Application of Context Programming and Data Binding

SAP NetWeaver 04

Copyright

Icons in Body Text

Icon	Meaning
	Caution
	Example
	Note
	Recommendation
	Syntax

Additional icons are used in SAP Library documentation to help you identify different types of information at a glance. For more information, see *Help on Help* → *General Information Classes and Information Classes for Business Information Warehouse* on the first page of any version of *SAP Library*.

Typographic Conventions

Type Style	Description
Example text	Words or characters quoted from the screen. These include field names, screen titles, pushbuttons labels, menu names, menu paths, and menu options. Cross-references to other documentation.
Example text	Emphasized words or phrases in body text, graphic titles, and table titles.
EXAMPLE TEXT	Technical names of system objects. These include report names, program names, transaction codes, table names, and key concepts of a programming language when they are surrounded by body text, for example, SELECT and INCLUDE.
`Example text`	Output on the screen. This includes file and directory names and their paths, messages, names of variables and parameters, source text, and names of installation, upgrade and database tools.
`Example text`	Exact user entry. These are words or characters that you enter in the system exactly as they appear in the documentation.
`<Example text>`	Variable user entry. Angle brackets indicate that you replace these words and characters with appropriate entries to make entries in the system.
`EXAMPLE TEXT`	Keys on the keyboard, for example, `F2` or `ENTER`.

Application of Context Programming and Data Binding

Purpose

You will use the example of sales orders to show you how to display business data in the view of a Web Dynpro application in two tables (for customer data and the corresponding order data). For this purpose, you should be familiar with fundamental parts of the Web Dynpro runtime environment like controller context, data binding, and the Web Dynpro programming model.

Features

You will learn how to:

- Declare an extended context structure, including a subnode. In this context structure, the data sets of multiple customers and their orders are stored at runtime
- Insert a subnode of the type *non-singleton* in the context structure
- Fill tables with data sets that are stored in the controller context using data binding
- Implement a *supply function* to fill a subnode with node elements

Process Flow

The development process involves the following:

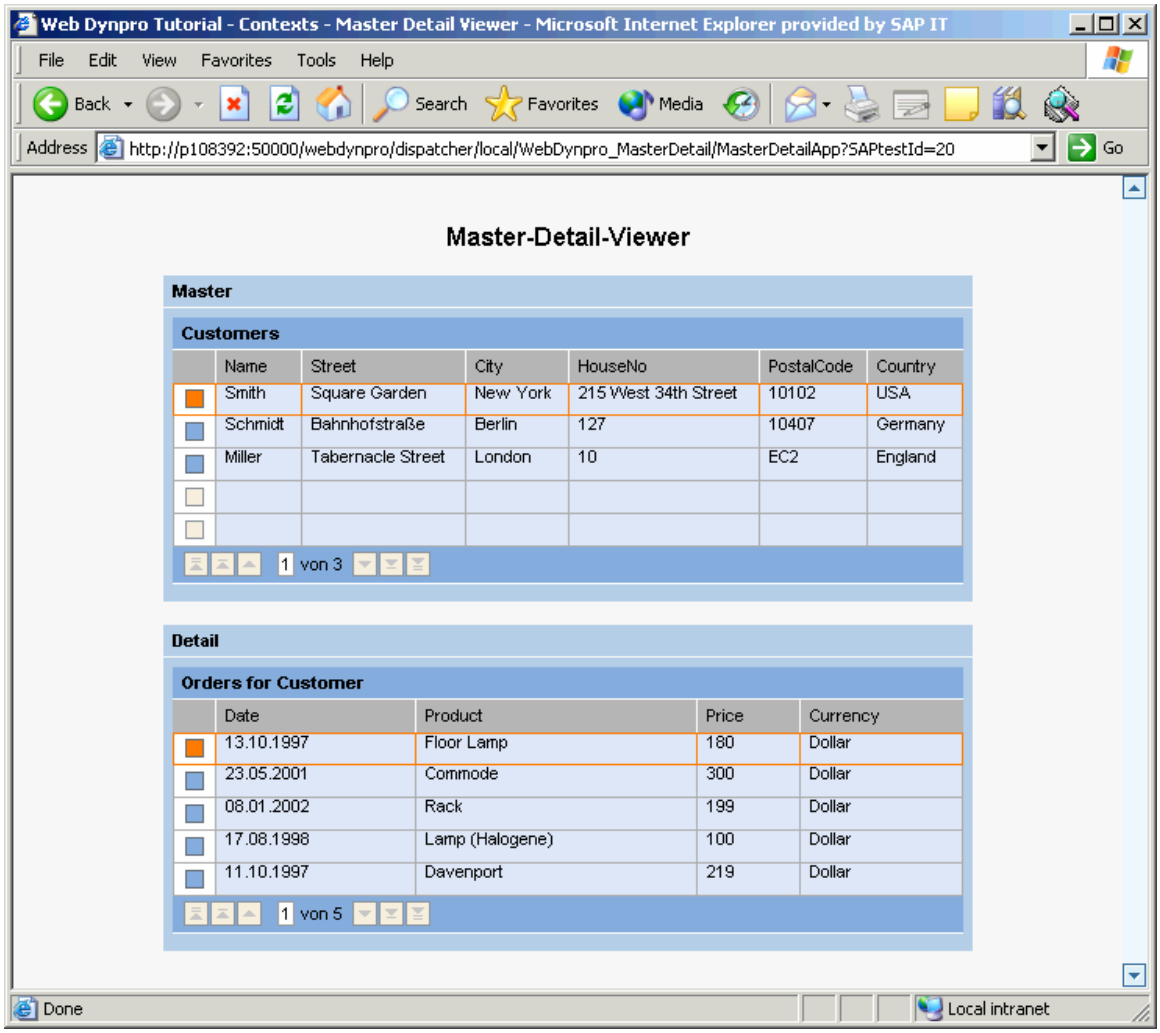
- Description of the master detail viewer application
- Specification of the project structure
- Context design in the master detail viewer
- Layout design of a view
- Data binding between tables and controller context of a View
- Controller implementation of a view
    - Demo business object layer
    - Context initialization
    - Implementation of a supply function

First, get to know the application scenario that is to be used in the master detail viewer [page 5].

The Master/Detail Viewer Application

The Master/detail viewer Application shows the following application scenario. The business data for several customers is displayed in two tables in a single view. The upper *Master* table shows the customers' names and addresses. The lower *Detail* table displays the order data for the customer that is currently selected.

If the user chooses a different customer in the upper table, the second table automatically displays the correct order data for that customer.




You can now start developing the master-detail viewer in your SAP NetWeaver Developer Studio with the help of the Web Dynpro tools [page 6]!

Specifying the Project Structure

Procedure

First define the following project structure for the master-detail sample application:

1. **Web Dynpro project:**

   Name: **`WebDynpro_MasterDetail`**

2. **Web Dynpro component:**

   Name: **`MDViewer`**

   Package: **`com.sap.tc.webdynpro.tutorials.masterdetail`**

3. **Web Dynpro view:**

   Name: **`Work`**

In Web Dynpro component: **`MDViewer`**

In Window: **`MDViewer`**

4. **Web Dynpro application:**

   Name: **`MasterDetailApp`**

   Package: **`com.sap.tc.webdynpro.tutorials.masterdetail`**

   Web Dynpro component: **`MDViewer`**

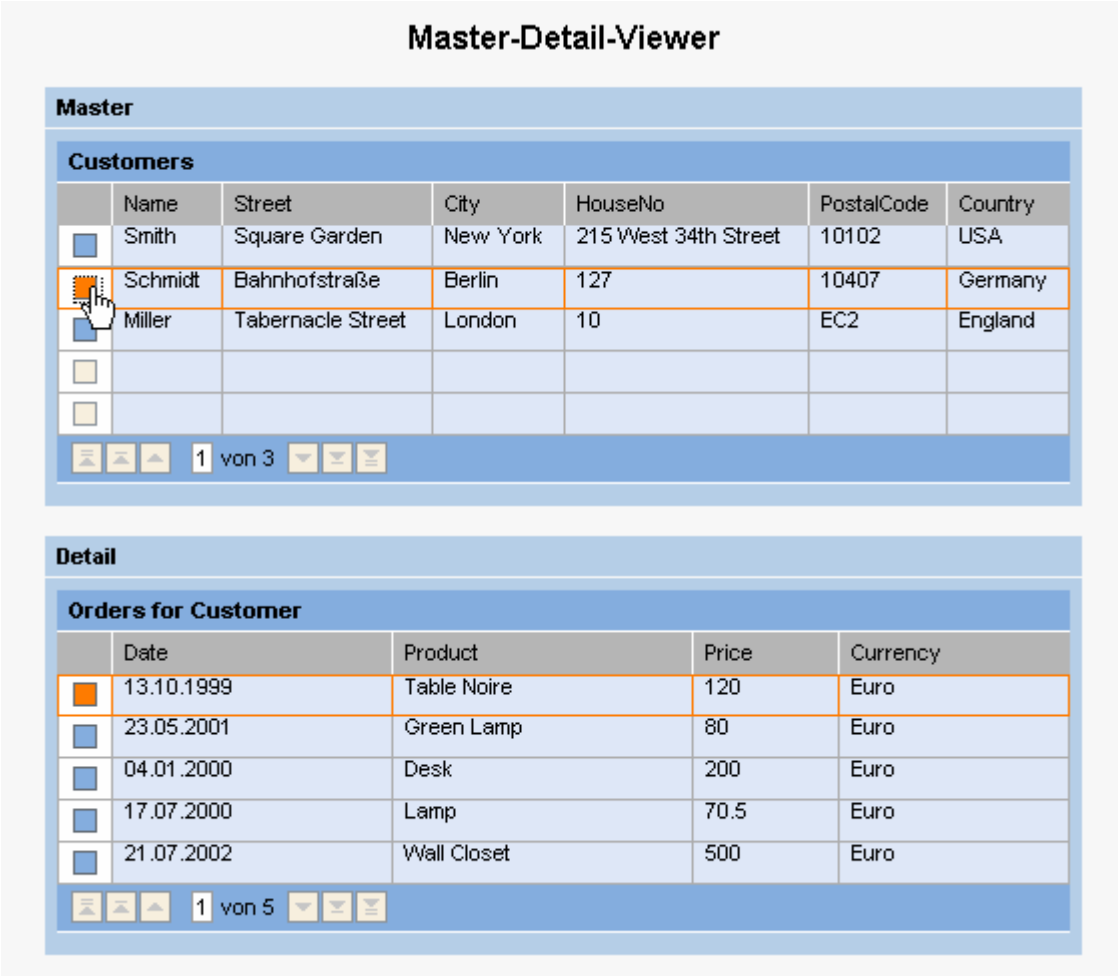
   Interface View: *MDViewerInterfaceView*

   Startup plug: *Default*

Now proceed to the Context Design in the Master-Detail Viewer [page 7].

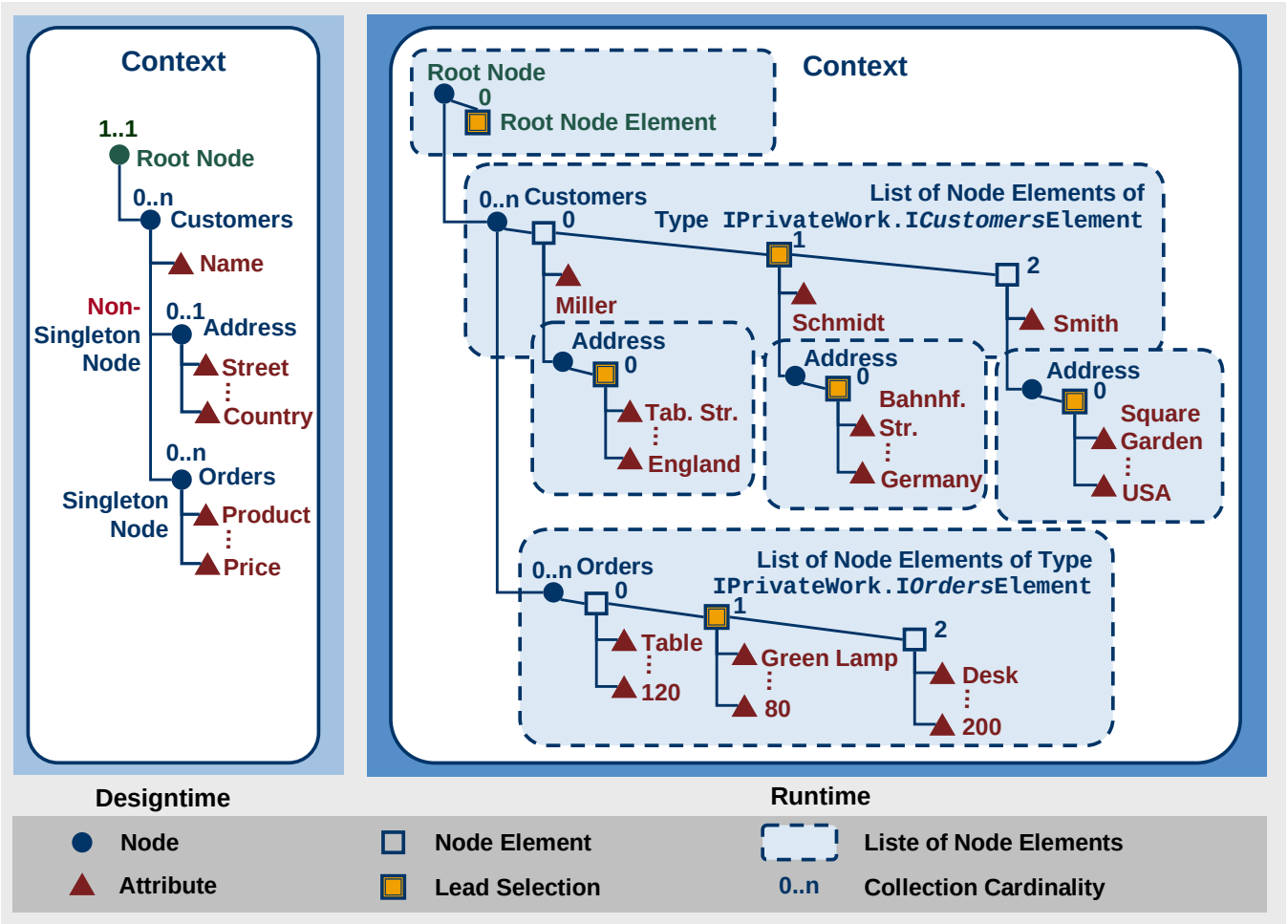
Context Design in the Master/Detail Viewer

The two tables in the Master-Detail Viewer will be filled with data records that are saved in the context of the view controller. The (upper) *Master* table displays a row for each customer, containing his or her name and address. The (lower) *Detail* table displays the order records for the currently selected customer. We identify each single record with the purchase date, product name, product price, and currency name.



How do we declare the records displayed in the two tables *Customers* and *Orders for Customers* in the controller context of a view at design time?

To clarify this, consider the following graphic:



The controller context of a view consists of an independent value node called *Customers*, with the cardinality (0-*n*), since several customer records are to be shown in the table. This value node contains a single value attribute called *Name* and two dependent value nodes called *Address* and *Orders*.

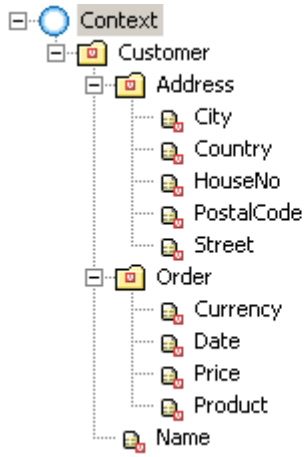
The value node *Address* has a cardinality from 0-1, since a single address record (corresponding to a single node element of type *Address*) is assigned to each customer. To display the addresses for all customers simultaneously at runtime, this inner value node must be declared as *Non-singleton node*. Otherwise, the Web Dynpro runtime environment will only have a single value node instance, in which the node element with the type *Address* for the currently selected customer is saved. To be able also to display the addresses for the other customers, the Web Dynpro runtime environment must create instances of several separate *Address* value nodes.

This is not the case for the inner value node called *Orders*. At runtime, you need only a single value node, which is filled with the order data records associated with the currently selected customer – that is, with the data in the *Orders node elements*.

Declare the context structure in the *Context* view of the *Work* view. (For information on how to do this, refer to Declaring a Controller Context of a View [page 10].)

Declaring the Controller Context of a View

In the *View Designer*, choose the *Context* tab and declare the following context structure in it:



Make the following settings for the properties of each context element:

Context element	Type	Attributes	Value
Customer	Value node	`cardinality`	**0..n**
		`singleton`	**true**
Name	Value attribute	`type`	string
Address	Value node	`cardinality`	**0..1**
		`singleton`	**false**
Street	Value attribute	`type`	string
City	Value attribute	`type`	string
HouseNo	Value attribute	`type`	string
PostalCode	Value attribute	`type`	string
Country	Value attribute	`type`	string
Order	Value node	`cardinality`	**0..n**
		`singleton`	**true**
		`supplyFunction`	
		We will add a *supply function* called `supplyOrdersForCustomer` later (refer to Adding a Supply Function [page 24]).	
Date	Value attribute	`type`	`string`
Product	Value attribute	`type`	`string`

Price	Value attribute	`type`	`string`
Currency	Value attribute	`type`	`string`

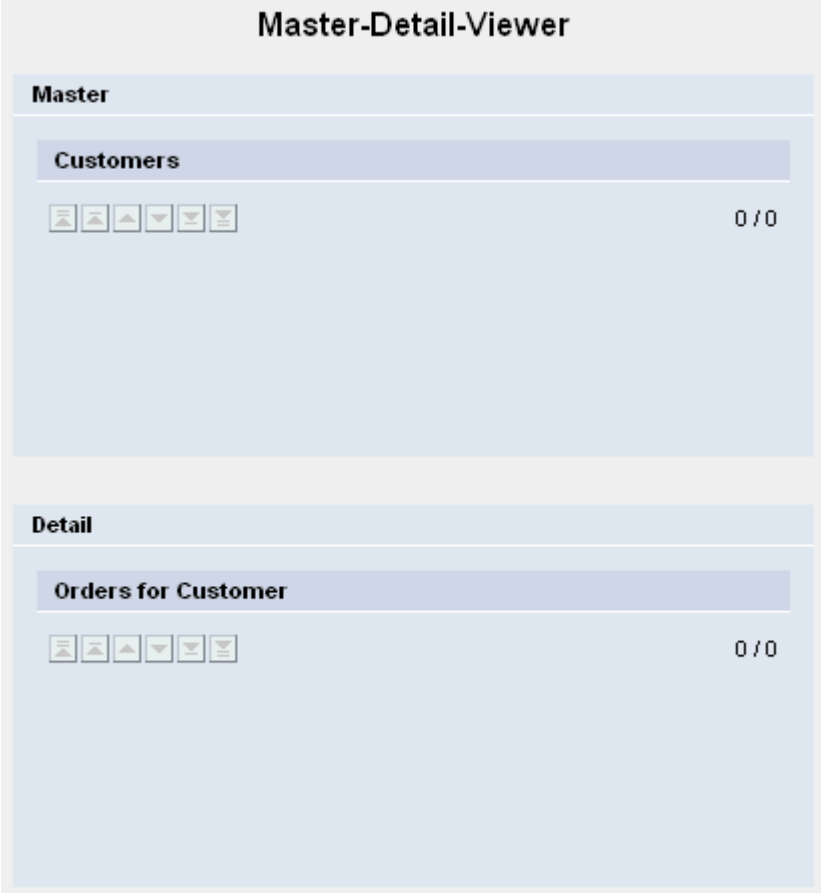
Remember that the **`singleton`** attribute for the context value node *Address* will be assigned the value **`false`**.

After you have declared this context structure, you can continue by Designing the View Layout [page 11]. You can then define the required data bindings between the UI elements of a view and the context elements.

Designing a View Layout

The view layout consists of a header, a *Master* table for *Customer* information, and a *Detail* table for the corresponding *Order* data.

In the View Designer of the Web Dynpro tools, the view layout looks like this:



Implement the following UI element structure in the **`Work`** view:

UI Element Type	UI Element Name	Embedded in ... (Container Name)	UI Element Property	Value
Transparent Container	RootUIElement-Container		*Properties of Transparent Container - layout*	***GridLayout***
			Layout - colCount	1
TextView	HeaderText	RootUIElementContainer	*Properties of TextView – design*	***header1***
			Properties of TextView – text	Master/Detail Viewer
			LayoutData – hAlign	***center***
			LayoutData – paddingBottom	***large***
			LayoutData – paddingTop	***large***
Group	MasterGroup	RootUIElementContainer	*Properties of Group – width*	75%
			LayoutData – hAlign	***center***
Caption	MasterGroup_ Header	MasterGroup	*Properties of Caption - text*	Master
Table	CustomerTable	MasterGroup	 The table is separately filled with columns using data binding	
			Properties of Table – width	100%
Caption	Header_1	CustomerTable	*Properties of Caption - text*	Customers
	 To add the table header for the *CustomerTable* table to the node  *CustomerTable[Table]*, choose *Insert Header* in the context menu.			
Group	DetailGroup	RootUIElementContainer	*Properties of Group – width*	75%
			LayoutData – hAlign	***center***
			LayoutData – paddingTop	***large***
Caption	DetailGroup_ Header	DetailGroup	*Properties of Caption - text*	Detail
Table	OrderTable	DetailGroup	 The table is separately filled with columns using data binding	
			Properties of Table - width	100%
Caption	Header_2	OrderTable	*Properties of Caption - text*	Orders for Customer

As you can see, neither table has columns yet. You can easily add table columns using data binding. SAP NetWeaver Developer Studio provides the appropriate tools.

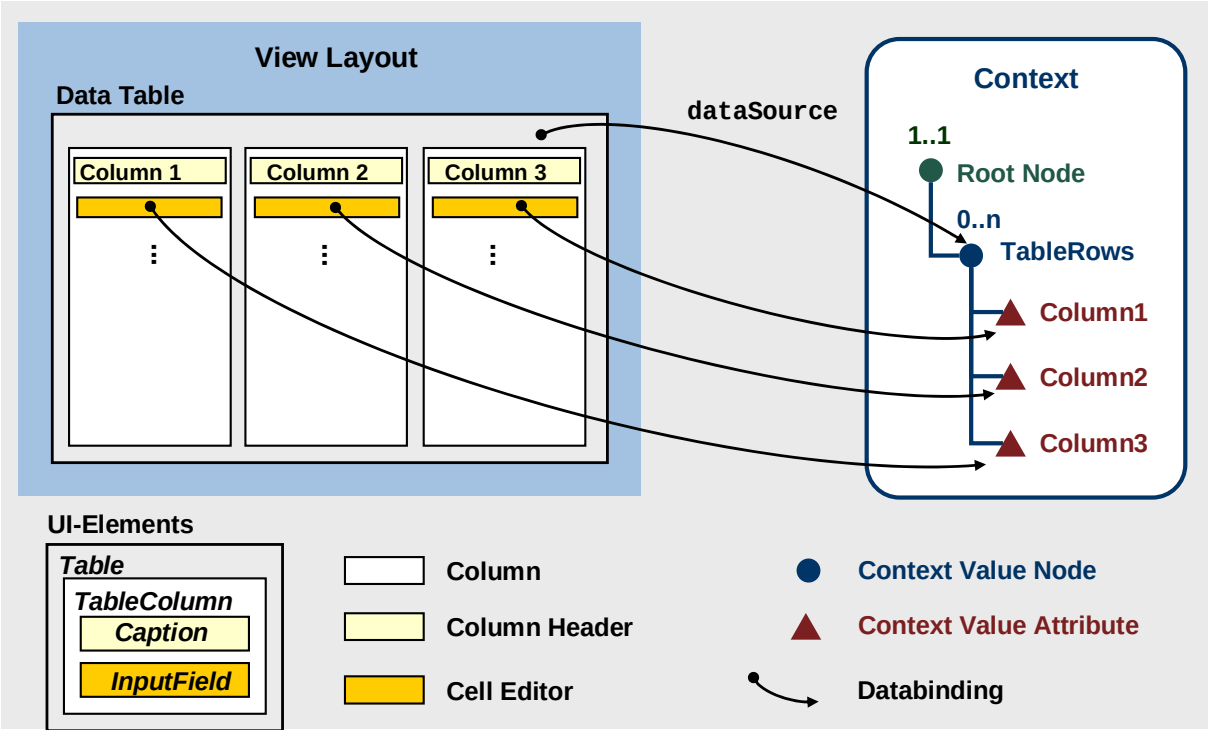
The following step describes how to add new table columns to the controller context of a view using data binding [page 13] .

Binding Tables to the Controller Context of a View

At the top level, the **`dataSource`** property of the table UI element must be bound to a value node with a cardinality of 0…*n* or 1…*n*, which at runtime represents a list of node elements. Each of these node elements corresponds to a single row in the table. The selected table rows are specified using node selection (subsets of node elements). The currently selected row corresponds specially to the lead selection of the value node.

The table columns are specified using the *TableColumn* UI element as a subnode of the *Table* UI element. This UI element specifies the number, sequence, header cells, and width of each table column.

The table content is specified for each column using the associated cell editor (for example, using an *InputField* or *TextView* UI element). Note that this cell editor does not only enable cell editing. You can also use the cell editor to trigger events (*Button)* or display texts or images (*TextView*, *Image*).



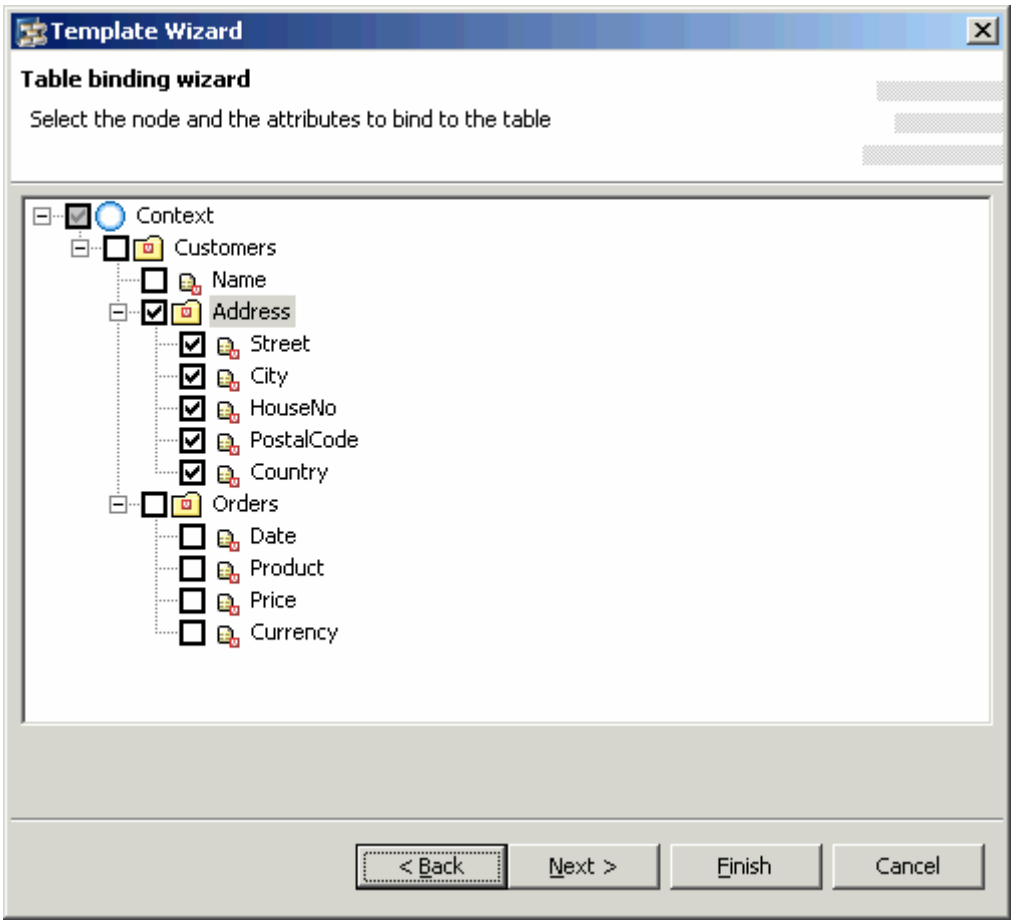
The Web Dynpro tools enable you to add table columns with a context reference (for example, using the UI element *TableColumn – Caption – InputField*) by providing wizards that automatically create the necessary data binding between UI element properties and context elements. This includes, for example, data binding between the property **`text`** in the *InputField* UI element of a column cell and the value attribute to be displayed in it.

Procedure

Customer Table

1. In the context menu for the UI element *CustomerTable[Table]*, choose *Create Binding*. To display the data structures stored in the controller context at runtime in the two tables *Customers* and *Orders for Customers*, you now use the concept for data binding.
2. In the dialog box that appears, select the *Address* context node.

   You add columns by selecting the *Value* node that contains the value attributes to be displayed. In this case, this is the dependent value node *Address*.



3. Choose *Next >*.

   In this dialog box, you can define how the table data is displayed in the *Editor* column. In the example application, you only want to display the table data, not edit it. Therefore, each column requires the UI element *TextView* as the cell editor (*Editor* column). This is already entered as the default value and you do not have to define it any further.

4. Choose *Finish*.

   In the *Outline* display, the *CustomerTable* UI element is enhanced by the corresponding columns. Each column contains two UI elements for displaying the column header (*Caption)* and the column cell *(TextView*).

Add the column for the customer name as follows:

5. In the context menu for the UI element *CustomerTable[Table]*, choose *Insert Column*. The *TableColumn* UI element is named automatically.
6. In the *Outline* perspective view, select the newly created *TableColumn* UI element and enter the value **`CustomerTable_Name`** in the perspective view *Properties* for the *id* attribute.
7. In the same way, rename the *Caption* UI element belonging to the new table column as **`CustomerTable_Name_Header`**.
8. Enter the value **`Name`** in the **`text`** property of the *CustomerTable_Name_header[Caption]* UI element.
9. In the *Outline* perspective view, use *Drag&Drop* to move the added *TableColumn* UI element to the uppermost position in the *CustomerTable[Table]*.
10. In the context menu for the UI element *CustomerTable_Name[TableColumn]*, choose *Insert Celleditor*.
11. Give the UI element the *Id* **`CustomerTable_Name_Editor`** and choose the type *TextView* from the dropdown list.
12. Bind the **`text`** property of the UI element *CustomerTable_Name_Editor[TextView]* to the value attribute `Customer.Name`.

You must now ensure that the **`dataSource`** property of the UI element *CustomerTable[Table]* is bound to the *Customers* value node, since the value attributes to be displayed are located in both the *Address* value node and in the superordinate *Customers* value node.

13. In the *Outline* perspective view, choose the UI element *CustomerTable[Table].*
14. Bind the **`dataSource`** property to the *Customers* value node .

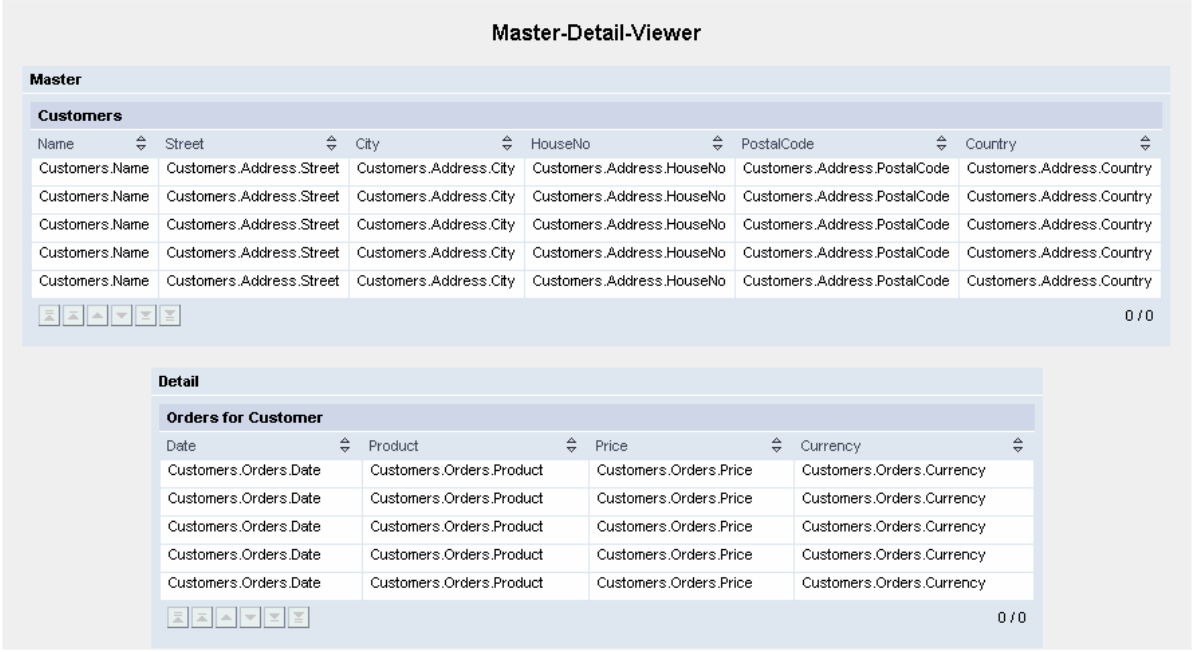
Order Table

1. In the context menu for the UI element *OrderTable[Table]*, choose *Create Binding*.
2. In the dialog box that appears, choose the *Orders* context value node and choose *`Finish`* to confirm.

   Choose *Finish*. The *Order* table is filled with the UI elements for four new columns.

Result

You have now bound the two tables *CustomerTable* and *OrderTable* to the controller context of the view. In the View Designer, the view looks like this:



Go! Now proceed to the technical program aspects in Implementing the Controller Context of a View [page 16].

Implementing the Controller Context of a View

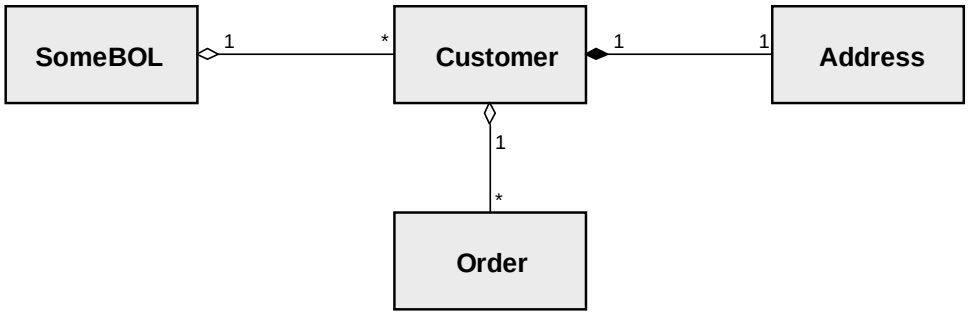
To implement the controller context of a view, you must work through the following three topics consecutively:

- **Demo business object layer (demo BOL)**: First, implement a class called *SomeBOL.java*. At runtime this class supplies data for the table rows to be displayed in *CustomerTable* and *OrderTable.*
- **Initializing context**: You initialize the declared context structure in the view controller method **`wdDoInit()`**. The main purpose of this method is to fill the context value nodes with instances of node elements of the appropriate type.
- **Supply function**: You implement a supply function, which is called from the Web Dynpro environment and fills the inner value node *Orders* with the node elements for the selected customer. To trigger communication with the runtime environment, you must bind an action to the property *LeadSelect* of the *Order* table.

Go! Save the class SomeBOL.java, in which you will implement a minimal business object layer used to represent the *Customer* and *Order* data. For more information on how to do this, refer to Implementing a Demo BOL [page 17].

Implementing a Demo BOL

To fill the two tables *CustomerTable* and *OrderTable* with example data at runtime, you simply implement individual auxiliary classes as a representation of a demo business object layers, in accordance with the following graphic:



After you instantiate and initialize the class *SomeBOL*, the records to be displayed in the customer and order tables can be accessed in the view controller using the methods **`getCustomers()`** and **`getOrdersForCustomer(String customerName)`**.

Procedure

1. In the directory `src/packages/<component-package-folder>` (that is, `src/packages/com.sap.tc.webdynpro.tutorials.masterdetail` in our example), add a new subdirectory called `bol`.
2. Save the following Java class, called *SomeBOL.java*, in this subdirectory. The classes *Customer*, *Address* and *Order* are included in this class as inner classes.

The class *SomeBOL.java*

```
/*
* SAP Copyright (c) 2003
* All rights reserved.
*/
package com.sap.tc.webdynpro.tutorials.masterdetail.bol;

import java.util.*;

/**
 * Demo Business Object Layer class used in Web Dynpro Tutorial Master-Detail-
Viewer
 */
public class SomeBOL {

  // ##### Inner Class for Business Objects of Type Customer ###############
  public class Customer {
    // ---- fields
    private String name;
    private Address address;
    private List orders;

    /** Constructors for Customer */
    public Customer(String name, Address address, List orders) {
      this.name = name;
      this.address = address;
      this.orders = orders;
    }
```

```
    // ====================== Getter Methods ========================
    public Address getAddress() {
      return address;
    }

    public String getName() {
      return name;
    }

    public List getOrders() {
      return orders;
    }
  }

  // ####### Inner Class for Business Objects of Type Address #######
  public class Address {
    // ---- fields
    private String street;
    private String houseNo;
    private String city;
    private String postalCode;
    private String country;

    /** Constructor for Address */
    public Address(
      // ---- fields
      String street,
      String houseNo,
      String city,
      String postalCode,
      String country) {
      this.street = street;
      this.houseNo = houseNo;
      this.city = city;
      this.postalCode = postalCode;
      this.country = country;
    }

    // ====================== Accessor-Methods ========================
    public String getCity() {
      return city;
    }

    public String getCountry() {
      return country;
    }

    public String getHouseNo() {
      return houseNo;
    }

    public String getPostalCode() {
      return postalCode;
    }

    public String getStreet() {
      return street;
    }
  }

  // ####### Inner Class for Business Objects of Type Order ########
  public class Order {
    // ---- fields
    private String date;
    private String product;
    private String price;
    private String currency;
```

```
    /** Constructor for Order */
    public Order(String date, String product, String price, String currency) {
      this.date = date;
      this.currency = currency;
      this.price = price;
      this.product = product;
    }

    // ====================== Getter Methods ========================
    public String getDate() {
      return date;
    }

    public String getPrice() {
      return price;
    }

    public String getProduct() {
      return product;
    }

    public String getCurrency() {
      return currency;
    }
  }

  // ------------------------ end of inner classes ---------------------------

  private Map customers;

  /** Constructor for SomeBOL */
  public SomeBOL() {
    this.customers = Collections.EMPTY_MAP;
  }

  public void initialize() {
    List orderList;
    customers = new HashMap();

    orderList = new ArrayList();
    orderList.add(new Order("12.10.2001", "Table", "120", "Pound"));
    orderList.add(new Order("23.05.2001", "Chair", "80", "Pound"));
    orderList.add(new Order("04.01.2000", "Desk", "200", "Pound"));
    orderList.add(new Order("17.07.2000", "Lamp", "70.5", "Pound"));
    customers.put(
       "Miller",
       new Customer(
        "Miller",
        new Address("Tabernacle Street", "10", "London", "EC2", "England"),
        orderList));

    orderList = new ArrayList();
    orderList.add(new Order("13.10.1999", "Table Noire", "120", "Euro"));
    orderList.add(new Order("23.05.2001", "Green Lamp", "80", "Euro"));
    orderList.add(new Order("04.01.2000", "Desk", "200", "Euro"));
    orderList.add(new Order("17.07.2000", "Lamp", "70.5", "Euro"));
    orderList.add(new Order("21.07.2002", "Wall Closet", "500", "Euro"));
    customers.put(
      "Schmidt",
      new Customer(
        "Schmidt",
        new Address("Bahnhofstraße", "127", "Berlin", "10407", "Germany"),
        orderList));

    orderList = new ArrayList();
    orderList.add(new Order("13.10.1997", "Floor Lamp", "180", "Dollar"));
    orderList.add(new Order("23.05.2001", "Commode", "300", "Dollar"));
    orderList.add(new Order("08.01.2002", "Rack", "199", "Dollar"));
```

```
    orderList.add(new Order("17.08.1998", "Lamp (Halogene)", "100", "Dollar"));
    orderList.add(new Order("11.10.1997", "Davenport", "219", "Dollar"));
    customers.put(
      "Smith",
      new Customer(
        "Smith",
        new Address(
          "Square Garden",
          "215 West 34th Street",
          "New York",
          "10102",
          "USA"),
        orderList));
  }

  public Collection getCustomers() {
    return customers.values();
  }

  public Collection getOrdersForCustomer(String customerName) {
     return ((Customer)customers.get(customerName)).getOrders();
  }
}
```

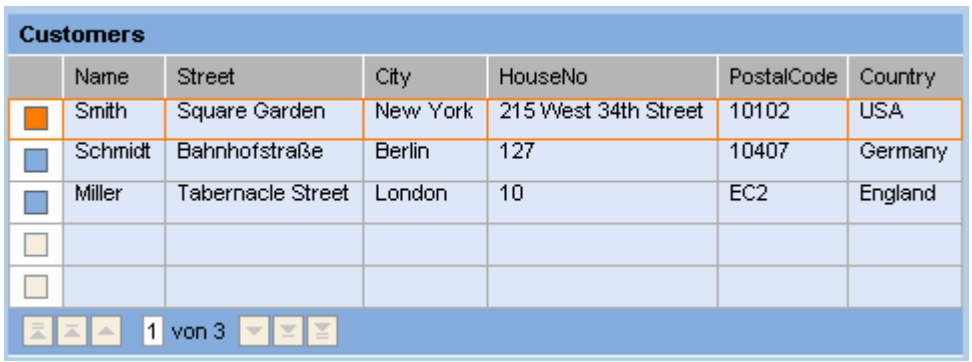
In the next topic, we will use an object of the class *SomeBOL* to fill the context structure in the view controller with data.

You can now initialize the context structure [page 20] in the controller implementation of the view using the class *SomeBOL*.

Initializing the Controller Context of a View

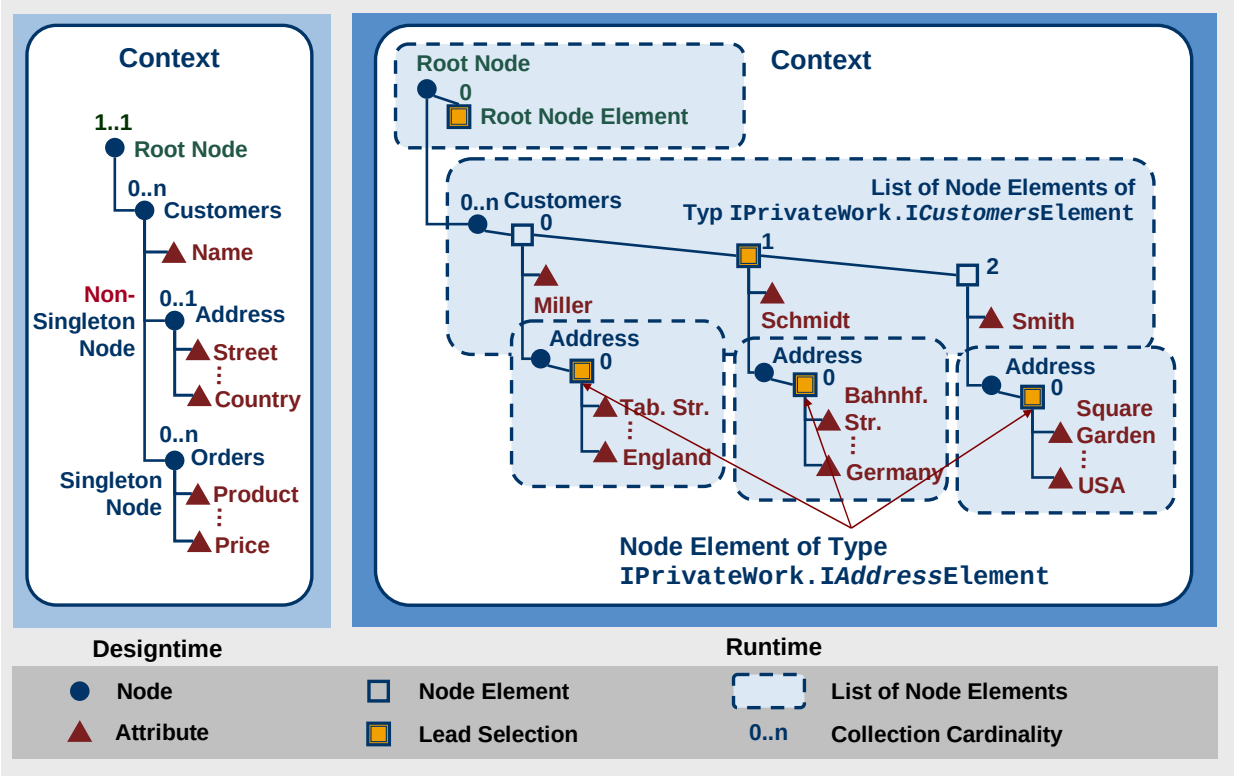
Initializing the declared controller context of the view is the crux of this example application, in a technical programming sense.

In this section, you will learn how to fill single value nodes with associated node elements at runtime using generated context interface methods, to display these using data binding in the two tables *CustomerTable* and *OrderTable*. The master table *CustomerTable* will then look like this in the Web browser:



In the above graphic, the **lead selection** of the first node element of the value node *Customers* is indicated by the fact that the first row in the table is highlighted.

Procedure



The graphic shows which context data structure must be available for filling the *Customer* table using data binding.

As you can see, several address data records (node element of type *IAddressElement*) must be instantiated at the same time, for you to be able to display them in all rows of the *Customer* table (and thus separately for each customer). This is why the value node *Address* is declared as a non-singleton node.

The value node *Customers* is filled step-by-step:

- **Instantiating node elements**: Repeated instantiation of a node element of the type *ICustomersElement*.
- **Defining value attributes**: Assigning values for the value attributes declared in the node element of the type *ICustomersElement* – in this case for the *Name* attribute.
- **Inserting node elements**: Inserting the node elements of the type *ICustomersElement* in the value node *Customers*. “Inserting” means adding the elements to the list of node elements that have been aggregated from the value node *Customers*.

A similar procedure is then carried out with each node element of the type *IAddressElement*. Each of these elements is inserted in the appropriated *Address* value node, which has been instantiated separately for each node element.

Note that each individual node element of the type *ICustomersElement* has its own *Address* value node instance only because the value node *Address* was declared as a non-singleton node at design time. Otherwise, you would only be able to access directly the one instance of *Address* associated with the current lead selection of the *Customer* value node, in the controller context, by calling the method **`wdContext.nodeAddress()`**.

Now implement the **`wdDoInit()`** method of the *Work* view controller. When doing so, make sure you add the `import` statements and the declare the instance variable *someBOL*.

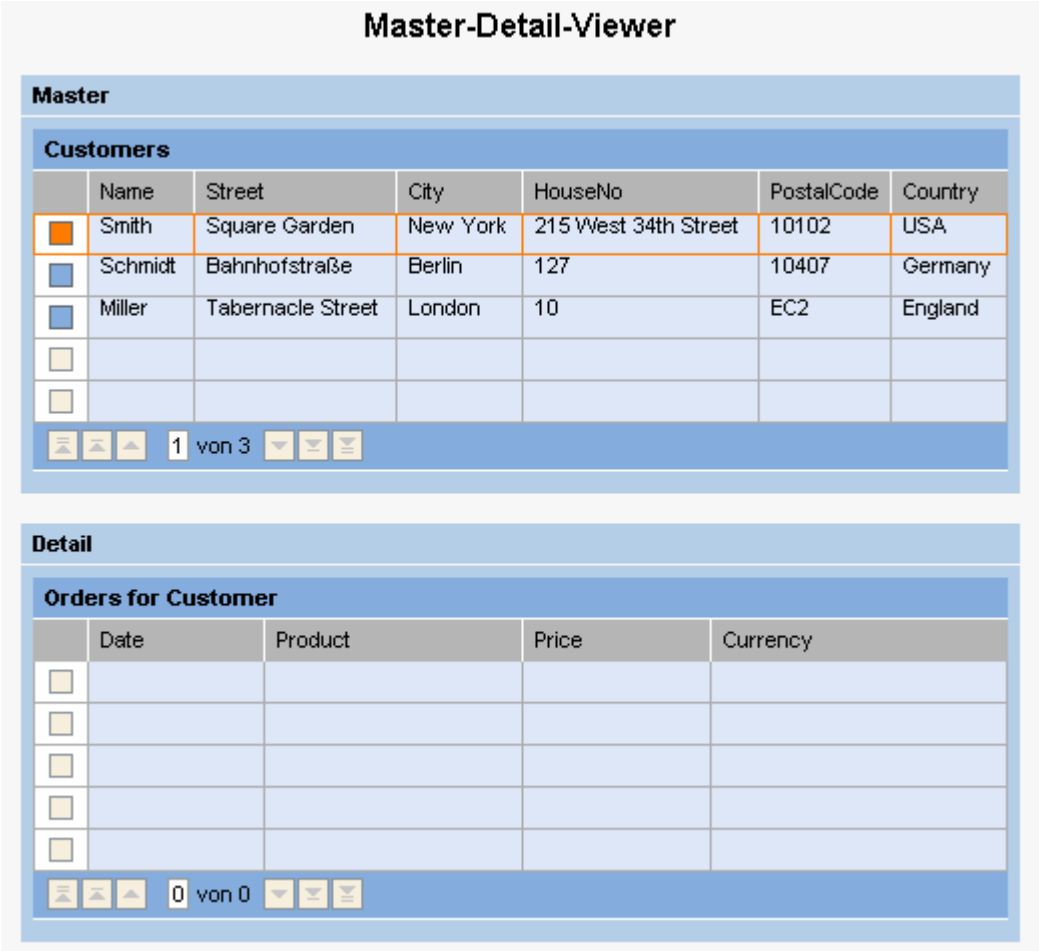
Implementation of the `wdDoInit()` method in the *Work* view controller

```
...
//@@begin imports
import java.util.*;
import com.sap.tc.webdynpro.tutorials.masterdetail.bol.SomeBOL;
import com.sap.tc.webdynpro.tutorials.masterdetail.wdp.IPrivateWork;
//@@end
...
/** Hook method called to initialize controller. */
public void wdDoInit()
{
  //@@begin wdDoInit()
  SomeBOL.Customer customer;
  SomeBOL.Address address;
  IPrivateWork.ICustomersElement newCustomerNodeElement;
  IPrivateWork.IAddressElement newAddressNodeElement;

  someBOL.initialize();
  Collection customers = someBOL.getCustomers();

  // ======== Populate context ==================
  for (Iterator iter = customers.iterator(); iter.hasNext();){
    customer = (SomeBOL.Customer)iter.next();
    address = customer.getAddress();
    // Instantiate new value node element of type ICustomersElement
    // and define attributes
    newCustomerNodeElement = wdContext.createCustomersElement();
    newCustomerNodeElement.setName(customer.getName());
    // The creation of a new inner node element instance requires
    // the existence of a parent node element already bound to the
    // parent node. So call method bind() (or alternatively addElement()) before.
    wdContext.nodeCustomers().addElement(newCustomerNodeElement);
    // Instantiate new value node element of type IAddressElement
    // and define attributes.
    newAddressNodeElement = wdContext.createAddressElement();
    newAddressNodeElement.setCity(address.getCity());
    newAddressNodeElement.setCountry(address.getCountry());
    newAddressNodeElement.setHouseNo(address.getHouseNo());
    newAddressNodeElement.setPostalCode(address.getPostalCode());
    newAddressNodeElement.setStreet(address.getStreet());
    // the dependent context value node Address can only be referenced here for
    // each context value node element of type ICustomersElement separately because
    // we defined it to be a non-singleton node.
    newCustomerNodeElement.nodeAddress().bind(newAddressNodeElement);
  }
  //@@end
}

...
/*
 * The following coding section can be used for any Java coding that has
 * not to be visible to other controllers/views or that contains constructs
 * currently not supported directly by Web Dynpro (such as inner classes or
 * member variables etc.). </p>
 *
 * Note: The content of this section is in no way managed/controlled
 * neither by the Web Dynpro Designtime nor the Web Dynpro Runtime.
 */
//@@begin others
public SomeBOL someBOL = new SomeBOL();
//@@end

...
```

Note that, if you call the method **`bind(newCustomerNodeElement)`** instead of **`addElement(newCustomerNodeElement)`**, only the last bound node element of the type *Customer* is contained in the list of node elements. That is, **`wdContext.nodeCustomers().bind(newCustomerNodeElement)`** overwrites the list of node elements of the type *Customer*.

Result

Once the Master/Detail Viewer has been generated again and deployed in the Web browser again, it looks like this:



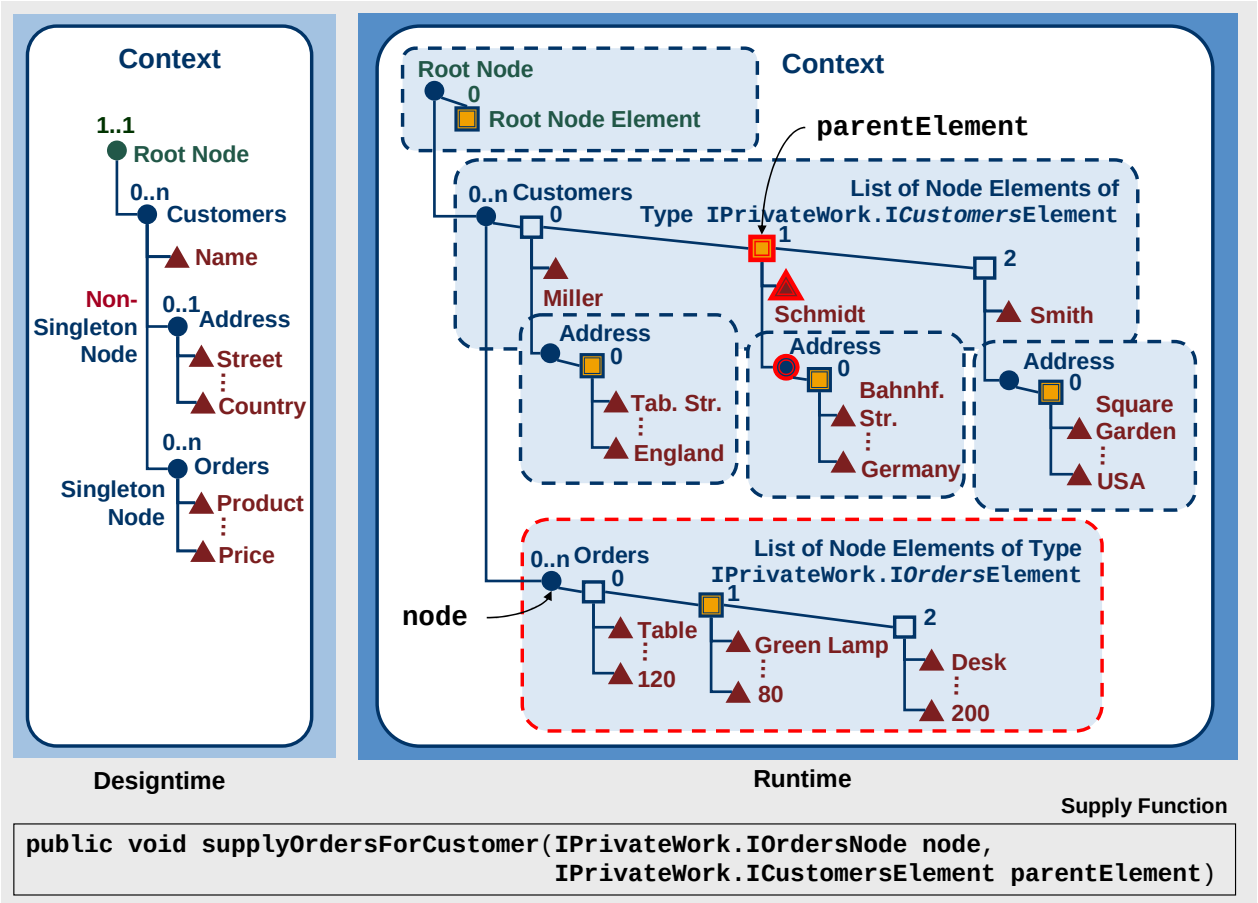
The (upper) *Customer* table is filled correctly by means of the data binding (already declared) between the UI elements associated with the table and the context elements. Conversely, the *Order* table remains empty. How can we display the order records, for the selected customer, in the Detail table (that is, the *Orders for Customers* table)?

To answer this question, refer to the section Implementation of a Supply Function [page 24]!

Adding a Supply Function

Our purpose here is to always adapt the *Order* entries in the (lower) detail table to match the entries for the currently selected *Customer* in the master table.

With respect to the context structure declared in the view controller, this means that the instantiated singleton value node *Orders* is to be filled with a new list of node elements of type *IOrdersElement*llen whenever the *Customer* selected in the upper table changes – that is, when the lead selection in the value node *Customer* changes. All you need to specify this list is the identifier of the currently-selected node element in the top-level node. In addition, a reference to the value node *Orders* must be known. The list of *Orders* node elements is to be bound to this node.



This graphic illustrates exactly this relationship. A supply function **`supplyOrdersForCustomer(Node node, NodeElement parentElement)`**, for the inner value node *Orders*, is called from the Web Dynpro runtime environment in the following cases:

- The node elements contained in the node are not available, but are to be displayed. In our example, that is the case when the application is launched, when the *Detail* table is to be filled with the *Order* records for the first customer.
- The lead selection of the top-level node changes. In the above example, this situation occurs when the user chooses a different *Customer* row in the master table. The *Orders* list in the *Detail* table is then invalid and the application must specify it again by calling the supply function.
- The inner value node, for which the supply function was declared, is invalidated by the application developer calling the method **`invalidate()`**. This means that the list of node elements associated with the value nodes is deleted. The supply function is then

called, if these lists are accessed again at runtime. Invalidating a node automatically implies invalidation of all the subnodes contained in it.

In our example, we only need to communicate the selection of a new *Customer* row in the upper table by declaring an action event called **`CustomerSelected`** and by binding the *Customer* table to this action event of the Web Dynpro environment. The associated event handler, **`onActionCustomerSelected`**(), does not need to be implemented. The supply function is called implicitly through the Web Dynpro runtime environment, which reacts when the lead selection changes.

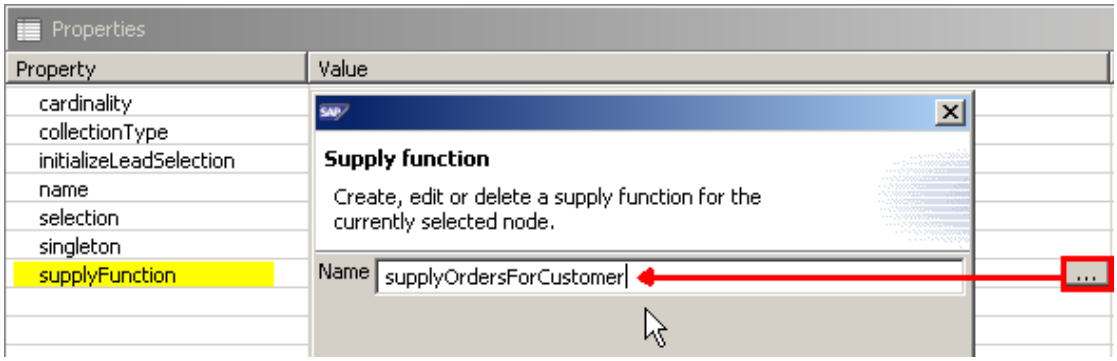
Procedure

To make the Master/Detail Viewer available:

Defining an event and binding it to the table

1. On the *Actions* tab of the *Work* view, declare an action and name it **`CustomerSelected`**.
2. On the *Layout* tab, bind the **`Event – onLeadSelect`** property of the UI element *CustomerTable[Table]* to the **`CustomerSelected`** action in the selection list on the right of the tab.

Declaring the supply function for an inner value node

3. In the *Work* view, choose the *Context* tab.
4. Select the inner context value node *Order.*
5. Assign the new supply function, **`supplyOrdersForCustomer`**, to the **`supplyFunction`** property. In the right-hand column, choose ... . In the dialog box that appears, enter the name of the supply function.
6. 

Implementing the supply function

7. Choose the *Implementation* tab to implement the view controller called *Work.java*.
8. After the supply function **`supplyOrdersForCustomer`** the controller implementation is extended to include this method. Insert the following source code in the user coding area:

Supply-Function `supplyOrdersForCustomer()` im *Work*-View-Controller

```
...
/**
 * Declared supply function for IPrivateWork.IOrderNode.
 * This method is called when the node is invalid and the collection is
 * requested. This may occur in any phase, even in the beginning to
 * initialize the node. The method is expected to fill the node
 * collection using IWDNode.bind(Collection) or
IWDNode.addElement(IWDNodeElement).
 *
 * @param node The node that is to be filled
```

```
 * @param parentElement The element that this node is child of. May be
 *        <code>null</code> if there is none.
 * @see com.sap.tc.webdynpro.progmodel.api.IWDNode#bind(Collection)
 * @see com.sap.tc.webdynpro.progmodel.api.IWDNode#bind(IWDNodeElement)
 */
public void supplyOrdersForCustomer(IPrivateWork.IOrdersNode node,
                                    IPrivateWork.ICustomersElement parentElement)
{
  //@@begin supplyOrdersForCustomer(IWDNode,IWDNodeElement)
  SomeBOL.Order order;
  IPrivateWork.IOrdersElement newOrderNodeElement;
  // get list of Orders for given customer from some BOL
  Collection orders = someBOL.getOrdersForCustomer(parentElement.getName());
  for (Iterator iter = orders.iterator(); iter.hasNext();){
    order = (SomeBOL.Order)iter.next();
    // create new instance of a node element of type Order
    newOrderNodeElement = wdContext.createOrdersElement();
    // set context value attributes contained in context value node of type Order
    newOrderNodeElement.setCurrency(order.getCurrency());
    newOrderNodeElement.setDate(order.getDate());
    newOrderNodeElement.setPrice(order.getPrice());
    newOrderNodeElement.setProduct(order.getProduct());
    // Add new node element of type Order to node of type IPrivateWork.IOrderNode.
    // This means: add node element newOrderNodeElement to collection of
    // node elements hold by singleton context value node node (Order)
    node.addElement(newOrderNodeElement);
  }
  //@@end
}

...
```

Result

After you include the supply function **supplyOrdersForCustomer**, the Master/Detail application behaves as desired in the Web browser. Each time the user selects a different row in the upper table, the details table is filled again with the associated order records. As an application developer, you simply need to implement a suitable supply function for the inner value node *Orders* and bind the `LeadSelect` event of the customer table to an action, to trigger communication with the runtime environment. Each row entry in the *Order* table is transported again using the mechanism of data binding between table UI elements and the controller context of a view.