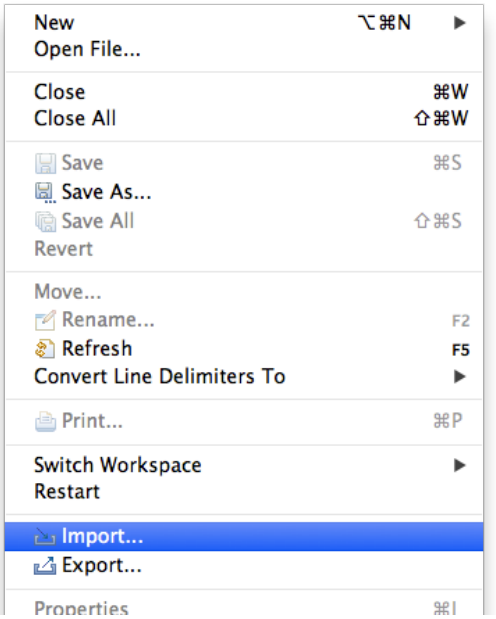


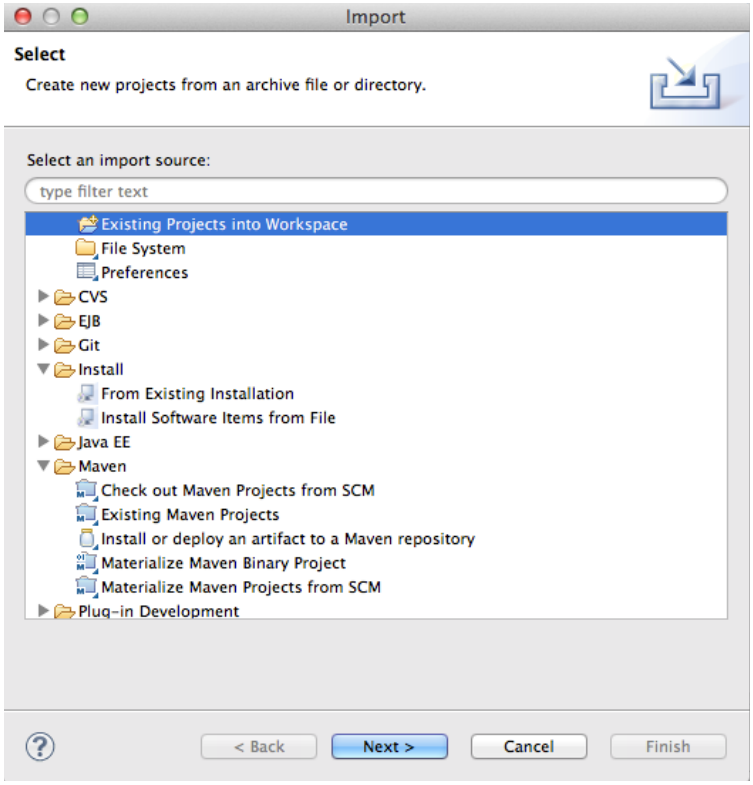
Instructions

Setting up the environment

Start by downloading the advanced.zip file and extract it to location of your choice.

Open Eclipse, choose File > Import and choose “Existing project into Workspace”





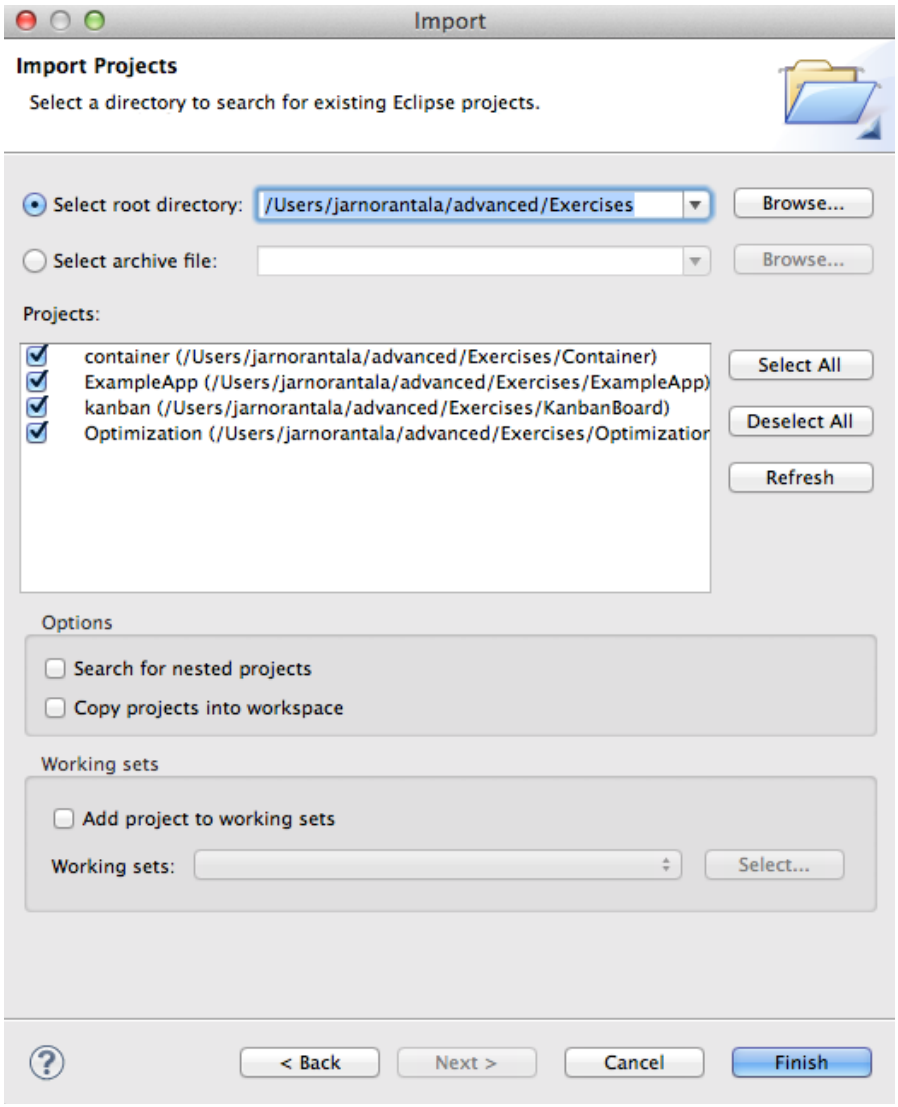
Browse to the folder where you extracted the zip, and choose the Exercises folder. Select the ExampleApp, Optimization and container projects and click on finish.

Right-click on the imported projects and choose Ivy > Resolve.

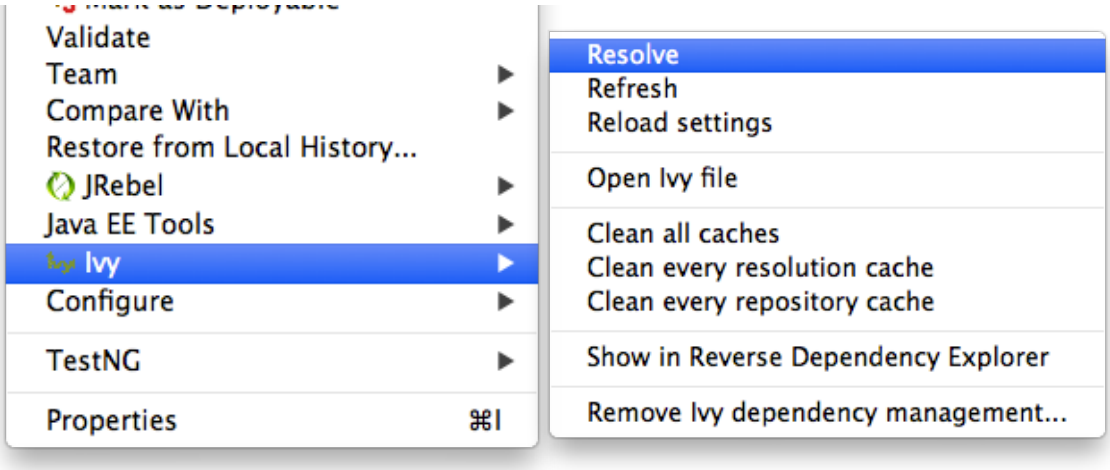
You should now be all set up. Try deploying the project to Tomcat and browse to http://localhost:8080/ExampleApp/

Background

Our client wants to create a web application for monitoring their expenses on a department basis. In the first sprint the application will contain three views: A Dashboard that is a simple placeholder view, an Auditing view which contains a list of audit log messages and a Department view that is used to maintain all of the employees in a department.



This initial version of the application will only show the expenses and employees of the “Services” department. The customer has provided a stub implementation of their domain model and services.



The customer is not satisfied with the current structure of the application code. As a distinguished Vaadin expert, you have been hired to help with the project. Your task will be to implement the navigation for the project and restructure the application to be more maintainable. The stub project contains the initial implementation of the application excluding navigation.

Exercise 1: Navigation

The first exercise is to implement navigation between views. Create the base application so that it uses the Navigator API to handle the views and navigation between them.

Things you'll need to do

1. There are three views in the application, make the views Navigator compatible by implementing the *com.vaadin.navigator.View* -interface
2. In the UI class, create a navigator instance and register views to the navigator
3. Implement the click listeners in the UI class so that clicking on a button will transfer the user to the corresponding view
4. Update any possible data in the views using the newly created enter() method.

Bonus task: The URL to access the department view is http://localhost:8080/ExampleApp/#!department

It is possible give parameters to views which can then be handled in the view's enter method. Your task is to check, if an person ID is given as a view parameter, then the person with that ID should automatically be selected from the table for editing.

Book of Vaadin
Navigator: 383

Exercise 2: Presenters

The customer has requested you to refactor their application to implement the Model-View-Presenter pattern. You'll start this task by refactoring application logic into presenters and creating a layer of abstraction (using interfaces) between the presenter and the view implementation.

Improve code compartmentalizing and testability by separating UI code and -logic from each other. Views should still listen to events from the UI components but the presenter will decide what should be done when an event happens. Presenters are also the entry point to other subsystems like services.

Things you'll need to do:
1. Make views implement an interface that the presenters will use
2. Create a presenter for the Department view.
3. The view will create its own presenter upon construction and give a reference to itself in the presenters constructor.
4. The contents of the view should be refreshed upon navigation
5. Move the business logic from the view implementations to the presenters.
6. DepartmentPresenter should handle at least:
   - Fetching employees from service and updating the view with them upon navigation
   - Handle the saving of a *Person*-entity through the service
   - Handle the cancel event
   - Decide what to do when a *Person* is selected

When using the Vaadin Navigator API, one can easily separate the navigation event from the view creation event. If the view is given as a class-reference to the navigator, the view will be recreated upon each navigation. However, if an instance of the view is given to the navigator the state of the view is preserved. We should try to make our views so that either mode can be used. This is why we should only create the components of the view in the constructor and fill/refresh the data of the view on the *enter*-event.

Below is illustrated how the enter pattern can be used in the Auditing view.

```
public class AuditingPresenter {
      private AuditingView view;

      public void setView(AuditingView auditingView) {
            view = auditingView;
      }

      public void enter() {
            fetchInitialData();
      }

      private void fetchInitialData() {
            for (String message : AuditLogService.getAuditLogMessages()) {
                  view.addAuditLog(message);
            }
      }

}
```

```
public class AuditingViewImpl extends VerticalLayout implements View,
          AuditingView {

      private AuditingPresenter presenter;
      private CssLayout messageLayout;

      public AuditingViewImpl() {
            ...

            presenter = new AuditingPresenter();
            presenter.setView(this);
      }

      @Override
      public void enter(ViewChangeEvent event) {
            messageLayout.removeAllComponents();
            presenter.enter();
      }

      @Override
      public void addAuditLog(final String message) {
            messageLayout.addComponent(new Label(message));
      }

}
```

Below is the View interface of the example solution. It should give you an idea of what needs to be done.

```
public interface DepartmentView {

      public void setEmployees(List<Person> employees);

      public void showEmployeeInForm(Person employee);

      public Person commitChanges();

      public void discardChanges();

      public void selectEmployee(Person employee);

      public void setDepartment(Department department);

}
```

Exercise 3: Push

Sometimes we have operations in our backend that takes a while to perform. If we perform there processes in the same thread as in which the HTTP request is processed, we will block any user interface interactions until the process is done. In most cases, we want the user to be able to continue using the user interface even though we are doing “heavy stuff” in the backend. Hence, the heavy process are typically performed in a separated thread.

If we modify the user interface from a thread that is outside the HTTP request, we won’t see the changes in the browser until the user does something that triggers an HTTP request (so that we can send the changes of the UI in the response of that request). For the changes to be seen in the user interface immediately, we can use AJAX push, which will allows us to “push” the changes to the browser, even if we don’t have an active HTTP request going on.

In this exercise, we will practice enabling push in a Vaadin application. The situation we are trying to simulate is that PersonService’s getEmployees method takes multiple seconds before it returns its data. We want to push the user information to the UI when the process is done. On a high level, we need to enable push support in the Vaadin application, make the heavy process to be accessed in a separate thread and then handle possible concurrency issues that may occur if we modify the UI from a thread outside the HTTP request.

Things you’ll need to do enable push support:

1. Add `<async-supported>true</async-supported>` to your web.xml under `<servlet>`
2. Adding push dependencies to ivy.xml

```
<dependency org="com.vaadin" name="vaadin-push"
rev="&vaadin.version;" conf="default->default" />
```

3. Enabling push for your application by adding `@Push` annotation for your UI class

Things you’ll need to do in your application:

1. We want to have a progress bar in the user interface showing that we are loading data. This component already exists within the DepartmentInfo class. To make it visible, we need to call setLoadingState on the class. It takes as an argument a float value between 0 and 1, indicating the loading progress. Given the value 0, the progress bar will become visible. Given the value 1, it will be hidden.

   Add the new method, `public void setDataLoadingState(float percentageProgress)`, to your DepartmentView. The implementation of this method should update the loading state in the DepartmentInfo class.

2. In your DepartmentPresenter class, create an ExecutorService that will provide you a safe way to get threads for executing long running processes in the backend.

```
private final ExecutorService pool =
Executors.newFixedThreadPool(10);
```

3. Create an inner class that implements the `Runnable` interface, this runnable is responsible for fetching the persons from the backend and giving the details to the view once the data has been loaded. Here is a pseudo implementation of the runnable you need

```
class UIUpdateRunnable implements Runnable {
        ProgressingFuture<List<Person>> employeesAsync;
```

```
        @Override
        public void run() {
            // call backend's getEmployeeAsync() to fetch data. Method will return
            // a future object

            // while the future is not ready to be accessed (isDone returns false),
            // wait for it to complete

            // once the future is done, get the employee list from the future (call
            // employeesAsync.get()) and set the employee list to the view

            // update the data loading state to 1, so that the progress bar will be
            // hidden
        }
    }
```

4. In the presenter's enter method, you need to
   4.1. clear the table from old data, you can do this my giving the view an empty list of employees
   4.2. trigger the UIUpdateRunnable by calling `pool.submit(new UIUpdateRunnable());`
   4.3. call the view's setDataLoadingState(0) in order to make the progress bar visible
5. Modify your view to handle UI changes from background threads in a thread safe way, this is done by using the `access()` method. Your backend threads make modifications to the view through three methods, selectEmployee(), setDataLoadingState() and setEmployees(). We need to secure those three methods. Below is the example implementation for setDataLoadingState.

```
@Override
public void setDataLoadingState(final float state) {
    UI ui = getUI();
    if (!isAttached() || ui.getSession().hasLock()) {
        departmentInfo.setLoadingState(state);
    } else {
        ui.access(new Runnable() {
            @Override
            public void run() {
                departmentInfo.setLoadingState(state);
            }
        });
    }
}
```

Bonus task: Make the progress bar show the actual loading progress. The ProgressingFuture will give you the loading state of the backend, now try to figure out what you need to do to get that visible in the UI.

Bonus task 2: Change the auditing view so, that whenever a new auditing message is added, the message will be automatically pushed to the browser. Note that the auditing service already has a notification mechanism which notifies listeners, when new messages are added.

Book of Vaadin
Server Push: 410

Exercise 4: Separate models from entities

In this exercise we will separate the UI from the back-end by using proxies and DTOs. This will provide resilience for the UI from potential back-end or entity model changes. Proxies will mediate data between the entities and the UI.

Things you'll need to do:
1. Create an EmployeeProxy which will function as the model for a Person entity
2. Create DepartmentModel as a DTO that will function as the view's model. It should contain two fields, a String field for the department's name and a list of employees.
3. In the presenter: Fill in the Employee proxies with data from the PersonService
4. In the presenter: Fill the DepartmentModel with data from the AuthenticationService User object
5. In the presenter: Fill the DepartmentModel with the employee proxies
6. Refactor the UI components to handle proxies instead of entities

Tip: Vaadin data binding API uses reflection to search for the getters and setters for fields. Hence if a field is named "firstName", Vaadin will look for a getter "getFirstName()" and a setter "setFirstName()". This correlates against the @PropertyId(<fieldName>) annotations in for example EmployeeEditor.

Bonus task: If you've implemented the first bonus task of exercise 3, you will notice that you have a small problem: the presenter is still giving information directly to the view - the loading status. This data should also be in the view, but then your problem is, how do you update the view without rerendering the department name and employees table?

You should set a model to the view once and just update the content of the model. The model is responsible for sending an event whenever its data changes - the view listens to these changes and updates the view accordingly. The easiest way to implement this observer-pattern, is to use PropertyChangeSupport and PropertyChangeListener. Your model should have an instance of PropertyChangeSupport and your view should implement PropertyChangeListener and register itself as a listener to the model. Here's a hint how it all works. This is code that goes into your model:

```
private final PropertyChangeSupport propertyChangeSupport = new
PropertyChangeSupport(this);
...
public void addPropertyChangeListener(PropertyChangeListener arg0) {
            propertyChangeSupport.addPropertyChangeListener(arg0);
      }
...

public void setEmployees(List<EmployeeProxy> employees) {
            List<EmployeeProxy> old = this.employees;
            this.employees = employees;
            propertyChangeSupport.firePropertyChange("employees", old,
employees);
      }
```


Optimization

Exercise 5: Optimization of layout rendering time

The key to this exercise is that not everything as it seems and to simplify, simplify, simplify.Your task is to optimize the rendering time of the given application. To see the rendering time of your application, add the parameter “?debug” to the URL and look at the bottom of the debug window for the phrase “Processing time was <x>ms for <y> characters of JSON”. X marks the spot, that is the rendering time.

Typical cause for slowly rendering layouts is the excessive use of components, deep layout hierarchies and slow layouts. With Vaadin 7, the layout performance between different layouts has decreased, but there are still differences. Play around and try to see what affects the rendering time.

Containers

Exercise 6: Implementing a basic container

The goal of this exercise is to create a simple read-only container implementation, make it fire ItemSetChangeEvents, and connect it to a provided data source. We will use Integers as itemIds in this container.

You will be provided with a pre-made Vaadin project which contains the following classes / interfaces:

DataProvider<T>

DataProvider is a generic interface for any data source implementation that provides data as objects of type T. This interface has (among others) the following methods which we will use in the first exercise:

```
public int getCount();
```
returns the current count of objects contained in the data source

```
public List<T> fetchItemBatch(int startIndex, int count);
```
this method fetches a count-sized batch of objects, starting at the given index

DataProviderImpl

This class is a simple implementation of the DataProvider interface and provides Contact objects. You can treat this class as a black box and there should be no need to interact with its code.

Contact

This is the type of the data objects provided by the backend. One object contains the contact information of a person, containing fields like firstName, lastName, phone and so on.

ContainerUI

This is the main UI class for the test application used for this exercise. The class will build a Table using the provided data source and the container implementation you will create.

MyContainer

MyContainer is the container implementation you will complete. Note that this class extends AbstractContainer, which provides implementations for the methods from the ItemSetChangeNotifier interface. The super calls for these methods are provided for you. Additionally, the write methods have been made to throw exceptions since you're implementing a read-only container.

Things you'll need to do:

1. Implement the constructor of MyContainer class which creates a BeanItem based on the prototype object
2. Implement the methods needed for readonly Container.
   - Use the BeanItem to resolve property ids and types.
   - The size()-method should fire ItemSetChangeEvent if the size of DataProvider is changed
   - Use integers from 0 to size()-1 as ItemIds

Detailed instructions:

Step 1 - Implement the constructor

As you can see from the MyContainer's constructor declaration, it expects the DataProvider<T> class as well as a prototype object of type T as parameters. Implement the given constructor and store the parameters into fields in your container class. Do note that the prototype object should be wrapped into a BeanItem for easier access to its properties.

Step 2 - Implement the read-only methods

We'll start with the easiest ones, which are related to properties and will just be delegated to the prototype BeanItem instance which wraps the prototype object:

```
public Collection<?> getContainerPropertyIds()
public Class<?> getType(Object propertyId)
```

Next we will implement the size() method. This call can be delegated to the DataProvider, but you should also store the size in a field in your container. That way, when the size() method is called, you can check if the size is still the same as the new size provided by the DataProvider. If it is not, you should fire an ItemSetChangeEvent. Note: make sure to always call the size() method if size is needed in any other methods - read access to the stored size field should only be done from the size() method!

```
public int size()
```

The four remaining methods deal with fetching itemIds, items and properties. We will now implement these with the following specifications:

`public Collection<?> getItemIds()`
: Since we're using Integers as itemIds, you can simply return a List which contains integers from 0 to size() - 1.

`public boolean containsId(Object itemId)`
: Again, since we're using integers, this should be very simple too. Check that the provided parameter is an Integer and that it's valid considering the container size.

`public Item getItem(Object itemId)`
: For this method you should first use containsId to check that the item really is in the container. After that you can just fetch the requested item from the

DataProvider and return it. Remember to wrap the object into a BeanItem before returning it!

`public Property getContainerProperty(Object itemId, Object propertyId)`
Here we can use getItem to fetch the actual item, and then just use the standard item interface for getting the property.

Now you can run the application. If everything works as intended you should see a Table showing a whole lot of contact information. However, if you look at the console output of the server, you will see some debugging output from the DataProvider which shows that this is most likely not the most efficient way to implement a container. Don't worry, we'll get back to that in subsequent exercises.

Bonus - Implement Container.Hierarchical

DataProvider has information about contacts. There are team leaders and team members. In this bonus task, the container is modified such that each team leader is a root of the container and the team members are the children of the item representing the team leader.

Things you'll need to do:

1. Implement the Container.Hierarchical interface
   - Remember that this is a readonly container
   - Ask DataProvider about the parent/children relations

Detailed instructions:

The easiest ones to implement are the methods related to the parent/children information of an item. These methods can be directly delegated to the DataProvider.

```
public Collection<?> getChildren(Object itemId)
public Object getParent(Object itemId)
public boolean hasChildren(Object itemId)
```

In our example, all the items for which getParent(itemId) return null are root elements so the following methods can now be implemented.

```
public boolean areChildrenAllowed(Object itemId)
public boolean isRoot(Object itemId)
```

Since we are using integers as item id, you can go through integers from 0 to size() and check if the item is root or not. Note that the collections returned should be unmodifiable.

```
public Collection<?> rootItemIds()
```

Exercise 7: Adding indexing, ordering and sorting

The starting point of this exercise is the container you created in the previous exercise, so please open that project now.

Things you'll need to do:

1. Implement Container.Indexed interface
   - Throw an exception in methods which modifies containers content
   - Be careful to check that in each method container actually contains the given itemId or index

2. Implement Container.Sortable interface
   - DataProvider already have methods for sorting
   - Make sure that ItemSetChangeEvent is fired after each sort action

Detailed instructions:

Step 1 - Implement Container.Indexed

Using just the Container interface is quite inefficient for fetching the data in some cases. To mitigate this we will now make our container implement the Container.Indexed interface, which will also effectively make us implement the Container.Ordered interface.

Once you've made your container implement the Indexed interface, you should end up with 13 new method stubs. Out of these you can easily identify four methods which modify the container contents. You can make these throw an exception and you should end up with nine methods to implement.

Now since we're using Integers as itemIds, most of these methods become very easy or even trivial to implement. Note that for each of these methods you should consider always checking that the container actually contains the given itemId. We'll go through these now:

```
public int indexOfId(Object itemId)
```
return the parameter as Integer or -1 if it's not in the container

```
public Object getIdByIndex(int index)
```
return a new Integer based on the index or
throws IndexOutOfBoundsException if it's not found

```
public Object nextItemId(Object itemId)
public Object prevItemId(Object itemId)
```
return either itemId + 1 or itemId - 1, if the resulting value is valid, otherwise return null

```
public Object firstItemId()
```
return 0 if there are some items in the container, null otherwise

```
public Object lastItemId()
```
return size - 1 if there are some items in the container, null otherwise

`public boolean isFirstId(Object itemId)`
`public boolean isLastId(Object itemId)`
implement these in terms of the firstItemId() and lastItemId()

`public List<?> getItemIds(int startIndex, int numberOfItems)`
First do two checks: numberOfItems must not be negative and startIndex should be valid within the container size. If numberOfItems is below zero, then throw an IllegalArgumentException. If the startIndex is outside the container indexes (index < 0 or index >= size), then throw an IndexOutOfBoundsException. Then determine the end index, taking the container size into consideration. Finally you can build a list of Integers between the start and end index and return that.

At this point you can try out the application again and confirm that everything still works. Since we do not (yet) have caching, the amount of calls fired at the DataProvider should still be at the same level.

Step 2 - Implement Container.Sortable

To end this exercise we will add sorting support for the container. This should be relatively simple, since the actual sorting is done in the DataProvider - in a real world case this would happen e.g. in a database. So what we actually need to do is just pass the relevant methods to the DataProvider and make sure our container fires an ItemSetChangeEvent after each sort action.

Please note the following methods of the DataProvider interface:

`public List<Object> getSortableFields();`
returns a list of sortable field identifiers - these will be names of fields from the Contact object

`public void sort(Object[] fieldNames, boolean[] ascending);`
performs the sort based on the given field name(s) and sort direction(s)

To implement the sorting you just need to make your container implement the Container.Sortable interface. After this you will get two methods that you need to implement. You should relay the calls to these methods to the DataProvider. Note that you also need to fire an ItemSetChangeEvent when the sort() method is called in order to make the Table realize that the contents have changed.

Finally try running your application again. You should now be able to sort the Table contents by clicking on the column headers.

Exercise 8: Implementing lazy-loading

If we imagine that the backend data source would contain a huge amount of objects, it would not be very efficient to load them all at once into memory within the container implementation. After all, most probably a single user is not going to interact with all the objects at once.

Another consideration is that we probably don't want to hit the backend with constant size checks and object fetching. Instead, we could agree that the objects and size are valid for either a limited time or until a manual refresh is commanded. In this exercise we'll use timeouts for size and object validity - 5 seconds and 30 seconds respectively.

Things you'll need to do:

1. Extend AbstractCachedContainer which is provided for this exercise
   - You can leave the new methods empty for now
2. Modify the getItem and size methods to use methods in the cached implementation instead of methods in DataProvider
3. Implement the abstract methods getCount and fetchItemBatch
   - Each batch should have 50 items starting from index which is a multiple of batch size (0, 50, 100, …)

Detailed instructions:

Step 1 - Switch to AbstractCachedContainer

As noted before, caching is crucial for optimizing the amount and frequency of calls to the backend. For this purpose we have provided you with a cached version of the AbstractContainer. You can take this class into use by making MyContainer extend AbstractCachedContainer<T> instead of AbstractContainer. The cache implementation used within is EhCache, but any caching mechanisms may be used.

After you've changed the base class, you need to make MyContainer implement two abstract methods introduced by the new super class. You can leave these methods empty for now.

Step 2 - Modify the getItem and size methods

First we want to modify two methods in a very simple way to ensure that both the size and the items are fetched through the cache implementation:

`public Item getItem(Object itemId)`
: Instead of getting the item from the DataProvider and wrapping it, you can simple call the getItem -method of the super class and return the result.

`public int size()`
: Similarly to the getItem -method, replace the call to DataProvider with a call to the size -method of the super class.

Step 3 - Implement the two abstract methods

The final step is to provide the cache implementation with the items and size it needs to cache. For this we'll implement the two methods introduced by the AbstractCachedContainer using the following specifications (for detailed contract, see the JavaDoc descriptions of the abstract methods):

`protected int getCount()`
: We'll start with the easier one. For the item count you can simply fetch and return the count directly from the DataProvider.

`protected Map<Integer, Item> fetchItemBatch(int requestedIndex)`
: First of all, you should select a batch size for fetching the data. In this exercise we should use **50** for the **batch size**. We will only fetch data from the DataProvider in batches of 50 objects and each batch should begin at a multiple of the batch size (e.g. 0, 50, 100, …).

  The method itself should implement the following logic:
  1. Determine the correct start index for the batch. For example, if requestedIndex = 73, the start index should be 50
  2. Fetch a list of objects from the DataProvider using the start index you calculated
  3. Create a result map
  4. Insert the objects to the map, using their indexes as keys
  5. Return the map

Finally, we're done! You can now try running the application. Scroll around in the Table a bit and look at the console output by the DataProvider. Also note that there is additional output about the caching from the cache implementation. If everything is implemented properly, you should see batches of objects fetched as you scroll - and also the occasional size update. The DataProvider will output the number of returned items as well as their indexes to the log output to help you notice any problems in your code.

The end result is a lazy-loading, sortable container that you can use with any backend system by implementing just four methods of the DataProvider interface yourself.

This type of container is in use in many real world projects, but please note that error handling is omitted in this exercise. In a real-world application you'll want to check that the DataProvider returns the same amount of objects it was asked for, handle any backend exceptions and so on.

Bonus - Implement Collapsible interface

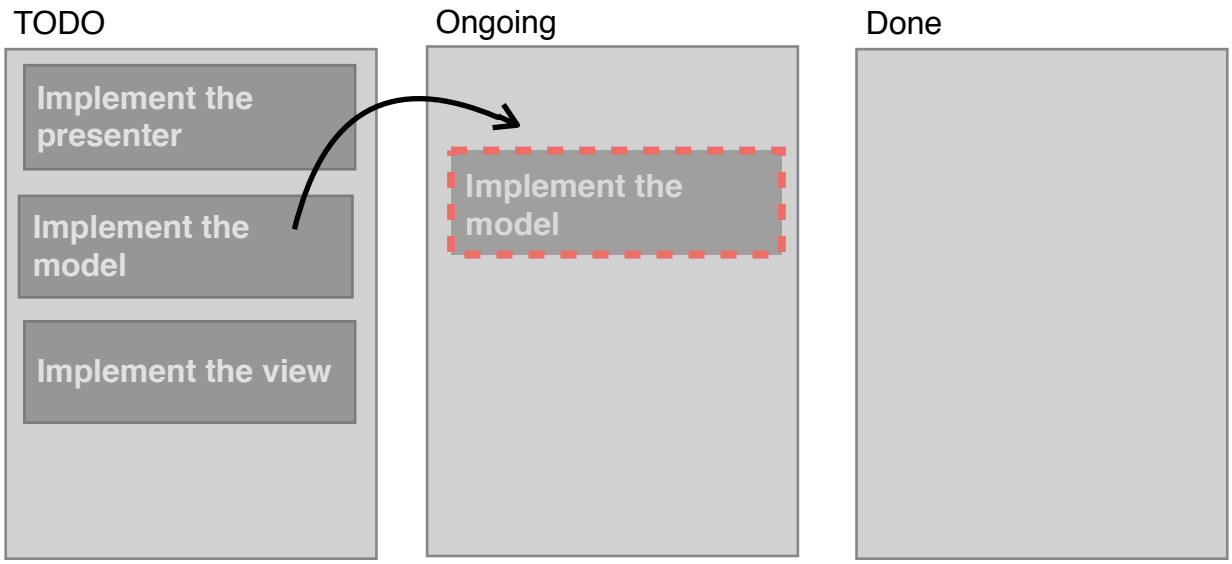
If you have done the bonus task to implement hierarchical container, you notice that all the children are also loaded from the DataProvider even though they are not visible. To avoid loading these unnecessary items, you could implement Collapsible interface, which notifies when an item is collapsed or expanded.

Things you'll need to do:

1. Implement the Collapsible interface
   - The children of an collapsed item should not be part of the container
   - If the item is collapsed or expanded the amount of item is changed so the ItemSetChange event should be fired
   - DataProvider has information about the collapsed items but you should tell it to hide collapsed items (size() and fetchItemBatch() does not take these into account)

Exercise 9: Drag and Drop

In this exercise we are going to practice the usage of drag and drop functionality in Vaadin. Your task is to create a Kanban board. The application should consists of three horizontally places panels, TODO, Ongoing and Done. The panels can contain tasks. The users should be able to move a task from one panel to another by dragging the task and releasing it over the new panel.



Things you'll need to do

1. Wrap the panels in a DragAndDropWrapper
2. Implement a DropHandler that handles the moving of a component from one panel to another. The criterion used should be AcceptAll.get()
3. Wrap each task in a DragAndDropWrapper
4. Set a drag mode for the tasks

Bonus task: Implement a server-side criterion that checks that no tasks are dragged directly from TODO on Done. All tasks should first go through Ongoing before being moved to Done.