Vaadin JPAContainer 1.0 Manual

Petter Holmström

Table of Contents

Introduction

Vaadin JPAContainer is a Vaadin data container that uses the Java Persistence API 1.0 (JPA) for retrieving and storing data. It supports the most common features required by JEE applications out of the box, such as lazy loading, advanced filtering, nested property names and caching. It will probably not solve all the container problems you as a JEE application developer might face, but it will hopefully make your life a little easier.

JPAContainer does not require any specific JPA-implementation or database. It generates standard JPA-QL queries that should execute properly on any compliant JPA 1.0 implementation. However, it has only been tested on Hibernate and EclipseLink.

It is possible to customize JPAContainer to use a completely different object persistence framework than JPA, as long as the entity classes are still annotated with JPA annotations (JPAContainer deduces primary keys and persistent/filterable fields from these annotations).

JPAContainer currently has limited support for joins in the form of a special kind of filter that can be applied to joined properties. JPAContainer implements the `Container.Hierarchical` interface, but the implementation is to be considered experimental in this version. IdClass-primary keys are not supported.

The purpose of this manual is to briefly explain how JPAContainer works and how you can use it in your applications. The manual describes the architecture of JPAContainer and the idea behind the components

that it consists of. The manual also covers the most important features of JPAContainer and points out some potential issues that developers should be aware of when using JPAContainer.

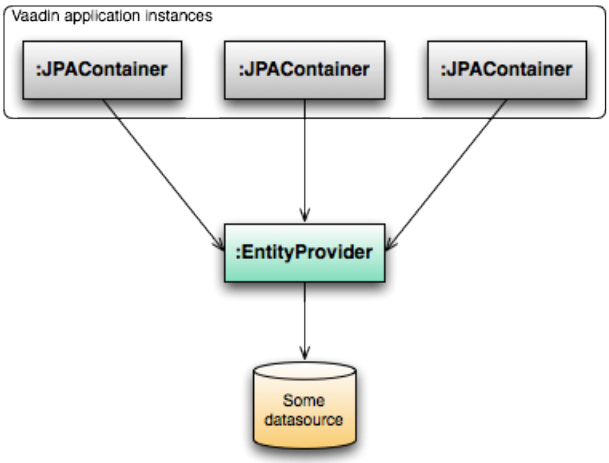
Future Plans

The following features are currently not implemented, but may be added in a future version, depending on the feedback received on version 1.0:

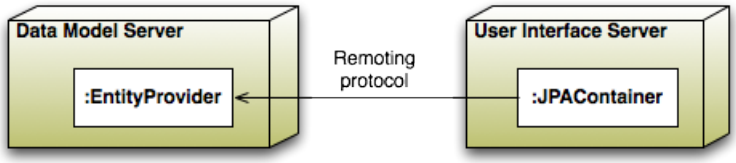
- JEE 6 / JPA 2.0
- Improved support for joined queries
- Improved implementation of the `Container.Hierarchical` interface
- Support for displaying query results that do not consist of entire entities, but arrays of property values
- Support for IdClass-primary keys
- Support for buffered master-detail editing, where the master container gets its data from a data store (like JPAContainer does now) and the detail container from a Collection-property of one of the entities in the master container.

Architecture

The architecture of JPAContainer consists of two main components: an `EntityContainer` and an `EntityProvider`:

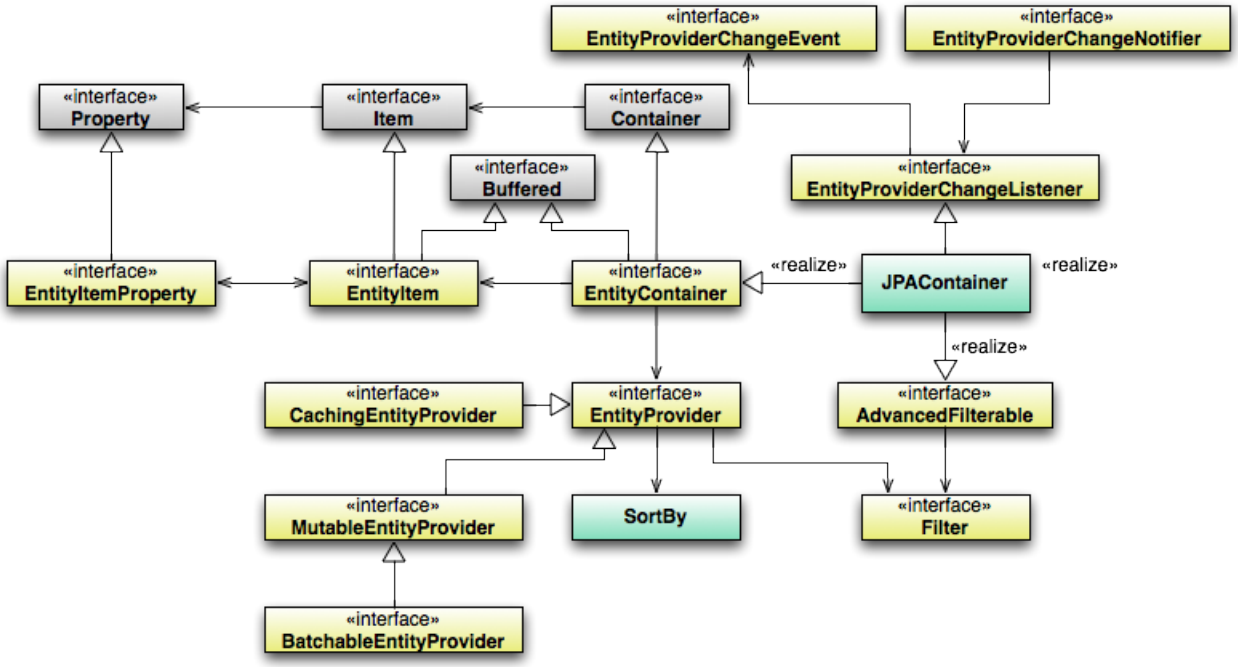


The `EntityContainer` is an extended version of the standard Vaadin container and can be used directly to power e.g. tables or combo boxes. The container gets its data from an `EntityProvider`, which in turn loads the data from some data source. It is possible to have several containers using the same provider, making it possible to e.g. add a second-level cache to the provider. It is even possible to deploy the containers and the provider to different JVMs (provided that detached entities are used instead of managed ones):



Entity providers are stateless in the sense that their behaviour should not depend on the entity container that accesses them. If two entity containers invoke an entity provider with the same arguments, both containers should get the same result, provided that the actual data in the entity provider has not changed between the invocations. This also means that it is possible to export entity providers as services using a stateless protocol such as HTTP.

Most of the public API of JPAContainer is defined in interfaces. The following diagram displays the most important classes and interfaces, and their relations. The gray interfaces are part of the official Vaadin API.



JPAContainer is both the name of the product and the name of the default implementation of the `EntityContainer` interface. In this manual, sections that refer to an "entity container" applies to any implementation of the `EntityContainer` interface (although there currently is only one). Sections that refer to "JPAContainer" applies only to this particular entity container implementation, or to the product as a whole, depending on the context.

Filtering

The filtering of an entity container (defined in the `AdvancedFilterable` interface) is more advanced than the standard Vaadin `Container.Filterable` filtering. Although standard Vaadin filtering is also supported, it is recommended to use the new filtering API in applications. The new filtering API only applies to entity containers, however.

Note

It is likely that a future JPA 2.0 version of JPAContainer will deprecate the advanced filtering API in favor of JPA 2.0 criteria.

Filters are specified by adding instances of the `Filter`-interface to the container using the `AdvancedFilterable.addFilter(..)` method. As filters are intended to be applied in the database, only persistent properties can be filtered. All filters generate standard JPA QL-language that can be used by entity providers when constructing queries to be passed to an `EntityManager`. Entity providers that do not use JPA can analyze the object graph of `Filter`s instead of using the generated JPA QL.

Note

All filtering is done in the database. Therefore special care should be taken to make sure filterable columns are indexed properly, especially if the number of records is large.

Editing

The entity container supports editing, but it uses a slightly different API than the default Container API. In order to make an entity container editable, the entity provider must implement the `MutableEntityProvider` interface. In addition, the container must not be marked as read-only. Attempts to a read-only entity container will result in an exception.

There is only one way of adding new items to the container and that is by using `EntityContainer.addEntity(..)`. Note, that this method takes the entity instance and not the `EntityItem` as a parameter. The returned value is an identifier that can be used to get the newly added entity's `EntityItem` by passing it to `EntityContainer.getItem(..)`.

Changes made to existing `EntityItem`s will automatically be propagated back to the container, which in turn will decide what to do with the change (see the next section about buffering). Existing items can be removed from the container using the standard `Container.removeItem(..)` method.

Note

All entity items are bound to their specific containers. Thus, it is not possible to move an entity item from one container to another!

Transactions

Transactions are handled by the entity providers. Depending on how they are deployed, transaction can be either handled by an external container such as Spring or EJB or internally by using the transaction methods of JPA. The JavaDocs contain more information about which methods need to run inside a transaction and which ones need not. However, the scope of a transaction must always be limited to a single method invocation. In other words, a transaction may not begin before the method invocation begins and it must be either committed or rolled back before the method invocation ends. The reason for this is to keep the entity providers stateless.

Some examples of how to use container managed transactions can be found in the section called “Using JPAContainer in Applications”.

Buffered mode

Both `EntityContainer` and `EntityItem` extend the `Buffered` interface. This means that it is possible to buffer changes made to either individual items or to the container as a whole.

The `Buffered` interface can be used to control both how data is read and how changes are written:

Write-Through	Controls whether changes should be written directly (on) or buffered (off).
Read-Through	Controls whether changes should be read from the original data store (on) or from a buffer or cache (off).

Batching Entity Providers

Before moving on to container buffering and item buffering, some words should be mentioned about the `BatchableEntityProvider` interface.

`BatchableEntityProvider` extends the `MutableEntityProvider` interface with an API for performing update operations (i.e. add, update and remove) in batches. The method in question is defined like this:

```
public void batchUpdate(BatchUpdateCallback<T> callback)
    throws UnsupportedOperationException;
```

A callback object is used to perform the actual batch:

```
public interface BatchUpdateCallback<T> extends Serializable {
    public void batchUpdate(
        MutableEntityProvider<T> batchEnabledEntityProvider);
}
```

The `batchEnabledEntityProvider` parameter is a `MutableEntityProvider` that is aware of the fact that all the operations are being performed inside a batch.

For example, the following code could be used to add a list of entities to the entity provider inside a single batch:

```
provider.batchUpdate(new BatchUpdateCallback<MyEntity>() {
    public void batchUpdate(MutableEntityProvider<MyEntity>
            batchEnabledEntityProvider) {
        for (Entity e : myListOfEntitiesToAdd) {
            batchEnabledEntityProvider.addEntity(e);
        }
    }
});
```

The reason for using a callback instead of directly performing the modifications on the entity provider is the stateless nature of entity providers and the requirement on a transaction to be limited to a single method invocation. If the batch were performed without a callback, each individual operation would run inside its own transaction that would be committed before moving on to the next next operation. This in turn would make it more difficult to roll the entire batch back if something went wrong.

There are a few potential issues to keep in mind while using batching entities that contain references to other entities inside the same batch. The first thing to keep in mind is how cascading has been configured. Let us say that two entities have been added to the list in the example above and that the first entity contains a reference to the second entity. If cascading is turned on, both entities will be persisted when the first entity is added to the entity manager. If this has not been detected by the batch enabled entity provider (as in the example above), it might proceed by adding the second entity to the entity manager. As a result, there might now be two copies of the second entity in the database (though with different entity IDs).

A similar problem might occur if some entity references a newly created entity that is not a part of the batch. Let us say that EntityA and EntityB have both intitially been added to the list of entities in the example above. EntityA is then updated to reference EntityB. However, the user realizes that he or she has made a mistake and removes EntityB from the list. EntityA will, however, still hold a reference to EntityB. When EntityA is persisted, the operation will cascade to EntityB, and suddenly EntityB is in the database although it was never included in the batch.

If you think your application might run into problems like these, there are a few things you could do. First, make sure you perform enough validation every time you remove or edit an entity so that any illegal references are cleaned up. Second, implement your own batching entity provider so that it is aware of the potential problem situations and is able to deal with them.

Container Buffering

The buffering capabilities of the container heavily depend on the capabilities of the underlying entity provider:

Write-Through
: When turned on, any changes are directly propagated to the entity provider. When turned off, all changes are buffered in the container and sent to the entity provider only when `commit()` is called. The changes can also be rolled back using `discard()`.

  It is important to remember that buffered changes are *not* considered when the data is filtered or sorted, as these operations are performed by the entity provider. Thus, newly added items always show up at the top of the container regardless of any sorting applied, and modified items are filtered and sorted according to their "unmodified state". Only after the changes have been committed, filtering and sorting will be applied to the new values.

  By default, write-through is turned off if the entity provider supports it. To support buffering of changes, the entity provider must implement the `BatchableEntityProvider` interface (see the previous section). `JPAContainer` will keep a log of all the changes that have been made and will then pass these changes on to the entity provider in the same order as they were made. If an entity is added and later removed before the changes are committed, the entity will not be included in the change log. It might still turn up in the database if it is referenced by other entities that are in the changelog, due to the cascading problems described in the previous section.

  Write-through is also called Auto Commit.

Read-Through
: Read-through cannot be explicitly changed by the user. Read-through is off if the entity provider implements the `CachingEntityProvider` and the cache is in use. Otherwise, read-through is always on, i.e. the data is read directly from the data store.

Item Buffering

The buffering capabilities of `EntityItem` are always the same regardless of the entity provider. One of the drawbacks when using write-through in the container is that every time a property of an item is changed, the change is sent directly to the database and saved. This means a lot of database round-trips and no way of discarding the changes. The solution to this problem is to either turn on buffering in the container or, if the container does not support buffering, use buffering on the item level instead.

Write-Through
: When write-through is on (default), all changes made to a property are directly propagated back to the underlying entity object and the container. When write-through is off, all changes are buffered inside the item and are only propagated to the underlying entity (and the container) when `Buffered.commit()` is called. `Buffered.discard()` discards the changes and reloads the item with the original property values. Write-through has to be explicitly turned off for each `EntityItem` instance that requires it.

Read-Through — When read-through is on (default), all data is read directly from the underlying entity object. This is possible even when write-through is off and there are buffered changes. When read-through is off, data is read both from the underlying entity object (unchanged properties) and the item buffer (changed properties). Read-through has to be explicitly turned off for each `EntityItem` instance that requires it.

Note

Item-level buffering does not work properly with collections, unless the entire collection instance is replaced with another collection instance when items are added or modified. The same goes for other mutable objects that are not modified via nested properties.

Experimental Hierarchical Support

It is possible to use JPAContainer as a hierarchical container if the entities in the container can be related to each other by means of a parent property. For example:

```
@Entity
public class Node {
    ...
    @ManyToOne
    private Node parent;
    ...
}
```

The API is defined in the `HierarchicalEntityContainer` interface.

Note

The `JPAContainer` class contains a limited and *experimental* implementation of this interface. When it is used as a hierarchical container, the data is always read directly from the entity provider regardless of whether it is using buffering or not. Therefore, this feature should be used with care in production systems!

Managed Entities

Normally, it is easier to use detached entities in an entity container, as this means that no changes will be automatically propagated back to the database unless explicitly requested. On the other hand, this also means that any references or collections that will be accessed by the container have to be eagerly fetched before the entity is detached, which in turn can lead to some serious performance problems.

If desired, JPAContainer is able to use managed entities, i.e. entities that are managed by a persistence context. This makes it possible to lazily load both references and collections. It also means that any changes made to the entities show up in the persistence context directly and are persisted to the database when the entity manager is flushed.

However, there are a few things to keep in mind when using managed entities:

1. The persistence context must be configured to use extended scope instead of transaction scope. That way, the persistence context will be available for several transactions, which is a requirement if lazy loading is to be used in a web application.

2. The entities returned from the entity provider must not be serialized and deserialized before they reach the container. In practice, this means that the entity provider and the container must run inside the same JVM.

3. If editing entities is allowed, the entity provider should take care of flushing the entity manager when needed.

It is possible to ask the entity provider to explicitly detach the entities before they are returned to an entity container. If the `EntityProvider.isEntitiesDetached()` method returns true, it is safe to assume that all entities returned by the provider are detached. If the method returns false, the entities may or may not be detached, depending on how the entity provider is implemented. For example, if an entity provider uses a transaction-scoped persistence context and each method runs inside its own transaction, the entities returned by the provider will be automatically detached.

Multi-User Environments

When used in read-only mode, JPAContainer works great in multi-user environments. However, as soon as data editing is introduced, there are a few things to keep in mind.

JPAContainer implements the `EntityProviderChangeListener` interface. If the entity provider implements the `EntityProviderChangeNotifier`, JPAContainer will register itself as a listener. Everytime the entity provider notifies JPAContainer that an entity has been added, updated or removed, JPAContainer will fire an item set change event. Note, however, that as Vaadin currently does not support server push, the client will not be updated until the browser sends a request to the server, e.g. when the user clicks on something or selects an item. If the item set change causes the previously mentioned selection to change, the user may experience some strange behaviour as the item he or she selected suddenly becomes unselected without warning. It is possible to turn off this functionality by using the `JPAContainer.setFireContainerItemSetChangeEvents(..)` method.

When using a Vaadin table to show the contents of a container, there is a risk of the view and the model becoming out of synch with each other. Everytime an item is selected in a table, the table will check with the container if such an item exists before the selection is changed. However, in the user interface, the selection will change regardless of this check. This means that if a user selects an item that has been deleted, the user interface will look like it has selected the deleted item, when the underlying model in fact contains the previous selection.

To work around this, there is a property in JPAContainer called `containsIdFiresItemSetChangeIfNotFound`. If this property is true, the container will fire an item set change event every time `EntityContainer.containsId(..)` is called and the result is false. Thus, if the user tries to select a deleted item, the table will automatically be updated and the deleted item(s) removed from the view.

Note

This is a hack that will hopefully be removed in the future once the problem has been solved in a better way.

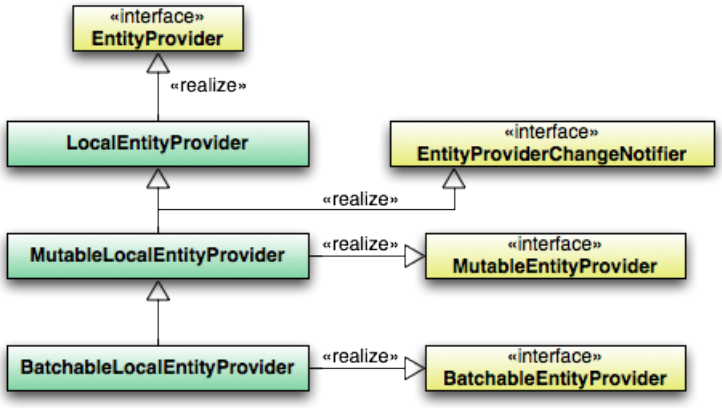
If JPAContainer will be used for data that is changed frequently (especially if existing data is updated or removed), it would probably be best to implement a custom entity provider that takes a snapshot of the data to be displayed and serves information from the snapshot. Otherwise, runtime errors might occur, e.g. if the data is removed between a call to `EntityProvider.getEntityCount()` and `EntityProvider.getEntityIdentifierAt(..)`.

Built-in EntityProviders

There are two kinds of built-in entity providers. The `LocalEntityProvider` is the simplest one, which reads all the information directly from an `EntityManager`. In applications where the amount of data is small and database round-trips are fast, this provider is the safest choice.

If the number of database round-trips should be reduced, `CachingLocalEntityProvider` should be used instead. It maintains a local cache of entities and query results and hence performs faster than `LocalEntityProvider` if database round-trips are slow. However, it also requires more memory than `LocalEntityProvider`.

The LocalEntityProvider



LocalEntityProvider

A read-only, lazy loading entity provider that performs no caching and reads its data directly from an `EntityManager`. In other words, basically all calls to the entity provider result in a query being sent to the `EntityManager`.

MutableLocalEntityProvider

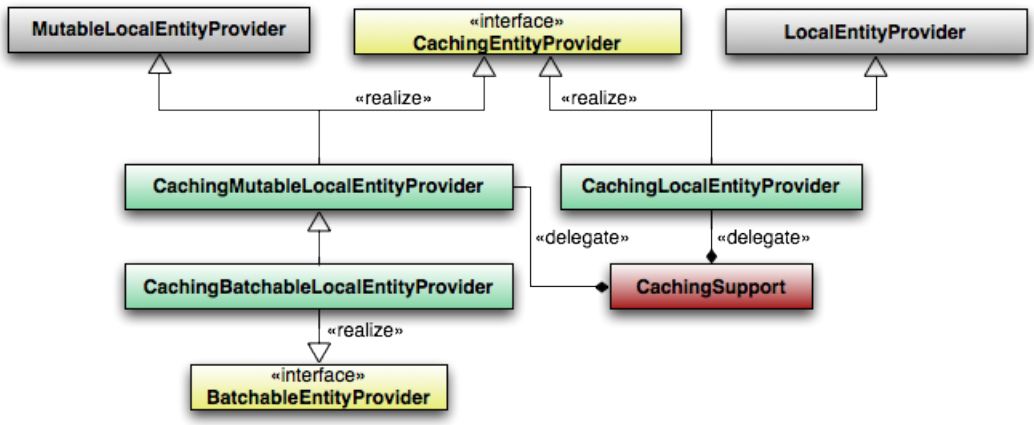
Extends `LocalEntityProvider` with write support. All changes are directly sent to the entity manager. Transactions can be handled either internally by the provider (default) or by the container (e.g. by extending the class and annotating the methods as shown in the section called “Using JPAContainer in Applications”). The class also implements the `EntityProviderChangeNotifier` interface, which means that it will notify any listening clients everytime an entity is added, updated or removed.

BatchableLocalEntityProvider

The simplest possible implementation of the `BatchableEntityProvider` interface - an extension of `MutableLocalEntityProvider` that simply passes itself to the `batchUpdate(..)` method. This will work properly if the entities do not contain any references to other entities that are managed by the same container (see the section called “Batching Entity Providers”).

The CachingLocalEntityProvider

All the caching entity providers are basically extensions of the local entity providers, with caching support added by a delegate class `CachingSupport`.



CachingLocalEntityProvider

A read-only, lazy loading entity provider that caches both entities and query results for different filter/sortBy combinations. When the cache gets full, the oldest entries in the cache are removed. The maximum number of entities to cache and the maximum number of entityIds to cache for each filter/sortBy combination can be manually set. The cache can also be manually flushed. When the cache grows full, the oldest items are removed.

CachingMutableLocalEntityProvider

A caching entity-provider with caching support. When an entity is added or updated, the cache is flushed in order to make sure the added or updated entity shows up correctly when using filters and/or sorting. When an entity is removed, only the filter/sortBy-caches that actually contain the item are flushed. Otherwise, this class works just like `MutableLocalEntityProvider`.

CachingBatchableLocalEntityProvider

An extension of `CachableMutableLocalEntityProvider` that implements the `BatchableEntityProvider` and passes itself to the `batchUpdate(..)` method. This will work properly if the entities do not contain any references to other entities that are managed by the same container (see the section called “Batching Entity Providers”).

Using JPAContainer in Applications

In this section we are going to look at how to use JPAContainer in your own applications. First of all, you have to add `vaadin-jpacontainer-VERSION.jar` to your application's classpath. Then, all you need is the following code:

```
EntityContainer<MyEntity> container = new JPAContainer<MyEntity>(
    MyEntity.class);
container.setEntityProvider(myEntityProvider);
```

The entity provider can be any of the built-in entity providers (see the section called “Built-in EntityProviders”) or your own custom built entity provider (see the section called “Developing Custom Entity Providers”). If you use the built-in entity provider, you also have to have JPA configured correctly in your application (e.g. a `persistence.xml` file). For example, to use the

`MutableLocalEntityProvider` to provide entities of class `Customer` from an existing entity manager, you could use the following code:

```
MutableLocalEntityProvider<Customer> myEntityProvider =
    new MutableLocalEntityProvider<Customer>(
        Customer.class, entityManager);
```

One thing to keep in mind when working with JPAContainer is that it is centered around *entities*, i.e. POJOs annotated with JPA annotations. JPAContainer will analyze the class definition and the annotations to access properties, determine which properties are sortable, etc. JPAContainer will *not* work with objects of classes that lack JPA annotations.

JPAContainer makes a difference between persistent properties and transient properties. Persistent properties are determined from the JPA annotations. If an entity class uses field annotations, the names of the persistent properties are the names of the non-transient fields. This is important to remember, especially if the field names and their corresponding getter/setter methods do not have matching names.

If an entity class uses method annotations, the names of the persistent properties are the JavaBean names of the non-transient getter methods. This also applies to transient properties, which are all the transient JavaBean properties of the entity class. Transient properties may be read-only, whereas persistent properties are always writable.

Nested Properties

JPAContainer supports nested properties. You define nested properties using the `EntityContainer.addNestedContainerProperty(..)` method. For example, let's say you have an entity class that looks like this:

```
@Entity
public class Person implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @Version
    private Long version;
    private String firstName;
    private String lastName;
    @Temporal(TemporalType.DATE)
    private Date dateOfBirth;
    @Embedded
    private Address address;

    // Getter and setter methods omitted
}

@Embeddable
public class Address implements Serializable, Cloneable {

    private String street;
    private String postalCode;
    private String postOffice;
```

```
    // Getter and setter methods omitted
}
```

Now, let's say you want to show a list of persons, together with their addresses, in a Vaadin table. You also want to be able to filter and sort by street, postal code and post office. All you need to do to make this possible is the following:

```
myContainer.addNestedContainerProperty("address.*");}
```

The wildcard asterisk will expand to all the properties of the `Address` class. After this method call, the container will contain the following new properties, in addition to the existing ones: `address.street`, `address.postalCode` and `address.postOffice`. These properties are no different from the other properties and can be used in the same way.

Note

Although the `address` property used in the example is embedded, this is not a requirement. The property could just as easily have been a ManyToOne reference or even a transient property. However, transient nested properties cannot be used for filtering and sorting.

It is also possible to add individual nested properties. In that case, simply replace the wildcard asterisk with the property name.

Nested properties can be removed from the container using the `EntityContainer.removeContainerProperty(..)` method. Removing a property from the container does not affect the property value, but prevents the container from accessing its value.

Nested properties can also be defined on the item level. Normally, the properties accessible from an item is all the available properties of the entity class and any nested properties defined in the owning container. If these are not enough, additional nested properties can be added to a specific item by using the `EntityItem.addNestedContainerProperty(..)` method. Nested properties added this way will not show up in the container.

Predefined Filters

There are several predefined filters available in the `Filters` class, found in the com.vaadin.addon.jpacontainer.filter package:

```
container.addFilter(Filters.like("firstName", "Jo%", false));
container.addFilter(Filters.or(
    Filters.eq("lastName", "Smith", false),
    Filters.eq("lastName", "Cool", false)
));
```

The above example would accept all entites whose first names start with "Jo" and last names are either "Smith" or "Cool" (case ignored). Filters can be applied to other data types as well, such as integers and timestamps. It is also possible to create custom filters by implementing the `Filter` interface.

Filters can be either applied immediately or explicitly, depending on the state of the `AdvancedFilterable.isApplyFiltersImmediately` flag. When applied immediately, each call to `AdvancedFilterable.addFilter(..)` forces the container to refresh itself. If only one filter is to be added, this may be desirable. However, if several filters are to be added it may be better to wait until all the filters have been defined before applying them. In this case, `AdvancedFilterable.applyFilters()` has to be called to apply the filters.

Filtering Joined Properties

The `JoinFilter` is treated in a different way than the other filters. The filter is used to apply filters to a joined property. The alias of the joined property is always the name of the joined property. For example, the following filter applied to a container of `Person` entities:

```
Filters.joinFilter("skills", Filters.eq("skill", s));
```

generates a JPA-QL query similar to this:

```
SELECT obj.id
    FROM Person AS obj
    JOIN obj.skills AS skills
    WHERE skills.skill = :someparametername
```

Note

The queries are not distinct. The reason for this is that some JPA implementations require the properties used in the order by clause to be distinct, which is not possible in JPAContainer.

A `JoinFilter` may not contain other `JoinFilter`s.

Editing Items Without Adding Them to the Container

Especially when using a Vaadin form for editing items, it might be necessary to wrap an entity object inside an `EntityItem` before the entity has been added to the container. This can be done by using the `EntityContainer.createEntityItem(..)` method. This method will create a new `EntityItem` that has access to all the fields defined in the container, but is not contained in the container yet. The following code example demonstrates its usage:

```
EntityItem<Customer> myNewItem = customerContainer.createEntityItem(
    new Customer());
// Do something with the item, e.g. open a modal dialog
Object id = customerContainer.addEntity(myNewItem.getEntity());
item = customerContainer.getItem(id);
```

EntityProviders as Spring-managed Beans

If you are creating an enterprise application using the Spring Framework, you might want to configure your entity providers as Spring managed beans. In the demo application, this is done by subclassing one of the built-in entity providers and adding the appropriate annotations to the subclass. For example:

```
@Repository(value = "myEntityProvider")
public class MyEntityProviderBean
    extends MutableLocalEntityProvider<MyEntity> {

    @PersistenceContext
    private EntityManager em;
```

```
    protected LocalEntityProviderBean() {
        super(MyEntity.class);
  setTransactionsHandledByProvider(false);
    }

    @Override
    @Transactional(propagation = Propagation.REQUIRED)
    public MyEntity updateEntity(MyEntity entity) {
        return super.updateEntity(entity);
    }

    @Override
    @Transactional(propagation = Propagation.REQUIRED)
    public MyEntity addEntity(MyEntity entity) {
        return super.addEntity(entity);
    }

    @Override
    @Transactional(propagation = Propagation.REQUIRED)
    public void removeEntity(Object entityId) {
        super.removeEntity(entityId);
    }

    @Override
    @Transactional(propagation = Propagation.REQUIRED)
    public void updateEntityProperty(Object entityId,
            String propertyName, Object propertyValue)
            throws IllegalArgumentException {
        super.updateEntityProperty(entityId, propertyName,
            propertyValue);
    }

    @PostConstruct
    public void init() {
        setEntityManager(em);
        /*
         * The entity manager is transaction scoped, which means that
         * the entities will be automatically detached when the
         * transaction is closed. Therefore, we do not need to
         * explicitly detach them.
         */
        setEntitiesDetached(false);
    }
}
```

EntityProviders as Stateless Session Beans

You can also deploy your entity providers as stateless session beans (or, if you are using one of the caching providers, as stateful session beans). The idea is the same as for Spring managed beans:

```
@Stateless
@TransactionManagement
```

```
public class MyEntityProviderBean extends
    MutableLocalEntityProvider<MyEntity> {

    @PersistenceContext
    private EntityManager em;

    protected LocalEntityProviderBean() {
        super(MyEntity.class);
  setTransactionsHandledByProvider(false);
    }

    @Override
    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public MyEntity updateEntity(MyEntity entity) {
        return super.updateEntity(entity);
    }

    @Override
    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public MyEntity addEntity(MyEntity entity) {
        return super.addEntity(entity);
    }

    @Override
    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public void removeEntity(Object entityId) {
        super.removeEntity(entityId);
    }

    @Override
    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public void updateEntityProperty(Object entityId,
            String propertyName, Object propertyValue)
            throws IllegalArgumentException {
        super.updateEntityProperty(entityId, propertyName,
            propertyValue);
    }

    @PostConstruct
    public void init() {
        setEntityManager(em);
        /*
         * The entity manager is transaction scoped, which means that
         * the entities will be automatically detached when the
         * transaction is closed. Therefore, we do not need to
         * explicitly detach them.
         */
        setEntitiesDetached(false);
    }
}
```

Developing Custom Entity Providers

Although the built-in entity providers should be sufficient in every-day usage, there are use cases where you might have to develop a custom entity provider. These include customized queries, complex object structures that require special handling, or loading data from a different data store. Although JPAContainer requires JPA annotations in order to properly analyze the classes and extract the available properties, the entities themselves need not necessarily come from an EntityManager. In fact, by implementing your own entity provider, you can store them in any way you like.

Before you start to implement your own entity provider, there are a few things you need to think of:

- Do you need read-only or read-write support?
- Do you need caching?
- Do you need filtering and/or sorting?
- Do you need container-level buffering?
- How are you going to handle transactions?
- Will you load data directly from the data store or from a snapshot taken of the data store?
- How are you going to handle concurrent editing?

Once you know what you need to build, you have to pick the interfaces to implement:

EntityProvider	Basic interface for all entity providers, provides read-only support.
MutableEntityProvider	If you entity provider requires read-write support.
CachingEntityProvider	If your entity provider uses caching or loads its data from a snapshot of the data store.
BatchableEntityProvider	If you require container level buffering.
EntityProviderChangeNotifier	If you want the containers to refresh themselves when data is changed.

Please check the JavaDocs for more information about how the interfaces should be implemented.