implementation;  
allows for swapping out implementation

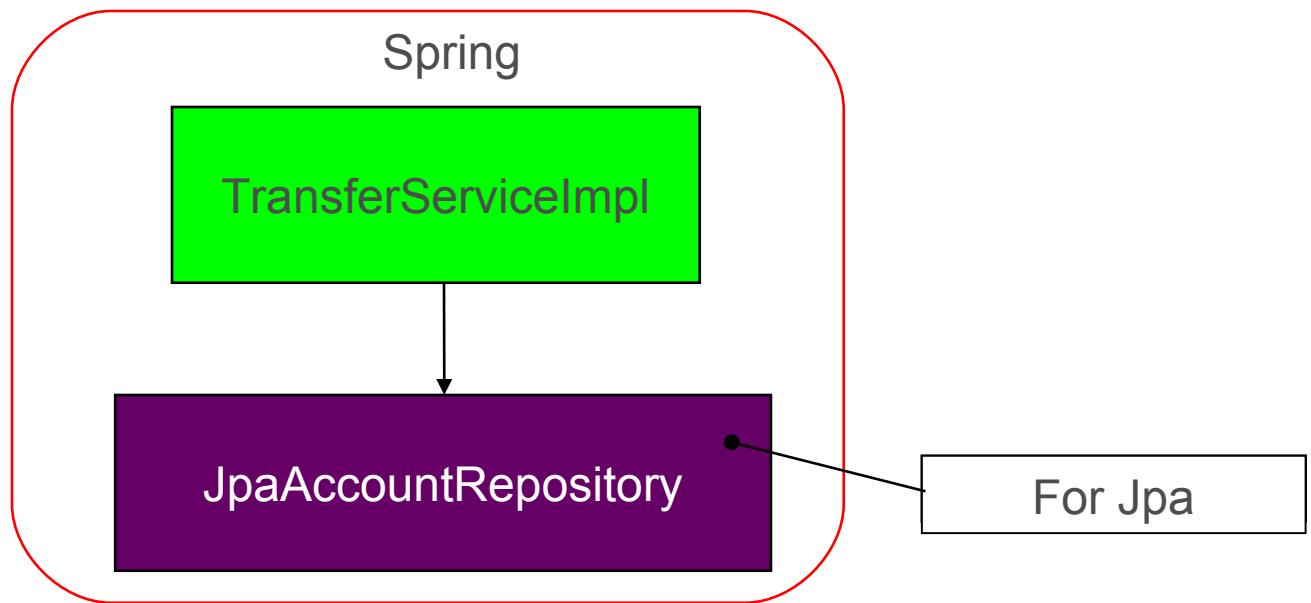
Part 2

# Swapping Out Part Implementations



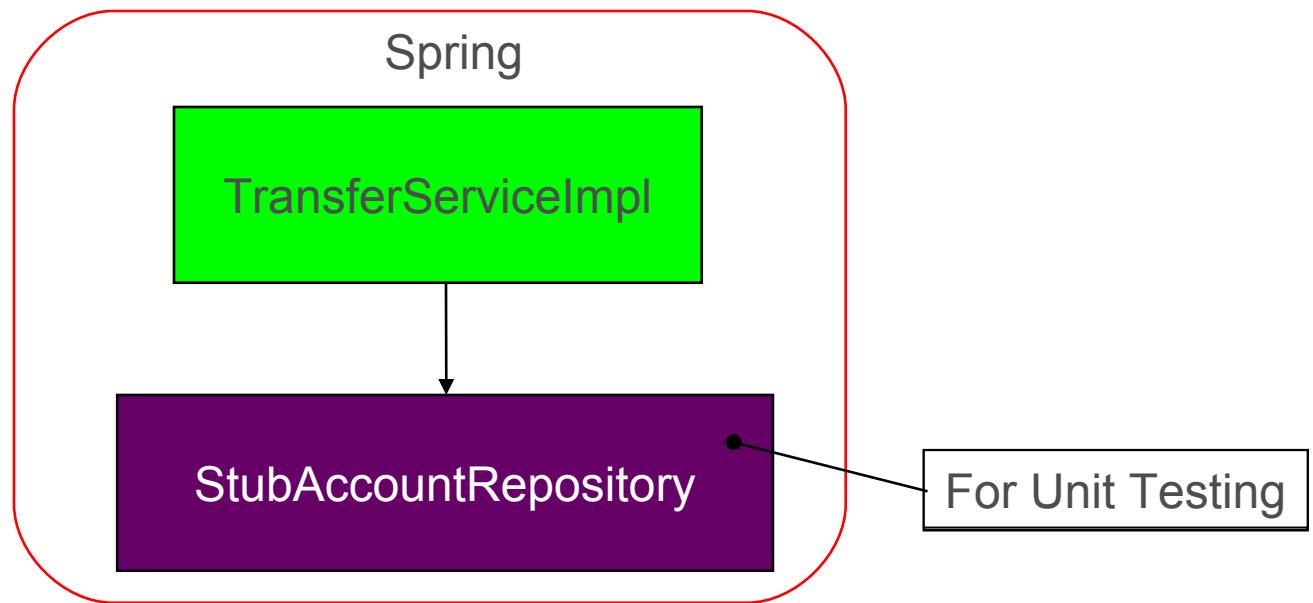
```
(1) new JdbcAccountRepository(...);  
(2) new TransferServiceImpl();  
(3) service.setAccountRepository(repository);
```

# Swapping Out Part Implementations



```
(1) new JpaAccountRepository(...);  
(2) new TransferServiceImpl();  
(3) service.setAccountRepository(repository);
```

# Swapping Out Part Implementations

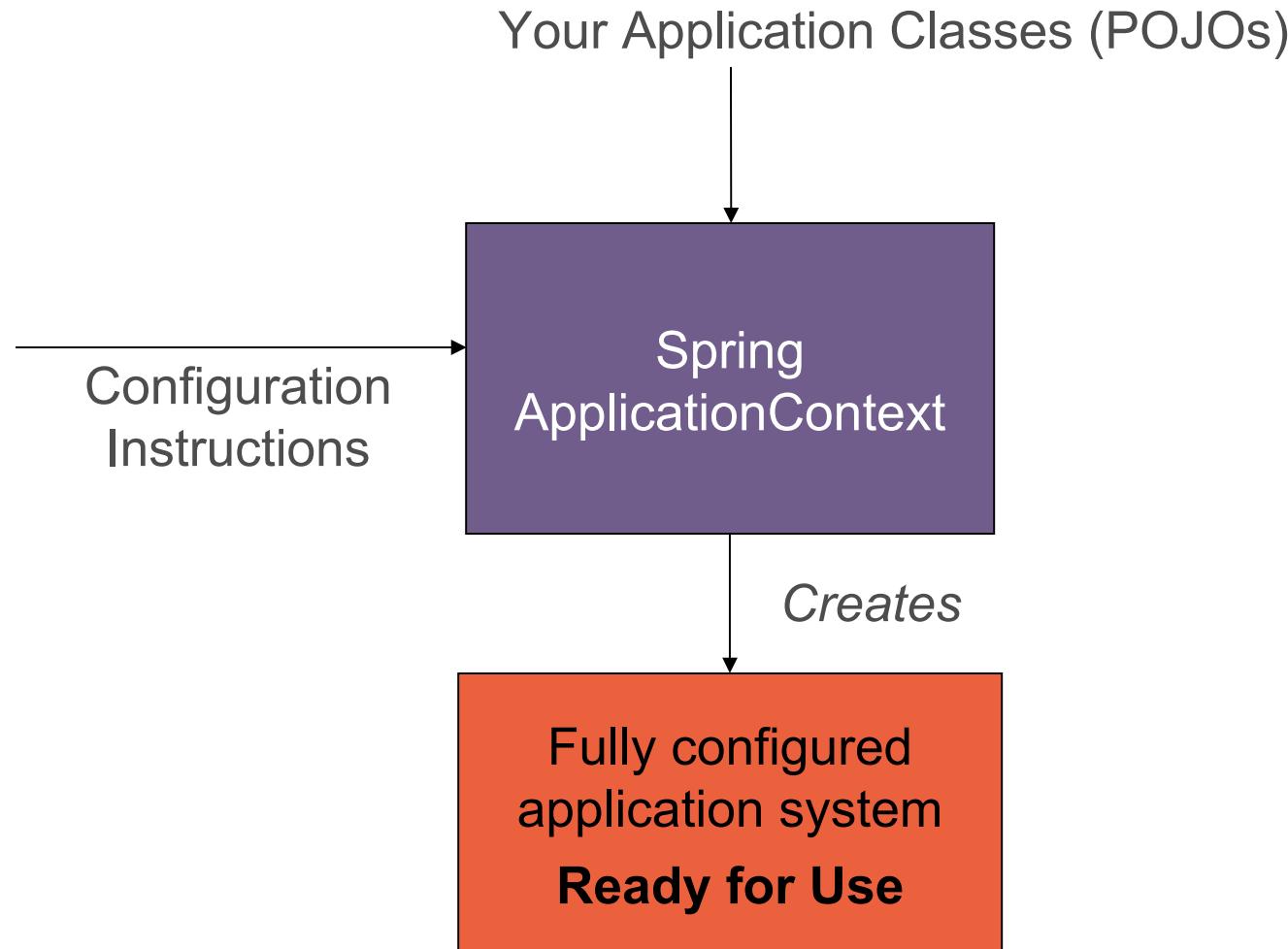


```
(1) new StubAccountRepository();
(2) new TransferServiceImpl();
(3) service.setAccountRepository(repository);
```

# Topics in this session

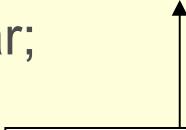
- Why Spring?
- Configuration using Spring
- Bean Creation

# How Spring Works



# Your Application Classes

```
public class TransferServiceImpl implements TransferService {  
    public TransferServiceImpl(AccountRepository ar) {  
        this.accountRepository = ar;  
    }  
    ...  
}
```



Needed to perform money transfers between accounts

```
public class JdbcAccountRepository implements AccountRepository {  
    public JdbcAccountRepository(DataSource ds) {  
        this.dataSource = ds;  
    }  
    ...  
}
```



Needed to load accounts from the database

# Configuration Instructions – Java

```
@Configuration  
public class ApplicationConfig {  
    @Bean public TransferService transferService() {  
        return new TransferServiceImpl( accountRepository() );  
    }  
    @Bean public AccountRepository accountRepository() {  
        return new JdbcAccountRepository( dataSource() );  
    }  
    @Bean public DataSource dataSource() {  
        DataSource dataSource = new BasicDataSource();  
        dataSource.setDriverClassName("org.postgresql.Driver");  
        dataSource.setUrl("jdbc:postgresql://localhost/transfer" );  
        dataSource.setUser("transfer-app");  
        dataSource.setPassword("secret45" );  
        return dataSource;  
    }  
}
```

**Dependency injection**

**Dependency injection**

**Bean ID defaults to method name or use @Bean(name=...)**

# Configuration Instructions – XML

```
<beans>
```

```
  <bean id="transferService" class="com.acme.TransferServiceImpl">
    <constructor-arg ref="accountRepository" />
```

```
  </bean>
```

Dependency injection

```
  <bean id="accountRepository" class="com.acme.JdbcAccountRepository">
```

```
    <constructor-arg ref="dataSource" />
```

```
  </bean>
```

Dependency injection

```
  <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
```

```
    <property name="driverClassName" value="org.postgresql.Driver" />
```

```
    <property name="url" value="jdbc:postgresql://localhost/transfer" />
```

```
    <property name="user" value="transfer-app" />
```

```
    <property name="password" value="secret45" />
```

```
  </bean>
```

Bean ID specified explicitly via id attribute

```
</beans>
```

# Implicit Configuration using Annotations

- Annotation-based configuration *within* bean-class

```
@Component ( name="transferService" )  
public class TransferServiceImpl implements TransferService {  
    @Autowired  
    public TransferServiceImpl(AccountRepository repo) {  
        this.accountRepository = repo;  
    }  
}
```

Annotations embedded *within* POJOs

Bean ID

Dependency injection

```
@Configuration  
@ComponentScan ( "com.bank" )  
public class AnnotationConfig {  
    // No bean definition needed any more  
}
```

Find `@Component` classes within designated (sub)packages

`<context:component-scan base-packages="com.bank">`

# Creating and Using the Application

```
// Create the application from the configuration
ApplicationContext context =
    SpringApplication.run( ApplicationConfig.class );

// Look up the application service interface
TransferService service =
    context.getBean("transferService", TransferService.class);

// Use the application
service.transfer(new MonetaryAmount("300.00"), "1", "2");
```



**NOTE:** This code uses Spring Boot's *SpringApplication.run()*  
You may be used to seeing older code like this:

```
context = new ClassPathXmlApplicationContext("beans.xml");
```

# Importing Configurations, Mixing Styles

- Importing other configurations

```
@Configuration  
@Import( SecurityConfig.class )  
@ImportResource( { "config/dao-config.xml",  
                  "classpath:config/service-config.xml",  
                  "file:postgres-infrastructure-config.xml" } )  
public class ApplicationConfig { ... }
```

- Use prefixes to specify resource locations
  - classpath:
  - file:
  - http:

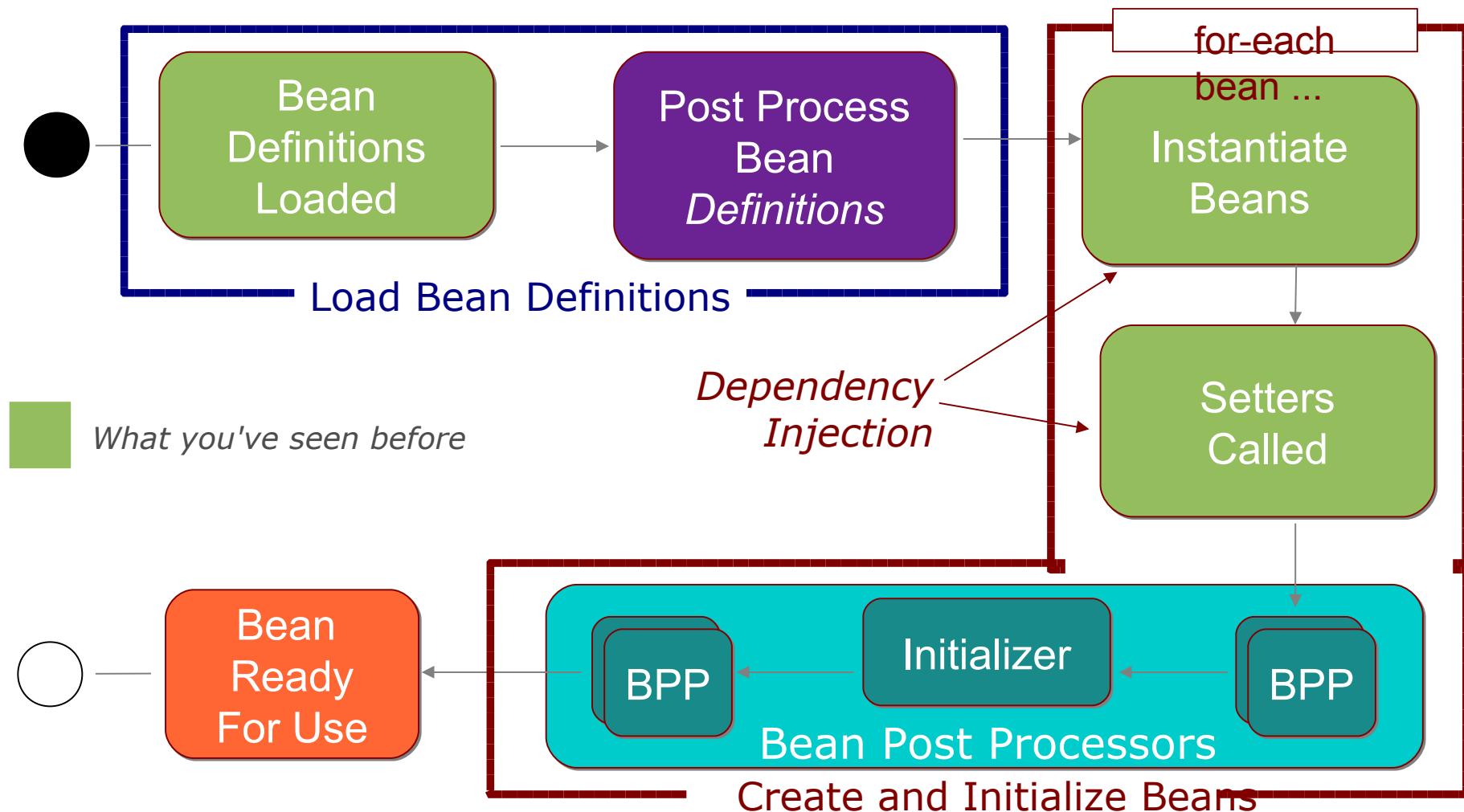


Prefixes can be used *anywhere* Spring needs to find resources

# Topics in this session

- Why Spring?
- Configuration using Spring
- **Bean Creation**

# Bean Initialization Steps



# Lifecycle Activities

- Bean Factory Post-Processor
  - Post-processes *definitions before creation*
    - PropertyPlaceholderConfigurer
- Bean Post-Processor
  - Post processes the actual beans
  - Extra initialization
    - @PostConstruct, init-method
  - Add behavior
    - Wrap bean in an AOP proxy
    - Security, @Transactional ...
    - Never cast a bean to its implementation



# Creating Bean Instances

- Each bean is eagerly instantiated by default
  - Created in right order with its dependencies injected
- After dependency injection each bean goes through a **post-processing** phase
  - Further configuration and initialization may occur
  - Initialization is a common special-case of a Bean Post Processor (BPP)



# The *BeanPostProcessor* Extension Point

- An important extension point in Spring
  - Can modify bean instances *in any way*
  - **Powerful enabling feature**
  - Spring provides many BPPs out-of-the-box



# BPPs Enable Proxies

- Spring *doesn't always* give you the bean you asked for

```
service = context.getBean("transferService", TransferService.class);
```

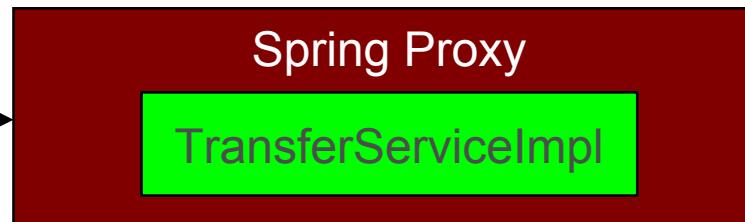
- Sometimes the bean is just your raw object

```
service.transfer("$50", "1", 2")
```



- Or your bean has been wrapped in a *proxy* by a *BPP*
  - To implement extra behavior (*aspects*)

```
service.transfer("$50", "1", 2")
```



# Developing GemFire Applications with Spring Gemfire

# Learner Objectives

After completing this lesson, you should be able to

- Understand and use Spring GemFire to configure an application
- Understand how to configure CacheListener and register interest
- Understand how to create Repository definitions for GemFire access
- Understand how to configure simplified function access

# Spring & Big Data

- **Spring Framework Overview**
- Spring Data Overview
- Spring Data GemFire

# Spring & Dependency Injection

- Spring widely used Java framework, created to simplify assembly of components
  - Partly as a reaction to complexity of old EJB
- Component decoupled from implementation of its dependencies
  - Component class
    - Uses interface types for its dependencies
    - Provides constructor argument or setter for each dependency
  - Spring injects implementation according to configuration
  - Technique known as Dependency Injection (DI)

# Spring Configuration Options

- Configuration of dependencies can be done with any combination of
  - XML files
    - non-intrusive, no Java to rebuild when changing
  - Annotations in component class itself
    - intrusive, but can be convenient
  - Separate Java `@Configuration` classes
    - non-intrusive, 100% Java

# Spring Projects

- Apart from the core Spring Framework, there are many other Spring projects

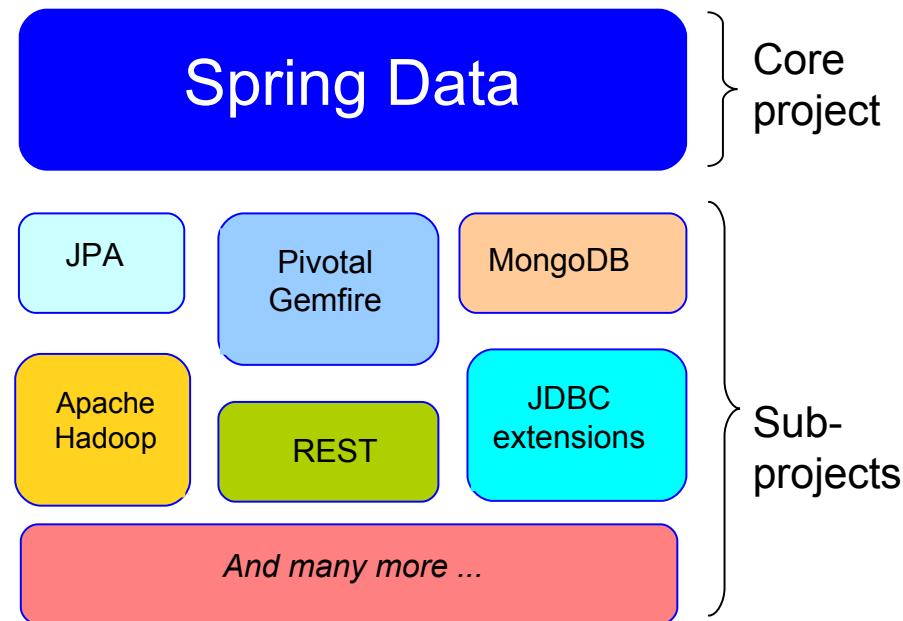
Project	Description
Spring Data	An umbrella project for related sub-projects to simplify and unify common data access patterns for relation and non-relational data repositories
Spring Integration	Spring implementation of the Enterprise Integration Patterns
Spring Batch	non-interactive, large scale processes with varied data input & output
Spring Security	web-level and method-level security
Spring Web Flow	conversational interaction with web application users

# Spring & Big Data

- Spring Framework Overview
- **Spring Data Overview**
- Spring Data GemFire

# What is Spring Data?

- Consists of several sub-projects related to NoSQL DBs
  - Sub-projects provide
    - simplified configuration of DB components
    - helper template class
    - integration with Spring transaction management, where appropriate



# Spring Data Commons

- Commons project defines common repository support
  - Specific implementations in most of the sub-projects
  - Common interface hierarchy
    - `Repository<T, ID extends Serializable>`
    - `CrudRepository<T, ID extends Serializable>`
    - `PagingAndSortingRepository<T, ID extends Serializable>`
  - To use, simply extend interface
    - Naming conventions, or annotations define query methods
    - Spring Data provides implementation
  - Avoids boilerplate repository code
  - Avoids limitations of code-generation frameworks (Grails)

# Spring Data Sub-projects

DI = Dependency Injection

- The following are all sub-projects of Spring Data
  - Full details in respective Spring Data web pages
    - Spring Data URL: <http://www.springsource.org/spring-data>

Project	Description
JPA	Repository support for JPA applications
Apache Hadoop	DI support for map-reduce job, tool & HDFS APIs, and related project APIs, along with their use in Spring Batch. No repository support
GemFire	DI support for API, GemFireTemplate. Repository support
Redis	DI support for API, RedisTemplate. No repository support
MongoDB	DI support for API, MongoTemplate. Repository support
Neo4j	DI support for API, Neo4jTemplate. Repository support

# Spring & Big Data

- Spring Framework Overview
- Spring Data Overview
- **Spring Data GemFire**

# What is Spring Data for GemFire?

- A sub-project of Spring Data
- Why?
  - GemFire is a complex product, complex to configure
  - In many cases configuration is repetitive
- What
  - Provides simplified configuration for GemFire
  - Use namespace to define a cache, regions, replication ...

<http://projects.spring.io/spring-data-gemfire>



# Spring & Big Data

- Spring Framework Overview
- Spring Data Overview
- **Spring Data GemFire**
  - Client Cache Configuration
  - GemFire Template
  - Using Repositories and defining domain objects

# The GemFire Namespace

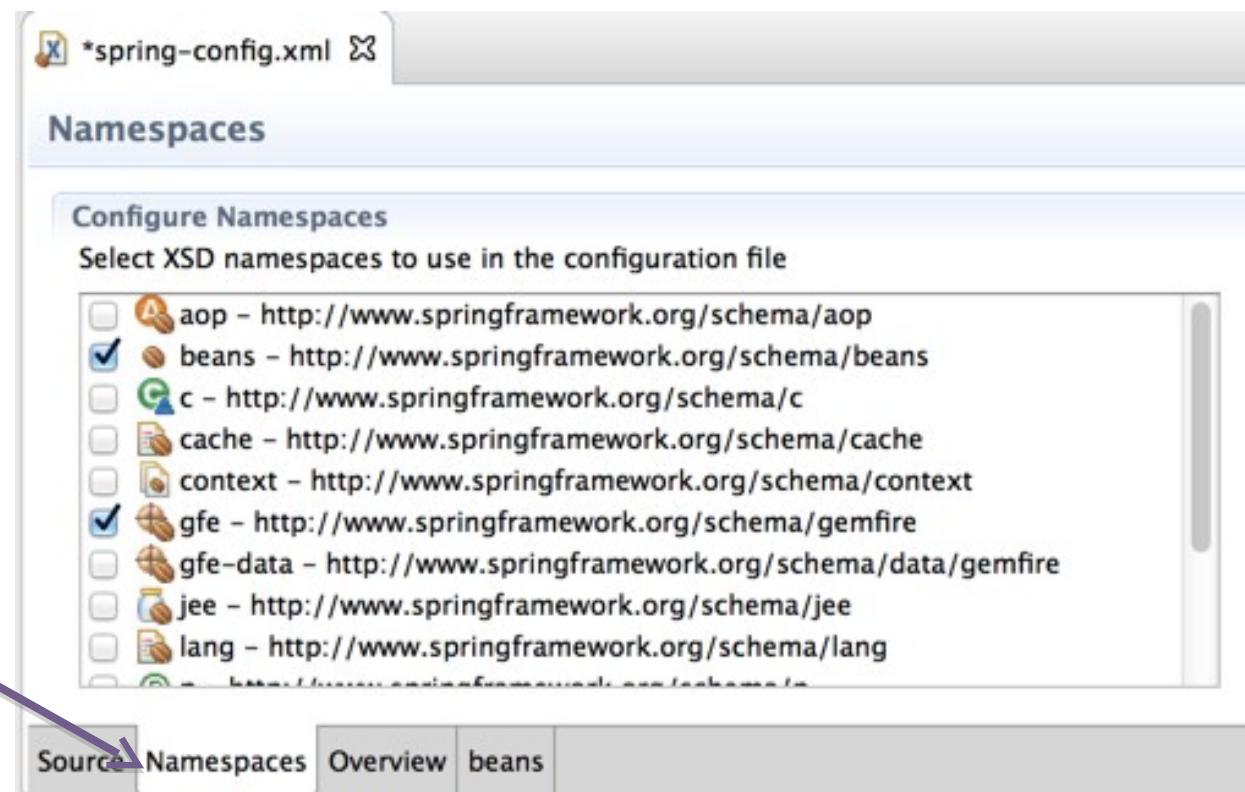
- Spring Data GemFire adds several namespaces to the standard set of namespaces found in core Spring

```
<beans ...  
    xmlns:gfe="http://www.springframework.org/schema/gemfire"  
    xsi:schemaLocation="...  
        http://www.springframework.org/schema/gemfire  
        http://www.springframework.org/schema/gemfire/spring-gemfire.xsd">  
  
    <gfe:client-cache />  
</beans>
```

Enables the Spring Data Gemfire namespace with the 'gfe' prefix

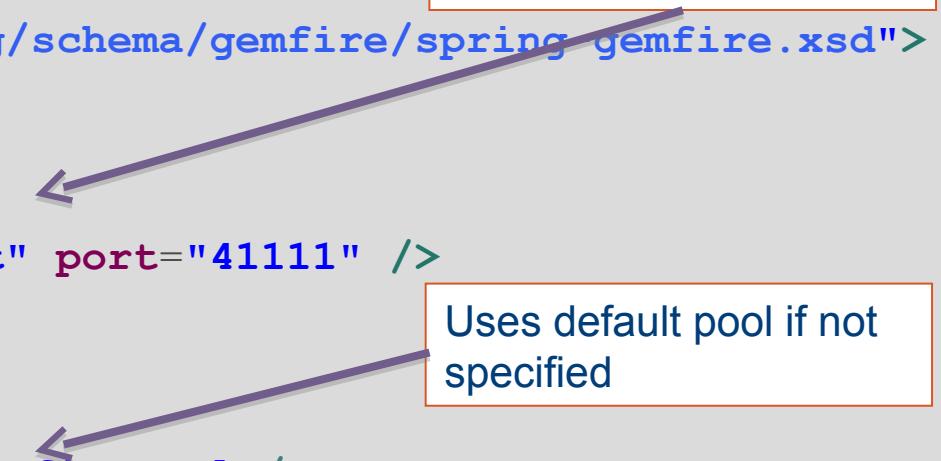
# Adding Namespaces Declarations

- Typing the appropriate namespace info is error-prone
  - Use the dedicated STS wizard



# Simplifying Common Configuration with Spring

```
<beans ...  
    xmlns:gfe="http://www.springframework.org/schema/gemfire"  
    xsi:schemaLocation="...  
        http://www.springframework.org/schema/gemfire  
        http://www.springframework.org/schema/gemfire/spring_gemfire.xsd">  
  
    <gfe:pool id="gemfirePool" >  
        <gfe:locator host="localhost" port="41111" />  
    </gfe:pool>  
  
    <gfe:client-cache pool-name="gemfirePool"/>  
  
    <gfe:client-region id="Orders" pool-name="gemfirePool"/>  
</beans>
```



# Using Java Configuration

- This part doesn't exactly leverage Spring Data GemFire

```
@Configuration  
public class SpringGemFireClientConfig {  
  
    @Bean  
    ClientCache cache() {  
  
        return new ClientCacheFactory()  
            .set("cache-xml-file", "xml/clientCache.xml").create();  
    }  
  
    @Bean  
    Region customerRegion(ClientCache cache) { ... }  
}
```

The code snippet illustrates Java Configuration for a Spring GemFire client. Annotations are highlighted in purple:

- `@Configuration`: An annotation indicating the class is a configuration source.
- `@Bean`: A declaration that the method returns a bean to be managed by the Spring container.
- `cache-xml-file`: A configuration property set via the `ClientCacheFactory`.

Annotations are annotated with callout boxes:

- A callout box labeled "Enables Java Config" points to the `@Configuration` annotation.
- A callout box labeled "Makes ClientCache a dependency" points to the `@Bean` declaration for `customerRegion`.

# A Simpler Way To Create a Client

- The gfe-data namespace adds several additional tags related to datasources and repositories for client cache

```
<beans ...  
    xmlns:gfe="http://www.springframework.org/schema/gemfire"  
    xmlns:gfe-data="http://www.springframework.org/schema/data/gemfire"  
    xsi:schemaLocation="... "  
  
    <gfe-data:datasource>      <  
        <gfe:locator host="localhost"  
            port="41111" />  
    </gfe-data:datasource>  
</beans>
```



Same as **gfe:pool** but:  
Automatically creates client regions for all regions on servers connected by locator(s)

# Configure Client using Spring

- Optionally define a pool and listener(s)

```
<!-- client region with pool and listener -->

<gfe:client-cache name="simple" pool-name="gemfire-pool"/>

<bean id="simpleListener" class="some.pkg.SimpleCacheListener" />

<gfe:pool id="c-listener" server-group="Portfolios"
           subscription-enabled="true" >
    <gfe:locator host="lucy" port="41111" />
</gfe:pool>

<gfe:client-region id="orders" pool-name="gemfire-pool" />
    <gfe:cache-listener ref="simpleListener">
</gfe:client-region>
```

# Spring Data GemFire: Continuous Query

- Defines a listener container
  - Similar to Spring JMS
  - Specify each query with its listener

```
<gfe:client-cache name="simple" pool-name="gemfire-pool"/>

<bean id="myListener" class="some.pkg.MyCQListener" />

<gfe:cq-listener-container>
    <gfe:listener ref="myListener"
        value="SELECT * FROM /Customers WHERE balance < 0"/>
    <gfe:listener ref="anotherListener" ... />
</gfe:cq-listener-container>
```

# Registering Interest

```
<gfe:client-cache name="simple" pool-name="gemfire-pool"/>

<gfe:client-region id="Customer" pool-name="gemfire-pool">
    <gfe:key-interest durable="true" result-policy="KEYS">
        <bean id="key" class="java.lang.String">
            <constructor-arg name="simple" />
        </bean>
    </gfe:key-interest>
</gfe:client-region>

...
```

# Dependency Injecting Regions

- Most GemFire components defined in Spring configuration can be injected in the standard way
  - Using the ref attribute in bean definition
  - Using **@Autowired**
- Regions are a little more challenging due to parameterization in region definitions
  - Use **@Resource(name="*regionName*")** instead

```
public class ConsumerDao {  
  
    @Resource(name="Customer")  
  
    private Region<Integer, Customer> customers;  
  
    public Customer getCust(int customerNo) {  
  
        return customers.get(customerNo);  
    }  
}
```

# Spring & Big Data

- Spring Framework Overview
- Spring Data Overview
- **Spring Data GemFire**
  - Client Cache Configuration
  - **GemFire Template**
  - Using Repositories and defining domain objects

# GemfireTemplate Overview

- GemfireTemplate provides many convenience functions to simplify interacting with region
  - Some mirror methods on GemFire Region object
  - Some additional convenience functions
  - Exception translation (checked => unchecked)

Method Category	Description
Constructors	Can be created with a region
CRUD methods	get, put, remove, replace, etc
Query support	query, find, findUnique
Additional helpers	containsKey, containsKeyOnServer, containsValue

# GemfireTemplate – 1

- Provides convenience methods

```
GemfireTemplate orders =
    new GemfireTemplate((Region) ctx.getBean("orderRegion"));

Collection results =
    (Collection) orders.query("status = 'delivered'");

System.out.println("Found " + results.size() + " orders.");
```

Only need to specify  
where clause

# GemfireTemplate – 2

- Also more explicit execution control if necessary

```
orders.execute(new GemfireCallback<Iterable<CustomerOrder>>() {  
  
    public Iterable<CustomerOrder> doInGemfire(Region region)  
        throws GemFireCheckedException, GemFireException {  
  
        region.put("ON000123", new CustomerOrder(...));  
  
        return region.query("totalCost > 100.0");  
    }  
});
```

```
orders.execute( region -> {  
  
    region.put("ON000123", new CustomerOrder(...));  
  
    return region.query("totalCost > 100.0");  
} );
```

Much cleaner  
with a *closure*

# Spring & Big Data

- Spring Framework Overview
- Spring Data Overview
- **Spring Data GemFire**
  - Client Cache Configuration
  - GemFire Template
  - **Using Repositories and defining domain objects**

# Using Repositories

- Configuring the domain class

```
@Region("Customer")  
public class Customer{ ... }
```

Assumes customer has an attribute accountNumber

- Writing the Repository

```
public interface CustomerRepository extends CrudRepository {  
    Customer findByAccountNumber(String acctNumber);  
  
    @Query("SELECT * from /Customer c where c.firstName = $1")  
    List<Customer> findByFirstName(String firstName);  
}
```

- Configuring Spring

```
<gfe-data:repositories id="custRepo" base-package="com.pivotal.demo" />
```

# Domain Classes and Identity Fields

- Typically, one field holds same value as the region key
  - Mark with @Id

```
@Region("Customer")  
public class Customer {  
    @Id Long id;  
    String firstname;  
    String lastname;  
  
    @PersistenceConstructor  
    public Customer (String firstname, String lastname) {  
        // ...  
    }  
}
```

Marks as Domain class and associates with 'customer' region

Flags this field to be used for key

Disambiguates multiple constructors

# Using Repositories

```
@Component  
public class CustomerRepositoryInvoker {  
    @Autowired ←  
    CustomerRepository customerRepo;  
  
    public List<Customer> customersByFirstName(String firstName) {  
        return customerRepo.findByFirstName(firstName);  
    }  
}
```

Repository Proxy implementation automatically injected

# Spring Gemfire supports Spring Transactions

- Uses the `GemfireTransactionManager`
  - Implements Spring's `PlatformTransactionManager`
  - Provides transactions, just as for any Spring application
  - Configure in Spring bean file
- Add `@Transactional` to class/method(s)
  - or use Aspects via beans XML

```
<gfe:client-cache id="cache" .../>
<gfe:transaction-manager id="transactionManager"
    cache-ref="cache">
<tx:annotation-driven/>
```

```
@Transactional
public void updateOrder(Order order) {
    orderRegion.put(order.getId(), order);
}
```

# Function Registration using Annotations

- Using annotations to define functions

```
@OnRegion(region="Customer", resultCollector="MyCustomCollector")  
public interface FunctionExecution {  
    @FunctionId("MyFunction")  
    String doIt (String s1, int s2);  
    @FunctionId("MyFunction2")  
    String getString(Object arg1, @Filter Set<Object> keys);  
}
```

- Enabling function annotations

```
<gfe-data:function-execution base-package="com.pivotal.functions" />
```

# Executing Function

```
@Component
public class MyFunctionInvoker {
    @Autowired
    FunctionExecution functionExecution;

    String doIt() {
        return functionExecution.doIt("hello", 123);
    }

    String doWithFilter(Set<String> keys) {
        return functionExecution.getString("hello", keys);
    }
}
```

# Conclusion

- Spring Data GemFire provides a number of simplifications
  - Client side – the focus of this presentation
  - Server side also
- Key simplifications
  - Cache configuration
  - Registering event listeners & CqListeners
  - Registering interest in events
  - GemfireTemplate
  - Common data access patterns using Repository interface
  - Simplified function execution

# Lab

In this lab, you will:

1. Configure a client cache using Spring GemFire
2. Use the GemfireTemplate
3. Configure an event listener
4. Register interest

# Pivotal

A NEW PLATFORM FOR A NEW ERA

# REST Support with GemFire

# Learner Objectives

After completing this lesson, you should be able to

- Understand basic REST support features in GemFire
- Enable REST support in GemFire
- Use the basic APIs to
  - Get an entry
  - List available regions
  - Perform a query
- Write a basic Java client to make RESTful requests

# Outline

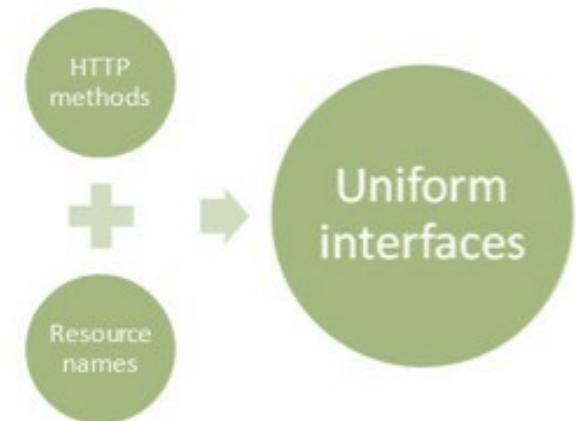
- GemFire REST Overview
- Setting up REST Support
- REST API Overview
- Using Spring RestTemplate

# REST Introduction

- Web apps not just usable by browser clients
  - Programmatic clients can also connect via HTTP
  - Such as: mobile applications, AJAX enabled web-pages
- REST is an *architectural style* that describes best practices to expose web services over HTTP
  - REpresentational State Transfer, term by Roy Fielding
  - HTTP as *application* protocol, not just transport
  - Emphasizes scalability
  - *Not a framework or specification*

# REST Principles (1)

- Expose *resources* through URIs
  - Model nouns, not verbs
  - <http://mybank.com/banking/accounts/123456789>
- Resources support limited set of operations
  - GET, PUT, POST, DELETE in case of HTTP
  - All have well-defined semantics
- Example: update an order
  - PUT to </orders/123>
  - don't POST to </order/edit?id=123>



# REST Principles (2)

- Stateless architecture
  - No HttpSession usage
  - GETs can be cached on URL
  - Requires clients to keep track of state
  - Part of what makes it scalable
  - Looser coupling between client and server
- HTTP headers and status codes communicate result to clients
  - All well-defined in HTTP Specification

# Why REST?

- Benefits of REST
  - Every platform/language supports HTTP
    - Unlike for example SOAP + WS-\* specs
  - Easy to support many different clients
    - Scripts, Browsers, Applications
  - Scalability
  - Support for redirect, caching, different representations, resource identification, ...
  - Support for XML, but also other formats
    - JSON and Atom are popular choices



# Basic REST Features in GemFire

- Data can be stored directly as JSON objects
  - Objects stored in PDX Serialized form (PdxInstance)
  - No class required for server
  - No class required for client – results returned as JSON data
- Other languages supported besides Java, C++ and C#
  - Ruby
  - JavaScript
  - Python
- Support for secure connections & authenticated access

# Prerequisites for Using REST in GemFire

- Server must be configured to use PDX Serialization and have '**read-serialized=true**' set
- REST interface runs as an embedded HTTP or HTTPS server within one or more members
  - Requests will go directly to the member having the REST service enabled
  - All functions & referenced classes must be on the target member's classpath

# REST Considerations for GemFire

- Only supported exchange format is **application/json** mime type
- \*\*Keys are always treated as strings\*\*
- Functions invoked via REST interface must return PdxInstance or object that can be written as JSON
- Single-hop not supported for Partitioned region access
  - Requests will always go to HTTP service endpoint
  - Host member will make the extra hop to member with specified entry
- No support for sub-regions

# Outline

- GemFire REST Overview
- **Setting up REST Support**
- REST API Overview
- Using Spring RestTemplate

# Starting a Server with REST Enabled

- Enable PDX serialization and `read-serialized=true`
  - Needs to be done on all servers hosting region accessed via REST
  - If done via gfsh, it should be done before starting servers

```
gfsh> start locator --name=locator1 --port=411111  
  
gfsh> configure pdx --read-serialized=true
```

- Start server with REST API enabled
  - Optionally, set the HTTP port (7070 is the default)

```
gfsh> start server --name=locator1 --server-port=40404  
--J=-Dgemfire.start-dev-rest-api=true  
--J=-Dgemfire.http-service-port=7071
```

# Additional REST Service Configuration

- Properties can be placed into a local properties file as well

```
start-dev-rest-api=true  
http-service-port=7071
```

gemfire.properties

- You can also set SSL properties

- Set in gemfire.properties or in gfsecurity.properties (recommended)
  - Same basic set of properties as for cluster security

```
http-service-ssl-enabled=true  
http-service-ssl-require-authentication=true  
http-service-ssl-keystore=/path/to/trusted.keystore
```

gfsecurity.properties

# Testing the Service

- Verify basic response – expect a list of deployed regions

```
$ curl -i http://localhost:7071/gemfire-api/v1
```

- Verify the logs

```
[info 2015/12/17 13:52:51.626 IST server1 <main> tid=0x1] Started  
o.e.j.w.WebAppContext@4c447c09{/gemfire-api,file:/Applications/GemFire-  
Developer-8.2.a.RELEASE/GemFire-Developer-8.2.a.RELEASE/mark_demo/server1/GemFire_msecrist/  
services/http/0.0.0.0_7071_gemfire-api/webapp/,AVAILABLE}{/Applications/GemFire-  
Developer-8.2.a.RELEASE/Pivotal_GemFire_820_b17919_Linux/tools/Extensions/gemfire-api.war}
```

```
[info 2015/12/17 13:52:51.631 IST server1 <main> tid=0x1] Started  
ServerConnector@120d3fd{HTTP/1.1}{0.0.0:7071}
```

```
[info 2015/12/17 13:52:51.631 IST server1 <main> tid=0x1] Started @5718ms
```

```
[info 2015/12/17 13:52:51.632 IST server1 <main> tid=0x1] HTTP service started  
successfully...!!
```

```
[info 2015/12/17 13:52:51.632 IST server1 <main> tid=0x1] Initializing region  
__ParameterizedQueries__
```

```
[info 2015/12/17 13:52:51.633 IST server1 <main> tid=0x1] Initialization of region  
__ParameterizedQueries__ completed
```

# Swagger UI Overview

- Swagger: An open source framework for standardized APIs via REST
- Swagger UI is a web app that visualizes that interface

The screenshot shows a web browser displaying the Pivotal GemFire Developer REST API documentation at [localhost:7071/gemfire-api/docs/index.html](http://localhost:7071/gemfire-api/docs/index.html). The page has a dark teal header bar with the title "Pivotal™ GemFire® Developer REST API". Below the header, there's a sub-header: "Developer REST API and interface to GemFire's distributed, in-memory data grid and cache." Underneath, there are links for "Terms of service", "Contact the developer", and "Pivotal GemFire Documentation". The main content area lists three categories: "functions : functions", "queries : queries", and "region : region", each with "Show/Hide", "List Operations", "Expand Operations", and "Raw" buttons. At the bottom, it says "[ BASE URL: <http://localhost:7071/gemfire-api/api-docs> , API VERSION: 1 ]".

# Swagger UI: Interacting With The Service

functions : functions

Show/Hide | List Operations | Expand Operations | Raw

queries : queries

Show/Hide | List Operations | Expand Operations | Raw

PUT	/v1/queries/{query}	update parameterized query
DELETE	/v1/queries/{query}	delete parameterized query
POST	/v1/queries	create a parameterized Query
GET	/v1/queries	list all parameterized queries
GET	/v1/queries/adhoc	run an adhoc query

Implementation Notes  
Run an unnamed (unidentified), ad-hoc query passed as a URL parameter

Parameters

Parameter	Value	Description	Parameter Type	Data Type
q	(required)	oql	query	string

Response Messages

HTTP Status Code	Reason	Response Model
200	OK.	Interactive interface (not just read-only)
500	GemFire throws an error or exception	

[Try it out!](#)

POST /v1/queries/{query} run parameterized query

The image shows several annotations with red arrows pointing to specific UI elements:

- An arrow points from the "Example URI specifications" callout box to the "/v1/queries" endpoint entry.
- An arrow points from the "Response Model" callout box to the "Interactive interface (not just read-only)" message under the 200 OK response.
- An arrow points from the "Try it out!" button to the "Try it out!" button at the bottom left of the main content area.

# Outline

- GemFire REST Overview
- Setting up REST Support
- **REST API Overview**
- Using Spring RestTemplate

# Understanding the API

- All invocations have a basic URI pattern  
ex [http://host:port/gemfire-api/v1/target/...](http://host:port/gemfire-api/v1/target/)
  - Where ‘target’ is one of region name, ‘query’ or ‘functions’
- Region operation to list entries
  - <http://localhost:7071/gemfire-api/v1/Customer>
- Query operation
  - <http://localhost:7071/gemfire-api/v1/queries/selectOrders>
  - ‘selectOrders’ is a pre-defined query
- Function operation
  - <http://localhost:7071/gemfire-api/v1/functions/genericSum>
  - ‘genericSum’ is a pre-defined function

# Outline

- GemFire REST Overview
- Setting up REST Support
- REST API Overview
- **Using Spring RestTemplate**

# RestTemplate

- Provides access to RESTful services
  - Supports URI templates, HttpMessageConverters and custom execute() with callbacks
  - Map or String... for vars, java.net.URI or String for URL

HTTP Method	RestTemplate Method
DELETE	delete(String url, String... urlVariables)
GET	getForObject(String url, Class<T> responseType, String... urlVariables)
HEAD	headForHeaders(String url, String... urlVariables)
OPTIONS	optionsForAllow(String url, String... urlVariables)
POST	postForLocation(String url, Object request, String... urlVariables)
	postForObject(String url, Object request, Class<T> responseType, String... urlVariables)
PUT	put(String url, Object request, String... urlVariables)

# Defining a RestTemplate

- Just call constructor in your code

```
RestTemplate template = new RestTemplate();
```

- Has default HttpMessageConverters
  - Same as on the server, depending on classpath
- Or use external configuration
  - To use Apache Commons HTTP Client, for example

```
<bean id="restTemplate" class="org.sfw.web.client.RestTemplate">
  <property name="requestFactory">
    <bean class="org.sfw.http.client.CommonsClientHttpRequestFactory"/>
  </property>
</bean>
```

# RestTemplate Usage Examples

```
RestTemplate template = new RestTemplate();
String uri = "http://localhost:7071/gemfire-api/v1/BookOrder/{id}";
```

{id} = 1

```
// GET all order items for an existing order with ID 1:
```

```
BookOrder bookOrder= template.getForObject(uri, BookOrder.class, "1");
```

```
// POST to create a new item
```

```
BookOrderItem item = // create item object
```

```
URI itemLocation = template.postForLocation(uri, item, "1");
```

```
// PUT to update the item
```

```
item.setAmount(2);
```

```
template.put(itemLocation, item);
```

```
// DELETE to remove that item again
```

```
template.delete(itemLocation);
```



*Also supports `HttpEntity`, which makes adding headers to the HTTP request very easy as well*

# Handling formatting of JSON

- Some types (ex Date) can be challenging with JSON
- Jackson JSON library provides rich set of tools for marshalling JSON elements to/from Java types
  - Annotations (example: `@JsonFormat`)

```
public class BookOrder {  
    private Integer orderNumber;  
  
    @JsonFormat(shape=JsonFormat.Shape.STRING, pattern="MM/dd/yyyy")  
    private Date orderDate;  
    ...  
}
```

- JsonSerializer – write your own custom serialization/deserialization

# Lab

In this lab, you will

- Create configure REST support in the server
- Verify correct configuration with the Swagger UI
- Write a basic client application using RestTemplate

# Pivotal

A NEW PLATFORM FOR A NEW ERA

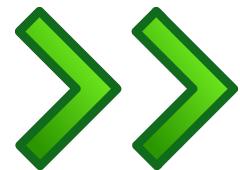
# Finishing Up

Course Completed

What's Next?

# What's Next

- Congratulations, you've finished this course
- What do do next?
  - Certification
  - Other courses
  - Resources
  - Evaluation
- Check out additional sections on (optional) ...



# Certification



- Computer-based exam
  - 50 multiple-choice questions
  - 90 minutes
  - Passing score: 76% (38 questions answered successfully)
- Preparation
  - Review all the slides
  - Redo the labs
  - Dig deeper with documentation

# Certification: Questions

## Typical question

- Statements
  - a. GemFire is an In Memory Distributed Database
  - b. GemFire offers High Latency processing
  - c. A GemFire cache is a container for one or more regions
- Pick the correct response
  - 1. Only a. is correct
  - 2. Both a. and c. are correct
  - 3. All are correct
  - 4. None are correct

# Certification: Logistics

- Where?
  - At any Pearson VUE Test Center
  - Most large or medium-sized cities
    - See <http://www.pearsonvue.com/vtclocator>
- How?
  - At the end of the class, you will receive a certification voucher by email
  - Make an appointment
  - Give them the voucher when you take the test
- For any further inquiry, you can write to
  - [education@pivotal.io](mailto:education@pivotal.io)

# Other courses



- Many courses available
  - Processing Big Data with Pivotal HD
  - GemFire Administration
  - HAWQ Architecture and Implementation
  - Data Science in Practice
- More details here:
  - <http://www.pivotal.io/training>

# Processing Big Data with Pivotal HD

- 4-day workshop
- Understand how to process Big Data with Pivotal HD and the BDS Suite
  - Hadoop HDFS and Map/Reduce
  - Hive
  - Streaming and Pig
  - HAWQ
- Pivotal Certified Big Data Architect (for example)

# Pivotal Support Offerings

- Global organization provides 24x7 support
  - How to Register: <http://tinyurl.com/piv-support>
- Premium and Developer support offerings:
  - <http://www.pivotal.io/support/offering>
  - <http://www.pivotal.io/support/oss>
  - Both Pivotal App Suite *and* Open Source products
- Support Portal: <https://support.pivotal.io>
  - Community forums, Knowledge Base, Product documents



# Pivotal Consulting

- Custom consulting engagement?
  - Contact us to arrange it
    - <http://www.pivotal.io/contact/spring-support>
    - Even if you don't have a support contract!
- Pivotal Labs
  - Agile development experts
  - Assist with design, development and product management
    - <http://www.pivotal.io/agile>
    - <http://pivotallabs.com>



Pivotal™

# Resources

- The GemFire reference documentation
  - <http://gemfire.docs.pivotal.io>
  - Already 1000+ pages!
- The GemFire Knowledge Base
  - <https://support.pivotal.io/hc/en-us/categories/200072748-P>
- GemFire Forums
  - <https://support.pivotal.io/hc/communities/public/topics/200072748-P>

# Thank You!

We hope you enjoyed the course

Please fill out the evaluation form

<http://tinyurl.com/PvtlAmerEval>



# Thank You.

# Pivotal

A NEW PLATFORM FOR A NEW ERA