Association and Collection Mapping

Originals of Slides and Source Code for Examples:
http://courses.coreservlets.com/Course-Materials/hibernate.html

Customized Java EE Training: http://courses.coreservlets.com/
Servlets, JSP, Struts, JSF/MyFaces/Facelets, Ajax, GWT, Spring, Hibernate/JPA, Java 5 & 6.
Developed and taught by well-known author and developer. At public venues or onsite at your location.

For live Spring & Hibernate training, see courses at http://courses.coreservlets.com/.

Taught by the experts that brought you this tutorial.
Available at public venues, or customized versions can be held on-site at your organization.

- Courses developed and taught by Marty Hall
  - Java 5, Java 6, intermediate/beginning servlets/JSP, advanced servlets/JSP, Struts, JSF, Ajax, GWT, custom mix of topics
- Courses developed and taught by coreservlets.com experts (edited by Marty)
  - Spring, Hibernate/JPA, EJB3, Ruby/Rails

Contact hall@coreservlets.com for details
Topics in This Section

• Understand Collection and Association relationships
• See how to realize relationships in Java and databases
• Walk through the Hibernate approach of mapping both Collections and Associations.
Relationship Types

• **Association**
  – Mapping relationships between two objects
  – Example
    • Account and AccountOwner

• **Collection**
  – Collection of values representing individual pieces of data
  – Example
    • Map of holidays
    • String array of months

Relationship Dimensions

• Relationships between entities can exist in multiple ways
  – Multiplicity
    • How many on each side of the relationship?
  – Directionality
    • From which side(s) of the relationship can you access the other?

• A single object may have multiple relationships
Relationship Multiplicity

- **One-to-Many**
  - A *single* Account has *many* Transactions
  - Reverse of a many-to-one relationship

- **Many-to-One**
  - *Multiple* Transactions belong to a *single* account
  - Reverse of a one-to-many relationship

- **One-to-One**
  - A *single* AccountOwner has a *single* HomeAddress
  - A *single* HomeAddress has a *single* AccountOwner

- **Many-to-Many**
  - *Multiple* Accounts have *multiple* AccountOwners
  - Often realized through two one-to-many relationships
    - A *single* Account has *multiple* AccountOwners
    - A *single* AccountOwner has *multiple* Accounts

Relationship Directionality

- **Unidirectional**
  - Can only traverse objects from one side of the relationship
  - Example: Account : Transaction
    - Given an Account object, can obtain related Transaction objects.
    - Given a Transaction object, *cannot* obtain related Account object.

- **Bidirectional**
  - Can traverse objects from both sides of the relationship
  - Example: Account : Transaction
    - Given an Account object, can obtain related Transaction objects.
    - Given a Transaction object, *can* obtain related Account object.
Realizing Relationships

Customized Java EE Training: http://courses.coreservlets.com/
Servlets, JSP, Struts, JSF/MyFaces/Facelets, Ajax, GWT, Spring, Hibernate/JPA, Java 5 & 6.
Developed and taught by well-known author and developer. At public venues or onsite at your location.

Java vs. Database

- **Java**
  - Objects are inherently directional
    - An object has a reference/pointer to another object
  - Transition by walking a networked graph of object references

- **Database**
  - Relations are *not* inherently directional
    - A table can arbitrarily join its columns with columns of other tables (not just those keyed to)
  - Transition by joining tables together through joins/foreign keys

*Source: Java Persistence with Hibernate*
Relationships in Java

- Object has an attribute referencing another related object
- For ‘Many’ side, Collections API
  - Set
    - No duplication allowed
    - Objects organized with or without order
  - Map
    - Duplicate values allowed, using different keys
    - Can be organized with or without order
  - List
    - Duplication allowed
    - Objects expected to be organized in an order
  - Arrays
    - Duplication allowed
    - Objects expected to be organized in an order
    - Strongly typed to particular object type, and lacks ability to resize

Relationships in Database

- Record relationships can be realized using several techniques
  - Denormalized table
    - Record repeated in same table, each time capturing different relationship data.
  - Foreign keys
    - Follow identifiers to related records on other tables
  - Join tables
    - Tables specifically setup to maintain a relationship between two identities (usually for M:M)
  - Ad hoc joins in a query
    - Arbitrary joins between columns relating data
- Each technique has its pros/cons
**Relationships in Database**

- **Denormalized Table**
  - **Pros**
    - Very fast
    - Easy to query against
  - **Cons**
    - Contains redundant data
    - Requires many nullable columns

- **Foreign Keys**
  - **Pros**
    - Reduce redundancy
    - Better modeling of data
  - **Cons**
    - Slower than denormalized table
    - Slightly more complicated to query against
Relationships in Database

• Join Tables
  – Pros
    • Built on foreign key model, enables many:many relationships
  – Cons
    • Slower yet, and even more complex querying

<table>
<thead>
<tr>
<th>CUSTOMER</th>
<th>CONTRACTOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSTOMER_ID</td>
<td>CONTRACTOR_ID</td>
</tr>
<tr>
<td>CUSTOMER_NAME</td>
<td>CONTRACTOR_NAME</td>
</tr>
<tr>
<td>CUSTOMER_PHONE</td>
<td>CONTRACTOR_PHONE</td>
</tr>
<tr>
<td>CUSTOMER_ADDRESS</td>
<td>CONTRACTOR_ADDRESS</td>
</tr>
<tr>
<td>CUSTOMER_CELL</td>
<td>CONTRACTOR_CELL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CONTRACT</th>
<th>CUSTOMER</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONTRACT_ID</td>
<td>CUSTOMER_ID</td>
</tr>
<tr>
<td>CONTRACTOR_ID</td>
<td>CUSTOMER_NAME</td>
</tr>
<tr>
<td>PRICE</td>
<td>CUSTOMER_ADDRESS</td>
</tr>
<tr>
<td>START_DATE</td>
<td>CUSTOMER_PHONE</td>
</tr>
</tbody>
</table>

• Joins
  – Pros
    • Allows for any possible query a user can think of without having to predefine the requirements in the schema design
  – Cons
    • No model enforcement
      – Can join ‘age’ and ‘office floor’ columns – but does it make sense?
    • Can be complicated/confusing to write; results may not appear as desired

SELECT CONTRACTOR_NAME FROM CONTRACTOR WHERE CONTRACTOR_STATE = (SELECT CUSTOMER_STATE FROM CUSTOMER WHERE CUSTOMER_NAME='JOHN SMITH');
Realizing Relationships with Hibernate

Domain object graph models

- Hibernate represents domain object relationships through a graph model.

- Bidirectional relationships require graph model updates for objects on both sides
  - Need to keep all in-memory objects up-to-date.
  - If Objects A and B have a bidirectional relationship, and Object B is added to Object A, need to make sure that Object A also gets added to Object B
  - Can cause complication when persisting the relationship
    - Which side should save the relationship?
Marking the Saving Side

“inverse” attribute
- Used for both 1:M/M:1 and M:M relationships
- For 1:M/M:1
  - Usually placed in the mapping file of the single object side of the relationship
- For M:M
  - One side of the relationship
  - If <idbag> used, must be on the non-idbag side

- “property-ref” attribute
  - Used for 1:1 relationships
  - Either side of the relationship, but only once

Setting up Relationships

1. Determine your domain model relationships
   - Define each as an association or collection
   - Identify the multiplicity between objects
   - Decide on the directionality
     - Does it need to be bidirectional?

2. Define your relationship implementation types and add the Interface representation of each association/collection to appropriate domain objects (List, Set, Array, etc...)

3. Create the mappings in the corresponding object mapping files

4. If bidirectional, optionally add bidirectional maintenance code
Collections in Domain Objects

• **Must always use Interfaces**
  – Collection, List, Map, Set, etc…

• **Should be instantiated right away**
  – Not delegated to Constructor, Setter Method, etc…

```java
List mylist = new ArrayList()
```

Mapping Multiplicity

• **1:M/M:1**
  – Example: EBiller has many EBills / EBill has one EBiller
  – On the ‘one’ side, map the Collection objects (EBiller)
    • `<one-to-many class="courses.hibernate.vo.EBill"/>
  – On the ‘many’ side, map the single object (EBill)
    • `<many-to-one name="ebiller" column="EBILLER_ID"
      class="courses.hibernate.vo.EBiller" />`

• **M:M**
  – Example: Account has many EBillers / EBiller has many Accounts
  – On both sides, map the Collection objects
    • `<many-to-many column="ACCOUNT_ID"
      class="courses.hibernate.vo.Account" />`
    • `<many-to-many column="EBILLER_ID"
      class="courses.hibernate.vo.EBiller" />`

• Only required on both sides if the relationship is bidirectional
Mapping Multiplicity 1:1

- **UNINTUITIVE MAPPING!**
- **Example:**
  - EBill has at most one AccountTransaction
  - AccountTransaction has at most one EBill
  - Foreign key exists on the EBill table (to AccountTransaction)
- **In the NON-STORING Entity mapping file** *(AccountTransaction)*
  - `<one-to-one name="ebill"`
    class="courses.hibernate.vo.EBill"
    property-ref="accountTransaction"/>

- **In the STORING Entity mapping file** *(EBill)*
  - `<many-to-one name="accountTransaction"
    column="ACCOUNT_TRANSACTION_ID"
    class="courses.hibernate.vo.AccountTransaction"/>

- Enforce one-to-one in the database with unique constraints
- If storing foreign keys on both sides, use two many-to-one tags.

Java-to-Database Through Hibernate

- **Association & Collection mapping tags practically identical**
- **Hibernate Collection Types**
  - `<set>`
    - Unordered/Ordered, requiring value column
  - `<map>`
    - Unordered/Ordered, requiring key and value columns
  - `<list>`
    - Ordered, requiring an index column on the referenced object table
Java-to-Database Through Hibernate

- `<array>`
  - Map to Java Type and Primitive Arrays
  - Ordered, requiring an index column on the referenced object table

- `<bag>`
  - No direct implementation available in Java
  - Unordered/ordered collection allowing duplicates
  - Realized through Collection/List
  - Requires value column

- `<idbag>`
  - Used for many-to-many relationships
  - Same as Bag, but with additional identifier column used for surrogate keys
  - Requires an ID Generator just like Entity classes

Association as `<set>`

- Maps to a ‘Set’ interface
  - ImplS include HashSet, TreeSet, etc…
- Can be optionally sorted

  EBill has many EBills (1:M / M:1)

**EBill Mapping**

```xml
<many-to-one name="ebiller" column="EBILLER_ID"
class="courses.hibernate.vo.EBiller" />
```

**EBiller Mapping**

```xml
<set name="ebills" inverse="true"
sort="unsorted|natural|my.custom.MyComparator">
  <key column="EBILLER_ID" not-null="true"/>
  <one-to-many class="courses.hibernate.vo.EBill"/>
</set>
```
Association as <map>

- Maps to a ‘Map’ interface
  - Impls include HashMap, TreeMap, etc…
- Must identify a column to be used for map key
  - Can be ‘column’, or ‘formula’ (any sql expression)
- Can be optionally sorted

```xml
<many-to-one name="ebiller" column="EBILLER_ID"
class="courses.hibernate.vo.EBiller" />
```

EBBill Mapping

```xml
<mapping name="ebills" inverse="true"
sort="unsorted|natural|my.custom.MyComparator">
  <key column="EBILLER_ID"/>
  <map-key column="EBILLID" type="long"/>
  <one-to-many class="courses.hibernate.vo.EBill"/>
</mapping>
```

Association as <list>

- Maps to a ‘List’ interface
  - Impls include ArrayList, LinkedList, etc…
- MUST have a dedicated list-index column on table
  - Sequential ordering of items for the parent specified
  - Skipped numbers result in Null values in list

```xml
<many-to-one name="ebiller" column="EBILLER_ID"
class="courses.hibernate.vo.EBiller" />
```

EBBill Mapping

```xml
<mapping name="ebills" inverse="true">
  <key column="EBILLER_ID" not-null="true"/>
  <list-index column="EBILLER_EBILL_NUMBER"/>
  <one-to-many class="courses.hibernate.vo.EBill"/>
</mapping>
```
Association as <array>

- Like ‘List’, Arrays **MUST** have a dedicated list-index column on table

  EBiller has many EBills (1:M / M:1)

**EBill Mapping**
<many-to-one name="ebiller" column="EBILLER_ID"
  class="courses.hibernate.vo.EBiller" />

**EBiller Mapping**
<array name="ebillsArray" inverse="true">
  <key column="EBILLER_ID"/>
  <list-index column="EBILLER_EBILL_NUMBER"/>
  <one-to-many class="courses.hibernate.vo.EBill"/>
</array>

Association as <bag>

- **Must be mapped to a Collection or List Interface**
  - ‘List’ can be used in combination with ‘order-by’ to preserve order
- **Can be optionally sorted**

  EBiller has many EBills (1:M / M:1)

**EBill Mapping**
<many-to-one name="ebiller" column="EBILLER_ID"
  class="courses.hibernate.vo.EBiller" />

**EBiller Mapping**
<bag name="ebills" inverse="true" order-by="DUE_DATE ASC">
  <key column="EBILLER_ID" not-null="true"/>
  <one-to-many class="courses.hibernate.vo.EBill"/>
</bag>
**Association as <idbag>**

- **Same as ‘Bag’** – but only used in many-to-many relationships
  - Allows for mapping of surrogate keys on join table
  - Hibernate will set the id on the join table
  - Can **NOT** use idbag on both sides of the relationship! Non-idbag side must have inverse="true"

- **Can be optionally sorted**

  EBill has many Accounts / Account has many EBillers (M:M)

**EBill Mapping**

```xml
<bag name="accounts" table="ACCOUNT_EBILLER" inverse="true">
  <key column="EBILLER_ID"/>
  <many-to-many column="ACCOUNT_ID"
       class="courses.hibernate.vo.Account"/>
</bag>
```

**Account Mapping**

```xml
<idbag name="accounts" table="ACCOUNT_EBILLER" order-by="DUE_DATE ASC">
  <collection-id column="ACCOUNT_EBILLER_ID" type="long">
    <generator class="native"/>
  </collection-id>
  <key column="EBILLER_ID"/>
  <many-to-many column="ACCOUNT_ID"
       class="courses.hibernate.vo.AccountOwner"/>
</idbag>
```

---

**Collection Mapping**

- **Can be used across all mapping types**
  - Just substitute relationship tag (<one-to-many, <many-to-one> etc...) with <element> tag
  - No inverse required (not an ‘association’)

**Previously Shown Association EBiller Mapping** (EBiller:EBill)

```xml
<bag name="ebills" inverse="true" order-by="DUE_DATE">
  <key column="EBILLER_ID" not-null="true"/>
  <one-to-many class="courses.hibernate.vo.EBill"/>
</bag>
```

**Collection EBiller Mapping** (stores balances across issued EBills)

```xml
<bag name="ebillBalances" table="EBILL" order-by="DUE_DATE">
  <key column="EBILLER_ID"/>
  <element column="BALANCE" type="double"/>
</bag>
```
Bidirectional Maintenance

- Developers must maintain bidirectional associations in order to keep in-memory objects up-to-date

```java
aParent.getChildren().add(aChild);
aChild.setParent(aParent);
```

- Hibernate recommends a strategy to ensure this process.

Hibernate’s Bidirectional Strategy

- Maintain associations on a single side of the relationship
- Make ‘setMySet()’ protected
- Create ‘addObject’ method instead of `object.getMySet().add(object);`
  - Within addObject(), set both relationships
- p.120 of Java Persistence with Hibernate
  Example EBiller→EBill (1:M/M:1)

```java
EBill (M:1)
protected void setEBiller(EBiller ebiller) {
    this.ebiller = ebiller;
}

EBiller (1:M)
public void addEBill(EBill ebill) {
    // Error in original code
    ebill.setEBiller(this);
    this.ebills.add(ebill);
}
```
**Modified Hibernate Strategy**

- Hibernate strategy might not solve all cases
  - Related objects are required to be in the same package
  - Developers need to remember which object to call
- **Modified Hibernate Strategy**
  - Maintain relationship on either side
  - Slightly varying implementations required for different relationship types
    - 1:1
    - M:M
      - Collections on both sides
    - 1:M/M:1
      - has single object that can potentially be null
- Collections should always be instantiated (early) – so never null

**Modified Bidirectional Strategy 1:1**

- Need to handle potential null object on each side
  - Objects might not be initialized

Example: EBill has one Transaction / Transaction has one EBill

**EBill Set Method**
```java
public void setTransaction(Transaction transaction) {
    this.transaction = transaction;
    if (transaction != null &&
        (transaction.getEbill() == null ||
        !transaction.getEbill().equals(this))) {
        transaction.setEbill(this);
    }
}
```

**Transaction Set Method**
```java
public void setEbill(EBill ebill) {
    this.ebill = ebill;
    if (ebill != null &&
        (ebill.getTransaction() == null ||
        !ebill.getTransaction().equals(this))) {
        ebill.setTransaction(this);
    }
}
```
Modified Bidirectional Strategy M:M

- Protected Setters for Collections on both sides
- Do not need to handle null checks
  - Collections should be initialized early

Example: Account has many EBillers / EBiller has many Accounts

**EBiller Set Method**
```java
protected void setAccounts(List<Account> accounts) {
    this.accounts = accounts;
}
```

**EBiller Add Method**
```java
public void addAccount(Account account) {
    this.accounts.add(account);
    if (!account.getEbillers().contains(this)) {
        account.addEbiller(this);
    }
}
```

**EBiller Remove Method**
```java
public void removeAccount(Account account) {
    this.accounts.remove(account);
    if (account.getEbillers().contains(this)) {
        account.removeEbiller(this);
    }
}
```

**Account Setter**
```java
protected void setEbillers(List<EBiller> ebillers) {
    this.ebillers = ebillers;
}
```

**Account Add Method**
```java
public void addEbiller(EBiller ebiller) {
    this.ebillers.add(ebiller);
    if (!ebiller.getAccounts().contains(this)) {
        ebiller.addAccount(this);
    }
}
```

**Account Remove Method**
```java
public void removeEbiller(EBiller ebiller) {
    this.ebillers.remove(ebiller);
    if (ebiller.getAccounts().contains(this)) {
        ebiller.removeAccount(this);
    }
```
Modified Bidirectional Strategy 1:M

- Protected Setter for Collection
- Need to handle null check on non-Collection side
  - Object may not have been initialized

  Example: EBiller has many EBills / EBill has one EBiller

```java
EBill Setter
public void setEbiller(EBiller ebiller) {
    this.ebiller = ebiller;
    if (ebiller != null && !ebiller.getEbills().contains(this)) {
        ebiller.addEbill(this);
    }
}
```

```java
EBiller Set Method
protected void setEbills (SortedSet<EBill> ebills) {
    this.ebills = ebills;
}
```

```java
EBiller Add Method
public void addEbill(EBill ebill) {
    this.ebills.add(ebill);
    if (!ebill.getEBiller().equals(this)) {
        ebill.getEBiller().getEBills().remove(ebill);
    }
}
```

```java
EBiller Remove Method
public void removeEbill(EBill ebill) {
    ebills.remove(ebill);
    if (ebill.getEBiller().equals(this)) {
        ebill.setEbiller(null);
    }
}
```
Bidirectional Concern

• **Recursive Issue**
  – Objects commonly refer to attributes contained within themselves during method execution
  – Can result in StackOverflowException
    • `hashCode`
    • `toString`
    • `equals`
    • Hibernate’s strategy for setting bidirectionality
  – Hibernate recommends NOT using associate objects in these methods
  – For 1:M, on the many side, set ‘access by field’ on the `<many-to-one>` tag.

• **In all, bidirectionality can be powerful, but complicated and overly involved to handle**
  – Ask yourself, does this really need to be bidirectional?
Many-to-Many Option

- Hibernate actually recommends *not* using traditional many-to-many relationships
- Proposes use of “Intermediate Associations”
  - Relationships often have data directly tied to them. If not, likely to later, so might as well start with this approach
  - “Intermediate” object to map the relationship
  - Can be accomplished in one of two ways
    - Composite element
    - Entity class

Intermediate Association: Composite

- Customer:Contractor (M:M)
- Parent object contains a Collection of “Contract” objects
  - Each “Contract” object contains the Customer, Contractor, and data about the relationship *(start date, contract price, etc...)*

Mapping for Customer
```
<set name="contracts" table="CUSTOMER_CONTRACTOR">
  <key column="customer_id"/>
  <composite-element class="Contract">
    <parent name="customer"/>
    <many-to-one name="contractor" column="contractor_id" not-null="true" class="Contractor"/>
    <property name="startDate" column="start_date" type="date"/>
    <property name="price" column="price" type="string"/>
  </composite-element/>
</set>
```
Intermediate Association: Composite

- **Advantages**
  - Lifecycle of composite element is tightly coupled to parent object
    - To create an association, add a new Contract to the Collection
      ```java
      Contract aContract = new Contract(aCustomer, aContractor);
      aCustomer.getContracts().add(aContract);
      ```
    - To delete, remove from Collection
      ```java
      aCustomer.getContracts().remove(aContract);
      ```
  
- **Disadvantages**
  - Bidirectional navigation is impossible
    - Composite element only exists within the context of the parent class
    - However – can write a query to retrieve the objects you need

Intermediate Association: Entity Class

- **Customer:Contractor (M:M)**
  - Realized through two one-to-many relationships
- **Each side contains a Collection of Contract objects**
  - Each Contract object contains the Customer, Contractor, and data about the relationship (*start date, contract price, etc...*)

**MAPPING FOR CUSTOMER**
```xml
<set name="contracts" inverse="true">
  <key column="customer_id" />
  <one-to-many class="Contract" />
</set>
```

**MAPPING FOR CONTRACTOR**
```xml
<set name="contracts" inverse="true">
  <key column="constractor_id" />
  <one-to-many class="Contract" />
</set>
```
Intermediate Association: Entity Class

MAPPING FOR CONTRACT (Relationship Entity)

```xml
<class name="Contract" table="CUSTOMER_CONTRACTOR"
     mutable="false">
    <id name="contractId" column="CUSTOMER_CONTRACTOR_ID">
        <generator class="native"/>
    </id>
    <property name="startDate" column="start_date" type="date"/>
    <property name="price" column="price" type="string"/>
    <many-to-one name="customer" column="customer_id" not-null="true" update="false"/>
    <many-to-one name="contractor" column="contractor_id" not-null="true" update="false"/>
</class>
```

**NOTICE CONTRACT HAS ITS OWN ID – (Think “Entity”)**

Intermediate Association: Entity Class

**Advantages**
- Bidirectional capabilities
  - Both objects obtain other through Contract
    ```java
    aContractor.getContracts()
    aCustomer.getContracts()
    ```

**Disadvantages**
- No direct access to collection on other side (need to loop through all the Contract objects to build collection)
- More complex code needed to manage the Contract entity instance to create and remove associations
  - Requires additional infrastructure code
    - Entity Class (Contract)
    - Identifier
  - Intermediate class has to be saved and deleted independently to create links between objects

```java
Contract aContract = newContract(aCustomer.getCustomerId(),
aContractor.getContractorId());
session.save(aContract);
```
Ternary Relationships

- Relationship across three objects
- Leverage the Intermediate Association to include a reference to an additional third Entity

MAPPING FOR CONTRACT (Relationship Entity)
```xml
<class name="Contract" table="CUSTOMER_CONTRACTOR"
    mutable="false">
    <id name="contractId" column="CUSTOMER_CONTRACTOR_ID">
        <generator class="native"/>
    </id>
    <property name="startDate" column="start_date" type="date"/>
    <property name="price" column="price" type="string"/>
    <many-to-one name="customer" column="customer_id"
        not-null="true" update="false"/>
    <many-to-one name="contractor" column="contractor_id"
        not-null="true" update="false"/>
    <many-to-one name="salesRep" column="sales_rep_id"
        not-null="true" update="false"/>
</class>
```
Summary

In this lecture, we:

• Learned that Associations are relationships between Entity classes and Collections are just groupings of scalar data

• Looked at the way Java and databases realize relationships
  – Java: Object references
  – Databases: Denormalized tables, foreign keys, join tables, and ad hoc joins

• Walked through the ways to realize relationships with Hibernate
  – Setting up the mapping files
  – Coding to Interfaces

• Discussed some Hibernate recommended approaches using an ‘intermediate’ object to realize M:M and ternary relationships
Preview of Next Sections

- Understand the differences between Component & Entity classes
- Learn how to map Components
- Walk through ways of realizing inheritance

Questions?

Customized Java EE Training: http://courses.coreservlets.com/
Servlets, JSP, Struts, JSF/MyFaces/Facelets, Ajax, GWT, Spring, Hibernate/JPA, Java 5 & 6.
Developed and taught by well-known author and developer. At public venues or onsite at your location.