In Chapter 2, you’ll get some of the the technical background you need to develop custom business logic for SAP CRM with SAP Web Client. Get up to speed with the BOL and GenIL, and then learn how to create models, implement queries, and more.

“Introducing the GenIL Component”

Contents

Index

The Authors

Tzanko Stefanov, Armand Sezikeye, Sanjeet Mall

SAP Web Client

624 Pages, 2011, $79.95

www.sap-press.com/2847
Web applications call for extensive programming, and complex systems like these require heavy-duty code. Data buffering, caching, transactions and various kinds of runtime optimization can add a lot to your workload. You need some help in terms of services, APIs, and programming best practices.

2 Introducing the GenIL Component

In this chapter, we will examine what it takes to develop custom business logic. We will use the hotel guest-booking scenario to present the technical concepts and tools introduced by the Web Client UI framework in SAP CRM to help expose application logic in a uniform and elegant way. These concepts are known as Business Object Layer (BOL) and Generic Interaction Layer (GenIL). We already mentioned them in the first chapter. However, even if you are already familiar with SAP CRM, the chances are that BOL and GenIL are still unfamiliar. In this chapter, we hope to clear up some of the mystery. We will start with a simple example and work from the backend up to the UI. Along the way, we will show some of the latest features that simplify and speed up GenIL component development.

This and the next few chapters will probably be the most technically challenging ones. Therefore, our example might seem too primitive at times. However, our main goal is to explain the GenIL programming model as comprehensively as possible, rather than to deliver great business value. Our explanations are accompanied by plenty of code, so we urge you to try to reproduce the sample yourself (if you have a development SAP CRM system at hand). This way you can experiment on your own and eventually master the best application development practices of SAP CRM (Web Client UI) application.

In addition to explaining the architecture of the GenIL layer, in the next sections we will show you how to build a custom GenIL model. We will demonstrate what it takes to implement data query and read operations. Last, we will look at the tools available for testing GenIL components.
2.1 Introduction

While we do not want to repeat the architecture discussions from the previous chapter, we feel that you should constantly keep the big picture in mind and know exactly where you are in terms of application layering. In this section, we review the overall Web Client UI architecture, and then briefly introduce the concepts of BOL and GenIL.

2.1.1 Web Client UI Architecture

By now you should be fairly comfortable with the concept of multi-tier software architecture. You know that the Web Client UI facilitates the development of classical three-tier applications. These consist of the user interface (presentation layer), business-logic implementation layer, and data access layer (persistency layer).

Those of you familiar with enterprise application development know that data buffering and transaction control are not only very important, but also complex. Any mistakes in these will impose a heavy penalty when developing an enterprise-ready application. By enterprise-ready, we refer to systems plugged into a complex landscape, accessible simultaneously by many users at the same time and across different channels. By users, we mean other systems as well as human beings. In the context of our business scenarios, if SAP CRM was consuming data from the existing booking system via Web services, it would be considered a user.

One has to distinguish between developing user interfaces and manipulating transactional data. In big projects, you may often find separate development teams focus on the presentation layer (frontend) and the underlying business-logic implementation (backend). This is often because the heavy-duty backend logic is implemented in a dedicated layer and uses completely different concepts and infrastructures than the frontend. Some might argue that one can build a backend as part of your UI. We respond that in programming we take pride in reusability. It is important to develop common logic once and use it everywhere. For example, service and sales orders involve the same users. Do you really want to code the logic to retrieve the related business partners in each page that contains a list of parties involved? Ideally you will create instead a reusable piece of code and plug it in wherever needed. As a matter of fact, you can look at the whole one-order implementation and see that the objects there are used across most of the SAP CRM business applications. If you have to code the logic for retrieving the one-order data in the presentation layer, you could easily multiply the code in SAP CRM by a factor of hundreds. The worst consequence will be maintaining that code. It will be "mission impossible."

Hopefully, we now agree on the need to encapsulate your business rules and logic in a separate layer. In addition, we need to do it in an object-oriented manner and keep the semantics of the SAP CRM business objects. This is what we will focus in this chapter. Figure 2.1 gives a high level overview of the different layers/tiers available in SAP CRM, but puts the focus on the business layer. In Web Client UI, this layer is supported by two frameworks: the Generic Interaction Layer (GenIL) and the Business Object Layer (BOL).

![Diagram of Web Client UI Multi-Tier Architecture](image)

Figure 2.1 Web Client UI Multi-Tier Architecture

2.1.2 Introduction to BOL and GenIL

The BOL provides a uniform API that is used by the presentation layer (UI Components) to access the required SAP CRM business data. Behind the BOL sits the GenIL. The latter wraps the SAP CRM application logic. You can think of the two frameworks as different faces of the same coin. When you are building a UI and reading backend data, you will be working with the BOL APIs. When you are
coding the backend logic (often wrapping existing application APIs) you will be implementing the GenIL APIs. In both cases, your business objects will be described via metadata as in the GenIL object model. Some might find it confusing that both BOL and GenIL use the same model, but in the end metadata is API-agnostic and both layers should be working with the same semantics. Figure 2.2 illustrates the concepts that we will examine in depth.

Figure 2.2 BOL versus GenIL

BOL allows you to work with entities. From an API point of view, you will be dealing with generic programming objects (for example CL_CRM_BOL_ENTITY, CL_CRM_BOL_QUERY_SERVICE, and so on). You should not expect specialized classes that represent concrete business objects (such as orders, items, business partners, etc). The concrete type of BOL entity will come from the GenIL model information associated with it (its metadata). You will be traversing a tree of entities connected via relations and requesting attribute values as per the defined model. All the entities will be represented by the same ABAP class (CL_CRM_BOL_ENTITY). But you will distinguish and access an entity by its object name (part of its metadata). The same applies to the entity attributes and the relationships between the entities. Those of you familiar with reflection should be comfortable with this concept. If you are not one of those, do not despair. We will provide plenty of examples. Apart from accessing entities and their properties, BOL allows you to execute queries. A query returns a result that is nothing but a collection of entities.

Let’s provide a simple example of BOL programming. Suppose we have a reference to a business transaction (an order); namely, the object ID attribute value. Our goal is to retrieve the description of the order. We will execute a query called BTQUERY10 and retrieve the DESCRIPTION attribute from the order’s header. The latter is represented via a GenIL business object called BTORDERHEADER. The sample code to achieve that task is provided in Listing 2.1.

Listing 2.1 Querying and Reading Transaction Data with BOL API

In addition to allowing you to work with entity and queries, BOL also has an in-memory cache service that reduces the unnecessary calls to the GenIL. The GenIL layer dispatches the requests to the appropriate GenIL component(s) and bundles the results before it returns the request. You can think of the GenIL components as an implementation of your business objects. Each GenIL component is represented by a dedicated class that implements a set of interfaces. In classical object-oriented programming, you would probably write one class per entity. Then you probably would link the entities in your hierarchy via composition and aggregation. For example, you might have a field property in each class that points to a concrete instance of another class (creating a relationship between classes). This is not how the GenIL works. GenIL requires only one class: the GenIL component class. This class has dedicated methods for reading, modifying, saving and searching for entities (all entities). The entity hierarchy is captured in the GenIL (object) model, namely as metadata. Your implementation of create, read, modify,
and search methods takes the model into consideration and retrieves whatever is requested by the consumer. The consumer is the BOL API.

The examples in this chapter will help you understand this. We will focus on the GenIL and ignore the BOL until later in the book. For now, it is important to remember that the BOL provides a uniform and yet simple way to access the SAP CRM business data. BOL is nothing but an add-on layer on top of GenIL. Among other things, BOL caches the retrieved data and provides transaction management. GenIL wraps the SAP CRM application APIs and delivers a uniform interface to retrieve and make persistent the application data. The later is semantically described via an object model used by both GenIL and BOL. We will refer to the entities in this model as business objects, or BOs.

### GenIL Business Object

The GenIL model defines the semantics of a business application. Object-oriented programming implies that a business application is implemented via the objects. Each object is defined in a programming class. In GenIL, you have only one class: the GenIL component class. It is typically designed to work with a specific set of metadata: the GenIL model. The latter consists of nodes that are connected via relationships. Each node has attributes; this is where the data comes from. There is clearly a parallel between the nodes/entities in the GenIL model and an object in object-oriented programming. Therefore, in the SAP documentation, these entities are called Business Objects (BOs). Unfortunately, this is pretty close to another term, namely the SAP CRM business objects (applications such as account, opportunity, and campaign).

#### 2.2 Creating your First GenIL Model

We often hear in computer science classes that the first thing we need to do when writing an object-oriented class diagram is to define our objects. From there, we can easily create a class diagram that contains our object types, their attributes, and the relationships between the objects. The equivalent of this process in a database design exercise is the entity relationship diagram. Implementing GenIL components is no different. You need to start with the object model (also known as the GenIL model).

#### 2.2.1 GenIL Model Overview

The GenIL model defines a hierarchy of objects. We agreed to refer to these objects as business objects (BOs). Each BO has a flat structure of attributes associated with it. There is always at least one root object node.

The object nodes are related to each other via GenIL relations. The relations could be aggregation, composition, and association. Aggregation and composition are possible between nodes of the same object model, while association is possible between nodes of different object model hierarchies. Relationships are unidirectional: They have a source and target object. They can also be one-to-one (1:1) or one-to-many (1:N).

Apart from root object and relationships, a GenIL model also contains access and dependent objects. Root, access, and dependent objects constitute the entities of an object model. A dependent object cannot be determined uniquely by its ID. Instead you need also the ID of its parent. Sometimes only the ID of the parent will suffice, and sometimes you might need both the ID of the dependent and the ID of the parent. We recommend that you always add the keys of the parent to the key of the dependent object. An access object is a special kind of object. It is not a root object, but it can be accessed by its own ID. Therefore an access object can be used to retrieve its dependent children. One can conclude that a root object is also an access object.

There are also node types that facilitate the process of searching for entities. A query object node is an object whose attributes are used as parameters of entity search requests. The result of such a query would be access objects or query result objects. One severe limitation of query objects is that a search parameter can take only a single value or pattern and use it to produce a match.

This limitation of the query objects is overcome by dynamic query objects. A search parameter in this case can also be associated with a logical operator, such as greater than, less than, or in between. As a result the match does not have to be a direct one.

As mentioned already, a query can return a special node type called query result object. Why not return an access or root object? This is also a valid approach, but imagine what will happen when you want to return a set of attributes from more than one object (a subset of the whole) for faster access. Alternatively, you might wish to return fewer attributes than in the original root or access BO, for
Introducing the GenIL Component

performance reasons. Being able to return a lean read-only object type will allow you to implement such scenarios.

One can also define view objects. These node types represent a named part of the object model. They are read-only. When they are requested, a hard-coded set of objects will be returned.

As of CRM 7.0 EhP1, one can define abstract objects. The idea is to promote reuse of the modeled data. An abstract object is defined in an object model just like any other node type. There needs to be at least one real access or dependent object inheriting from the abstract one. At runtime, only the concrete node is used.

If you want to explore an existing object model, take a look at the SAMPLE GenIL component. You need to run Transaction GENIL_MODEL_BROWSER and provide SAMPLE as a component name. You should see something very similar to Figure 2.3.

Creating your First GenIL Model

2.2 Defining a GenIL Model

How can you define a GenIL model? You could implement the IF_GENIL_APPL_MODEL interface. Although this approach offers lots of flexibility it is not the easiest one. As of EhP1, you can use Transaction GENIL_MODEL_EDITOR and define your model interactively there. You edit an existing model by simply clicking the Edit button in GENIL_MODEL_EDITOR. There is another transaction that is identical to the model editor: GENIL_MODEL_BROWSER. The “browser” precedes the “editor” (before EhP1). In the past, one had to define the model manually. In this book, we will use the model editor. The easiest way to use this approach is to ensure that your GenIL component class extends the default implementation of IF_GENIL_APPLMODEL_PERSISTENCY, namely CL_WCF_GENIL_ABSTR_COMPONENT. We will discuss the GenIL interfaces and classes in more detail later in this chapter.

There are three model implementation strategies that we would like to quickly describe:

- You can have dynamic models based on a custom IF_GENIL_APPL_MODEL implementation.
- Using the model editor, you can easily create static models. An alternative to the model editor is to implement IF_GENIL_APPLMODEL_PERSISTENCY and store your model metadata in custom tables.
- You can also have a static model with dynamic parts. To do this, use the model editor but extend IF_GENIL_APPL_MODEL methods to enhance the model dynamically.

2.2.3 Creating a GenIL Model

In case you are confused at this point, let’s create a simple GenIL model. We will use our favorite example, namely the hotel guest-booking application. In our database model, we have two tables. One will store the hotel bookings of our guests and the other will have all the products and services used by our guests during each stay. The booking table will be the master and the products used will be the details. If you browse these tables in Transaction SE11 you will have something similar to Figure 2.4.

Our guest-booking data (table ZBOOKING_RESERV) consists of the ID of the guest, the name of the hotel (the assumption is that it is unique), the check-in date, the stay-in days, the number of people who stayed in the room, and the membership type at the time of booking. As part of each booking, we will keep track of the products used by the guest, the date of use and how the guest rated the product or service (ZBOOK_PRODUCTS). As you can see, the key of ZBOOKING_RESERV (our booking table) participates in the key of ZBOOK_PRODUCTS. In addition to this, we will limit the possible entries for membership and rating via a check table and domain fixed values respectively (see Figure 2.5).
Creating your First GenIL Model

At this point you might be tempted to start the model editor and define your model. But first we will create a GenIL component. You will recall, however, that a GenIL component implies that there is an implementation class. Below are the activities that you have to perform.

1. The first task is to start Transaction SE80 or SE24 and create a new ABAP class.
2. As we want this class to use a statically defined model in the model editor we will inherit from CL_WCF_GENIL_ABSTR_COMPONENT (as discussed previously).
3. Give this class a name; in our example it will be ZCL_ZGENIL_COMP_BOOKINGS.
4. Save and activate the class.
5. Once done, you can register the GenIL component. Go to Transaction SM34, enter the view-cluster name CRMVC_GIL_APPDEF and hit the maintain button (you can also use maintenance view CRMV_GIL_COMP). You will notice many other component definitions; for example the SAMPLE component should be there.
6. Define your component as per Figure 2.6.

Creating and Registering the GenIL Component

In the view cluster there is a node called COMPONENT SET DEFINITION. A GenIL component set is a predefined set of components that will be used together. For example, when you want to use BOL/GenIL in your UI components, you need to provide a component set. Once a component set is loaded, all its components get loaded. Why not load all the available components in your system? Fortunately, you cannot do that (unless you include all the components in a dedicated component set). This would lead to an unnecessary high system load and is usually not required. The GenIL core (CL_CRM_GENERIC_IL_NEW) requests a component set as a parameter when it is instantiated. After instantiation, it is possible to load further
GenIL components and component sets dynamically (LOAD_COMPONENT, LOAD_COMPONENT_SET). The model exposed by each component is integrated into the overall runtime model. In our example, see Figure 2.7, we will create a component set called ZBOOK and assign our ZBOOK component to it (we use the same name for the component and the component set).

Creating GenIL Business Objects

It is time to start the model editor and create our model. Launch Transaction GENIL_MODEL_EDITOR, enter "ZBOOK" as a component name and click on CHANGE. The model node is selected by default and shows our component definition. First we need to create a root object. Click on the root objects node and select CREATE ROOT OBJECT. A fairly complex menu will appear. Do not worry; we do not need to fill all the entries. What we need to provide is an object name, Bookings, and three structures: KEY STRUCTURE NAME, ATTRIBUTE STRUCTURE NAME, and CREATE STRUCTURE NAME. As the notation suggests, the first structure is used to store the key of an object. The second one carries all the properties of the object, and the third one contains the data required to create an object (for example all the required fields). In our example, we use the same structure as key and creation data. The attribute structure exposes all the attributes from our table. Let’s take a short detour and create these structures in the DDIC. While we are there, let us also define table types based on these structures (as line types). Figure 2.8 shows what we have used in our example.

With all the above in place, we can return to our model and finish the definition of our Bookings object. Once you are done, your definition should look very similar to Figure 2.9. Note that we have checked the WEBSERVICE ENABLED checkbox, as we want to be able to generate Web services out of this object node. We will talk about the Web Service Tool (WST) later.

We could define some methods on a BOL node, but we will leave this exercise for later. For now we will stick to the basics and create a dependent object for our
used products. As this is a dependent BO, we will need key and attribute structures only. Make sure you have them in DDIC as per Figure 2.10.

**Figure 2.10** DDIC Structures for the Used Products BOL Node

Creating your First GenIL Model

As with the Bookings we will go ahead and create table types using the above structures as line types. The name of these tables will be `ZBOL_BOOK_PRODUCTS_T_KEY` (for the keys) and `ZBOL_BOOK_PRODUCTS_T_ATT` (for all used product attributes).

In the model editor, right click on the DEPENDENT OBJECTS node and select to create a dependent object. Fill in the required data as per Figure 2.11.

Note that we are specifying a root object, namely our Bookings BOL node. The **Web Service Enabled** box is checked.

**Creating GenIL Relations**

Once we have the root and dependent objects, we have to connect them via a relation. We briefly described the concept of relations before, but let’s dig a bit deeper. The relation between a root object and a dependent object is expected to be an aggregation or composition. What if we need to specify a relation between two root objects? In that case, we shall define a relation of type association between them. The association in general can be used to link any kind of object. It can be used across components with root or access objects as a target.

A relation (aggregation, composition, or association) is always unidirectional; that is, it starts from a source (parent) object and points to a target (child) object. In addition a relation has cardinality. Although there are more options, stick to 1:1 or 1:N relations.

In our example we will define a 1:N aggregation between Bookings and UsedProducts. We could have used composition, but this would imply that there is always a child object (i.e., a guest will always use a product, something we do not assume). Figure 2.12 shows our relation definition.

**Figure 2.12** BookingProductsRel Relation
Introducing the GenIL Component

GenIL Business Object Attribute Properties

Let’s go back to our booking object and expand the Attribute Structure node. You will see all the attributes. If you double-click on an attribute, the attribute details will open on the right. There is one property that we want to specify there: Attr. It could take several possible values such as Read-Only, Changeable, Hidden, Required Entry, and Technical Field. The names of the options are largely self-explanatory. We want our key fields to be read-only, while the others are changeable. Figure 2.13 shows how the STAYDAYS should look. You can modify the rest of the fields according to whether they are part of the key. Do the same for the UsedProducts object.

Setting Attribute Properties

Modeling GenIL Dynamic Queries

We also need queries to search for data. We will show you how to create a query that returns a collection of bookings matching certain criteria. As we want to use logical operators such as GREATER THAN, we will define a dynamic query. If you recall, the attributes of a query object are used as search parameters. In our example we want to search by guest ID, hotel name, check-in date, and products used. Therefore we will define a new attribute structure that contains these attributes. We chose to name it ZBOL_SEARCHBOOKINGS_ATT. The query will return a collection of Bookings. Hence, this will be our result object. A result object can be of THE typeS root, access, or query result. Queries require knowing which root object within the component they relate to. We have only one root object: Bookings. In the model, we will specify that we will use all the possible query logical operators, although we will not implement all of them. Figure 2.14 shows what the final result should look like.

Implementing a GenIL Query

The minimum that you need in order to read data via GenIL is an implementation of a query and a method that retrieves the attributes of an object identified by a query. Before we implement anything, it will help to make a quick overview of the GenIL interfaces and classes that we will use. These are shown in Figure 2.15. IF_GENIL_APPL_MODEL is related to the GenIL object model. Because we are using the Genil Model Editor to define our model, we do not need to worry about this one. Recently, SAP added an optional interface for dynamic metadata: IF_GENIL_APPL_DYN_META_INFO. We will not be focusing on this either. However, we will be dealing extensively with IF_GENIL_APPL_INTLAY because it defines the way we access our data. So does IF_GENIL_APPL_ALTERNATIVE_DSIL. DSIL stands for Delete Save Init Lock. We will use both of them, although some of the methods in
Introducing the GenIL Component

The interfaces IF_GENIL_APPL_INTLAY have been deprecated, and you should use the versions from IF_GENIL_APPL_ALTERNATIVE_DSIL. Instead of implementing these interfaces from scratch, you should use the abstract base classes CL_CRM_GENIL_ABSTR_COMPONENT and CL_CRM_GENIL_ABSTR_COMPONENT2. A key difference between these two classes is that the first one restricts the data type of the primary key for root and access objects to a RAW16 GUID, while the second one allows arbitrary keys. If you want to take advantage of the model editor you should ensure that you implement IF_GENIL_APPLMODEL_PERSISTENCY.

By simply assessing our requirements about the GenIL Model Editor and understanding the above class diagram, it is clear that it will be best to use CL_WCF_GENIL_ABSTR_COMPONENT as a super class. SAP recommends that you either use this abstract class or CL_CRM_GENIL_ABSTR_COMPONENT2. We will discuss IF_GENIL_WEBSERVICE_SUPPORT later in Chapter 12.

Implementing a GenIL Query

CL_WCF* vs. CL_CRM* Classes

You might have noticed that the naming convention of the GenIL classes changes from CL_CRM* to CL_WCF*. In the past, Web Client UI was used only in the context of SAP CRM. Later, it was declared an SAP user interface technology in its own right.

Based on the above discussion, it should be clear why we chose CL_WCF_GENIL_ABSTR_COMPONENT to be the superclass of our GenIL component class (ZCL_ZGENIL_COMP_BOOKINGS).

In order to produce a dynamic query we need to implement IF_GENIL_APPL_INTLAY~GET_DYNAMIC_QUERY_RESULT. This method requires the following parameters:

- IV_QUERY_NAME
  - Contains the name of the query currently processed. As all of the queries will be handled by this method, you need the query name to trigger different implementations/APIs.

- IS_QUERY_PARAMETERS
  - Some additional query parameters, like maximum number of hits, are passed through this structure.

- IT_SELECTION_PARAMETERS
  - Contains the selection parameters (along with operations) passed to the query.

- IV_ROOT_LIST
  - A reference to an empty data container. You should fill it with the result objects from your query.

For SearchBookings we will have to query our database tables using the query selection and query parameters provided by the user. We need to then process the result, and for each found entry to create a Bookings object instance. This sounds fairly easy, but we need to get familiar with the GenIL APIs that will help us achieve that. But before we do so, we will walk you through the backend APIs that will wrap the database access logic of our application.

2.3.1 Defining the Backend APIs

The backend APIs should be GenIL-agnostic (not depend on any GenIL interfaces and classes). We will resort to static methods in a class called ZCL_BOOKINGS_BACKEND_API. Feel free to do this in another way. For the query, we will have a method...
Introducing the GenIL Component called SEARCH_BOOKINGS. It will accept a string that contains the WHERE clause of a SELECT statement and the maximum number of records. The method will return the matching records in internal table ZBOL_BOOKINGS_T_KEY (represented by keys of the found Bookings). The complete programming code can be found in Listing 2.2.

Method SEARCH_BOOKINGS.

REFRESH et_results.

IF iv_string IS NOT INITIAL.

SELECT GUEST_ID HOTEL_NAME LASTCHECKIN
FROM ZBOOKING_RESERV
UP TO iv_max_hits ROWS
INTO CORRESPONDING FIELDS OF TABLE et_results
WHERE (iv_string).
ELSE.

SELECT GUEST_ID HOTEL_NAME LASTCHECKIN
FROM ZBOOKING_RESERV
UP TO iv_max_hits ROWS
INTO CORRESPONDING FIELDS OF TABLE et_results.
ENDIF.
endmethod.

Retrieving Bookings from Database

Listing 2.2

The above code is simple and has nothing to do with GenIL programming. It does highlight what we believe is the recommended approach: Put your persistence and business logic in backend APIs and use the GenIL component only as a wrapper that contains Web Client UI specific code.

2.3.2 Implementing GET_DYNAMIC_QUERY_RESULT

In order to implement a dynamic query we have to redefine IF_GENIL_APPL_ INTLAY~GET_DYNAMIC_QUERY_RESULT. You can see the result in Listing 2.3

DATA: lr_msg_cont TYPE REF TO cl_crm_genil_global_mess_cont,
lv_num_hits TYPE i,
lt_results TYPE ZBOL_BOOKINGS_T_KEY,
iv_max_hits TYPE char5,
iv_max_hits_tmp TYPE int4,
lr_root_object TYPE REF TO if_genil_cont_root_object,
lt_request_obj TYPE CRRM_REQUEST_OBJ_TAB.
FIELD-SYMBOLS: <fs_results> TYPE ZBOL_BOOKINGS_KEY.

* Retrieve the message container to log eventual messages
  lr_msg_cont = iv_root_list->get_global_message_container( ).

* When the max hits is set to 0.
  IF is_query_parameters-max_hits IS INITIAL.
    lv_max_hits_tmp = 100. "100 when no max_hits is set
  ELSE.
    lv_max_hits_tmp = is_query_parameters-max_hits.
  ENDIF.

* select the right query
  case IV_QUERY_NAME.
  when 'SearchBookings'.
    Call the API and get the result.
    CALL METHOD SEARCH_BOOKINGS
      EXPORTING
        it_search_criteria = it_selection_parameters
        iv_max_hits = lv_max_hits_tmp
      IMPORTING
        et_results = lt_results.
    IF lt_results IS NOT INITIAL.
      * Log a message if search result exceed the max hit limit
        DESCRIBE TABLE lt_results LINES lv_num_hits.
        IF lv_num_hits > lv_max_hits_tmp.
          lv_max_hits = lv_max_hits_tmp.
          lr_msg_cont->add_message(  iv_msg_type = 'I'
            iv_msg_id = 'ZBOOKING_MSG'
            iv_msg_number = '000'
            iv_msg_v1 = lv_max_hits
            iv_show_only_once = abap_true ).
        ENDIF.

      * Loop through the results to build search result objects
        LOOP AT lt_results ASSIGNING <fs_results>.
        
        TRY.
        * Try to create a new result object
          lr_root_object = iv_root_list->add_object(  iv_object_name = 'Bookings'
            iv_object_key = <fs_results> ).

        * Flag it as direct query result
          lr_root_object->set_query_root( abap_true ).
CATCH CX_CRM_GENIL_DUPLICATE_REL CX_CRM_GENIL_MODEL_ERROR.
* Since the given object name is correct this could not happen!
ENDTRY.
* Note: The request object restricts the attributes to read.
* If there is no request object entry or the attributes
  table is empty all attributes are requested.
* read the attributes and relation using the GET_OBJECTS
  ME->IF_GENIL_APPL_INTLAY~GET_OBJECTS( IT_REQUEST_OBJECTS = LT_REQUEST_OBJ
  IV_ROOT_LIST = IV_ROOT_LIST ).
ELSE.
    lr_msg_cont->add_message( iv_msg_type = 'W'
      iv_msg_id = 'ZBOOKING_MSG'
      iv_msg_number = '001'
      iv_show_only_once = abap_true ).
ENDIF.
when others.
  return.
endcase.

Listing 2.3 Redefining IF_GENIL_APPL_INTLAY~GET_DYNAMIC_QUERY_RESULT

In our code, we first get a reference to the global message container. We will discuss this later. At this point, it’s sufficient to say that you use the message container to register all kinds of messages (the error message being among the most popular). The framework will later propagate these messages to the user.

lr_msg_cont = iv_root_list->get_global_message_container( ).

Next we will put in a safety net and ensure that we will not overload the system by processing too many records. In case the maximum number of hits is not set, we will limit the result to 100 entries.

IF is_query_parameters-max_hits IS INITIAL.
  lv_max_hits tmp = 100. "100 when no max_hits is set
ELSE.
  lv_max_hits tmp = is_query_parameters-max_hits.
ENDIF.

As you know, our GenIL component will be used by all GenIL objects in the related models. Although we have only one query object, it is good practice to always evaluate the IV_QUERY_NAME. If its value is SearchBookings, we execute our query logic.

case IV_QUERY_NAME.
  when 'SearchBookings'.
    In our example, we process the booking searches in a separate method in the GenIL component, SEARCH_BOOKINGS. Please do not confuse it with the SEARCH_BOOKINGS of the ZCL_BOOKINGS_BACKEND_API. We will see how these two are related in just a moment.

    CALL METHOD SEARCH_BOOKINGS
      EXPORTING
        it_search_criteria = it_selection_parameters
        iv_max_hits = lv_max_hits_tmp
      IMPORTING
        et_results = lt_results.

    The instance method SEARCH_BOOKINGS returns a table of the matching Bookings keys (lt_results of type ZBOL_BOOKINGS_T_KEY). If we get IF lt_results IS NOT INITIAL, we know that the search produced results and we need to process them. We want to inform the user when the number of results exceeds the maximum hits value (that is, when there are more results than the ones presented to the user). We will use the message container and log an informational message.

    IF lv_num_hits > lv_max_hits tmp.
      lv_max_hits = lv_max_hits tmp.
      lr_msg_cont->add_message( iv_msg_type = 'I'
        iv_msg_id = 'ZBOOKING_MSG'
        iv_msg_number = '000'
        iv_show_only_once = abap_true ).
    ENDF.

    Using the message container is really simple: You simply add messages to it. You need to indicate the message type/severity (for example, "I" is an information message), message class, message ID, and so on. At the end the framework looks at all the message containers and ensures that the messages are presented to the user.

    Next, we loop through the found keys.

    LOOP AT lt_results ASSIGNING <fs_results>.
For each record in the result table we create a new Bookings GenIL object and add it to the list of container objects (we will discuss the container object concept in Section 2.5.2).

```abap
lr_root_object = iv_root_list->add_object(
  iv_object_name = 'Bookings'
  iv_object_key = <fs_results> ).
```

The structure that we pass to the add_object method must correspond to the create DDIC structure that we used when we declared our Bookings object. What you observe here is the creation of a data container object (if_genil_cont_root_object). This is a very important concept in GenIL programming. The data container is the way GenIL transports objects (remember that unlike object-oriented programming, the GenIL business objects do not have dedicated classes to represent them). The data container also has an API for setting and accessing the data. You can call SET_ATTRIBUTES and GET_ATTRIBUTES to set and read data. In our example, we are calling the data container’s SET_QUERY_ROOT method to flag this object as root (you need to do this for root objects):

```abap
lr_root_object->set_query_root( abap_true ).
```

Once we have compiled a list of matching root objects, we call IF_GENIL_APPL_INTLAY~GET_OBJECTS to read the Bookings attributes (and maybe even the related objects). Remember, so far we have simply identified its keys. We will look at the GET_OBJECT implementation very soon.

```abap
ME->IF_GENIL_APPL_INTLAY~GET_OBJECTS(
  IT_REQUEST_OBJECTS = LT_REQUEST_OBJ
  IV_ROOT_LIST       = IV_ROOT_LIST ).
```

In our IF_GENIL_APPL_INTLAY~GET_DYNAMIC_QUERY_RESULT there is no application specific code (except for the name of the query). Everything is strictly related to what you need to do from a GenIL perspective.

### 2.3.3 Implementing Helper Methods

It is time to examine the private SEARCH_BOOKINGS method of our GenIL class (Listing 2.4).

```abap
DATA: LV_MAX_HITS TYPE I,
     LV_STRING TYPE STRING,
     LT_SEARCH_CRITERIA TYPE GENILT_SELECTION_PARAMETER_TAB,

FIELD-Symbols: <SEARCH_CRITERIA> TYPE GENILT_SELECTION_PARAMETER.

* NOTE: 
* This is a good place to implement Authority checks!!!
*
LV_MAX_HITS = IV_MAX_HITS.
IF LV_MAX_HITS EQ 0.
  LV_MAX_HITS = 100.
ENDIF.

LT_SEARCH_CRITERIA = IT_SEARCH_CRITERIA.
SORT LT_SEARCH_CRITERIA BY ATTR_NAME.
LOOP AT LT_SEARCH_CRITERIA ASSIGNING <SEARCH_CRITERIA>.
  * NOTE: * The current implementation takes care only of CP and EQ
    search criteria options.
  * 
    IF ( <SEARCH_CRITERIA>-OPTION EQ 'CP' OR <SEARCH_CRITERIA>-OPTION EQ 'EQ' ).
      IF LV_STRING IS NOT INITIAL.
        IF <SEARCH_CRITERIA>-ATTR_NAME NE LV_LAST_ATTR_NAME.
          CONCATENATE LV_STRING ') AND (' INTO LV_STRING SEPARATED BY SPACE.
        ELSE.
          CONCATENATE LV_STRING 'OR' INTO LV_STRING SEPARATED BY SPACE.
        ENDIF.
      ELSE.
        LV_STRING = '('.
      ENDIF.
      CONCATENATE LV_STRING <SEARCH_CRITERIA>-ATTR_NAME <SEARCH_CRITERIA>-OPTION '''' INTO LV_STRING SEPARATED BY SPACE.
      CONCATENATE LV_STRING '''' INTO LV_STRING.
    ENDIF.
    LV_LAST_ATTR_NAME = <SEARCH_CRITERIA>-ATTR_NAME.
  ENDLOOP.

* retrieve data from the backend
```
CALL METHOD ZCL_BOOKINGS_BACKEND_API=>SEARCH_BOOKINGS(
    EXPORTING IV_STRING = LV_STRING
    IV_MAX_HITS = LV_MAX_HITS
    IMPORTING ET_RESULTS = ET_RESULTS ).

Listing 2.4 GenIL Component’s SEARCH_BOOKINGS

The first thing we do is throw in another safety net related to the maximum number of returned results. The idea is that the SEARCH_BOOKINGS can be used by other methods. Next we make a copy of the query criteria and sort it by name.

LT_SEARCH_CRITERIA = IT_SEARCH_CRITERIA.
SORT LT_SEARCH_CRITERIA BY ATTR_NAME.

The largest section of code is inside the LOOP AT LT_SEARCH_CRITERIA. If you think in terms of SQL SELECT statements, this is where we build our WHERE clause string based on each search criterion. As you can see, the structure of the query parameter (GENILT_SELECTION_PARAMETER) is pretty comprehensive, storing not just name-value pairs, but also operators, lower and upper values, and so on.

At the end we finally call the backend API that we introduced earlier in this section, namely the ZCL_BOOKINGS_BACKEND_API=>SEARCH_BOOKINGS. It is the one that performs the DB select using the WHERE clause that we constructed in the helper method SEARCH_BOOKINGS from our GenIL component.

2.3.4 Conclusion

This concludes our query logic. To summarize, we evaluated the query name parameter, and when it was equal to SearchBookings we called the backend APIs to execute the search. We took into account the query parameters and limited our result list to the maximum number of hits. Once we obtained the keys of the matching Bookings records we filled in the root list data container with Bookings entities (container objects).

One might ask what happened to the GET_OBJECTS call at the end of our GET_DYNAMIC_QUERY_RESULT. Before we answer that, let’s just comment it out and test our code without that call. But how can we test it? Some of you probably are already thinking about using BOL APIs. That would work, but there is an easier way.

2.4 Introducing the BOL Browser

Transaction GENIL_BOL_BROWSER is a convenient tool with which to test GenIL components. When you start the transaction, you have to provide a COMPONENT NAME (or COMPONENT SET NAME). In our example, we will use component ZBOOK.

You will be presented with a screen divided into several panels. In the left-hand panel, you will find all the query objects present in the GenIL model. In our example there is only one query object. When you double-click on it, the top of the right-hand panel will show the query parameters. Upon execution, they will be passed to the query method via IS_QUERY_PARAMETERS. Set the MAX_HITS to "5."

Below the search criteria, there is a section where you can define the dynamic search selection parameters (the IT_SELECTION_PARAMETERS importing the table of method GET_DYNAMIC_QUERY_RESULT). The possible values correspond to the DDIC structure that we assigned to our SearchBookings BOL node in the GenIL model (ZBOL_SEARCHBOOKINGS_ATT). To execute the query, press the Find button above the query parameters. To see if there are entries in the DB tables that match the query parameters, check the results in the left panel, just below the list of search objects (Figure 2.16).
If you double-click on one of the result entries, you should be able to see all your attributes. Recall, however, that we removed the call to the `GET_OBJECTS` method in our query method implementation. For this reason, do not double-click on the found entries. At this point you cannot see much, because the framework shows only the GUID (an internal ID assigned by GenIL to each entity instance). Once we implement the mechanics to retrieve all the `Bookings` data, we will come back to the BOL browser and execute the query again.

We will be using the BOL browser frequently in this and following chapters. So we do not want to waste precious time and explain something that will become obvious as our sample implementation progresses. In general, the tool is quite intuitive, and you can experiment with it even now. For example, you can open the SAMPLE GenIL component and try to traverse it. The left-hand panels will contain the GenIL nodes, and as you double-click on the entries there, the right-hand side will present the details of each selection. To move from a parent object down to its children, you can click the `CHILDREN` button above the object details.

### 2.5 Retrieving a BOL Entity

The `IF_GENIL_APPL_INTLAY~GET_OBJECTS` method is probably one of the most complex methods in your GenIL classes. This is the place where you read the requested attributes for your root and any of the related objects. The method accepts the following parameters:

- `IT_REQUEST_OBJECTS` is a table with request object entries.
- `IV_ROOT_LIST` is a reference to the data container.

`IV_ROOT_LIST` comes pre-filled with objects. Because the GenIL objects are not represented by dedicated programming classes, they are passed around in containers. Container objects can be packaged in `container lists`. You need to traverse the objects in the container and provide them with whatever they request. This might seem a bit confusing at first. The container always starts with root or access objects.

There can be multiple root objects, but they are all of the same type. These objects can contain children following the object-node hierarchy as defined in the GenIL model. The leaves of the container-object hierarchy do not necessarily match the leaves in the model. The hierarchy ends as soon as an access object is reached. The reason for such a realization is that the calling application demands data from the GenIL component in chunks.

#### 2.5.1 BOL Splitting

The consumer splits the object model into containers of objects that always start with root or access objects. This procedure is called `BOL splitting`. You can actually control the level of splitting in the GenIL Model Editor in the advanced settings of the model node. You have three types of splitting:

- Split only at root objects
- Default (discussed above)
- No splitting

The container that is being split from the rest of the model contains access objects that are filled only with their key attributes. The keys are being retrieved and populated while processing the previous container (representing a higher level of the model hierarchy). Thus, each subsequent call to `GET_OBJECTS` passes in a container that starts with access objects that have their key attributes prefilled. The implementation has to fill in the rest of the attributes and continue traversing and populating the requested object hierarchy. Relations to other root or access objects appear as foreign relations (we will discuss these in Chapter 3). According to the above rules, a foreign relationship indicates a BOL split.

#### 2.5.2 Container Objects

The query will return a collection of `Bookings` that match the search criteria. This will be realized as a container list filled with container objects. The key attributes of these objects will be populated, so that if we have such a container list we can pass it to `GET_OBJECTS` and read all the data.

If you are implementing a transactional buffer, you should keep the splitting in mind. You might want to buffer your data into chunks, starting with the root/access object, storing the data related to the dependent objects and finishing with an access object. Be aware that you should not buffer query data (or at least be careful when doing so). However, you should buffer your data when the objects are locked and a transaction is occurring. We will talk about this later.
When you get the container in your GET_OBJECTS implementation, there are two main scenarios that you will be dealing with:

- The container contains only one root object. The key of this object is initial. This is an indication that you need to read all the objects belonging to this GenIL object type (BO) and fill the container with it. This functions as a placeholder indicating that a full read is required. Consider the keys of the parent (if available) and pull only the relevant data. For example, if you receive a UsedProduct with an initial key, you will take a look at the parent Bookings instance and read all the products belonging to this booking.

- The objects found in the container have valid keys. You will need to retrieve the data based on these keys and set the object’s attributes.

There is more to consider in the above two scenarios, as you will have to also consider the children relationships. Sometimes the related data is requested, and sometimes not. You need to check for this.

2.5.3 Implementing GET_OBJECTS

We will base the implementation of our GET_OBJECTS method on the data flow outlined in Figure 2.17. In this section we will implement the following data processing steps.

1. Retrieve an object from the received object list
2. Check if the object attributes are requested
3. If the attributes are requested, retrieve and set them on the current object
4. Check if the object relations are requested
5. If the relations are requested, retrieve the children from these relations. For each child repeat Steps 1 to 5.

Sometimes, depending on your model, you might find that you do not need to implement all these steps. We recommend, however, that you always consider the complete algorithm.

In order to implement the flow in Figure 2.17, you will have to work with interface IF_GENIL_CONTAINER_OBJECT. Every container object implements this interface.

Step 1 involves looping through the objects from the container list (IV_ROOT_LIST). In order to loop through a container object list, IF_GENIL_CONTAINER_OBJECTLIST interface exposes methods to get the first and the next container object in the list: GET_FIRST and GET_NEXT (among others). To check whether you should fill in object attributes (Step 2), call CHECK_ATTR_REQUESTED. For Step 3 (filling in attributes) you will use container object’s SET_ATTRIBUTES (or GET_ATTRIBUTES to read what is set already). CHECK_RELS_REQUESTED will tell you if you need to pull the related child objects (Step 4). GET_RELS_REQUESTED and GET_RELATION will allow you to implement Step 5.

From an interface point of view, GET_OBJECT receives an IF_GENIL_CONTAINER_OBJECTLIST (IV_ROOT_LIST), which contains IF_GENIL_CONTAINER_OBJECT. GET_RELATION returns an IF_GENIL_CONTAINER_OBJECTLIST for the children linked via the specified relationship (passed as an argument to the method).
It is time to look at some code. Go ahead and redefine `IF_GENIL_API\_INLAY\_GET\_OBJECTS` in your GenIL class. Listing 2.5 provides an exemplary redefinition.

```abap
DATA LV_ROOT TYPE REF TO IF_GENIL\_CONT\_ROOT\_OBJECT.
DATA LV\_BOOKING\_KEY TYPE ZBOL\_BOOKINGS\_KEY.
DATA LV\_BOOKING\_ATT TYPE ZBOL\_BOOKINGS\_ATT.
DATA LV\_TEMP\_ATT TYPE ZBOL\_BOOKINGS\_ATT.

LV_ROOT = IV_ROOT\_LIST->GET\_FIRST().
WHILE LV_ROOT IS BOUND.
  LV_ROOT->GET\_KEY( IMPORTING ES\_KEY = LV\_BOOKING\_KEY ).
  * Check if attributes should be read
    IF LV_ROOT->CHECK\_ATTR\_REQUESTED( ) = ABAP\_TRUE.
      ZCL\_BOOKINGS\_BACKEND\_API\_READ\_BOOKINGS(
        EXPORTING IS\_BOOK\_KEY = LV\_BOOKING\_KEY
        IMPORTING ES\_BOOK\_ATT = LV\_BOOKING\_ATT ).

  * Return the object only if it still exists
    IF LV\_BOOKING\_ATT IS NOT INITIAL.
      * Put attributes to the container
        LV\_ROOT->SET\_ATTRIBUTES( LV\_BOOKING\_ATT ).
      * You could set attribute properties like readonly.
      ENDIF.
    ENDIF.

  * check if dependent objects should be read
    IF LV\_ROOT->CHECK\_RELS\_REQUESTED( ) = ABAP\_TRUE.
      process the directly dependent objects
      PROCESS\_CHILDREN( IT\_REQUEST\_OBJECTS = IT\_REQUEST\_OBJECTS
          LV\_ROOT = LV\_ROOT ).
    ENDIF.

  "process foreign relations
  PROCESS\_FOREIGN( LV\_ROOT = LV\_ROOT ).
  continue with the loop
  LV_ROOT = LV\_ROOT\_LIST->GET\_NEXT( ).
ENDWHILE.
```

Listing 2.5 GET\_OBJECTS Implementation

In accordance with the first step of the data flow from Figure 2.17, we have to loop through the root container list.

```abap
LV_ROOT = IV_ROOT\_LIST->GET\_FIRST( ).
WHILE LV_ROOT IS BOUND.
```

Inside the loop, we will check whether we need to read the container object’s attributes, and if the answer is yes we will do so (Steps 2 and 3 from the GET\_OBJECTS data flow). However, we will not implement the logic that gets the data inside our GenIL component. We already agreed that we will try to assign such activities to a dedicated API that we refer to as the backend API. In this case, the robust backend logic takes place in `ZCL\_BOOKINGS\_BACKEND\_API\_READ\_BOOKINGS`. The code that implements Steps 2 and 3 inside GET\_OBJECTS is as follows:

```abap
IF LV_ROOT->CHECK\_ATTR\_REQUESTED( ) = ABAP\_TRUE.
  ZCL\_BOOKINGS\_BACKEND\_API\_READ\_BOOKINGS(
    EXPORTING IS\_BOOK\_KEY = LV\_BOOKING\_KEY
    IMPORTING ES\_BOOK\_ATT = LV\_BOOKING\_ATT ).

  * Return the object only if it still exists
    IF LV\_BOOKING\_ATT IS NOT INITIAL.
      * Put attributes to the container
        LV\_ROOT->SET\_ATTRIBUTES( LV\_BOOKING\_ATT ).
    ENDIF.
  * You could set attribute properties like readonly.
ENDIF.
```

In the above snippet, we pass the key to the backend API and get back the complete attribute structure of our Bookings BO. If the structure is not initial (that is if data already has been read), we will use it to set all the attributes of the Bookings object instance (represented by its container). We will discuss `ZCL\_BOOKINGS\_BACKEND\_API\_READ\_BOOKINGS` shortly.

`IF_GENIL\_CONT\_ROOT\_OBJECT\_CHECK\_ATTR\_REQUESTED` is part of the container’s API. You can deduce its function from its name; it returns TRUE if the consumer has requested the object’s attributes. Another helpful method of the container API is `SET\_ATTRIBUTES`. It takes an importing structure `IS\_ATTRIBUTES` and uses it to populate all the object’s attributes. With this last call, we successfully complete Step 3 of the GET\_OBJECTS data flow.

At this the point, you could set additional properties on the container object (for example mark some fields as read-only). For our sample implementation, we will keep the properties simple.

In Step 4, we need to check whether the related entities have been requested by the consumer (for example BOL). We will implement this step by checking whether
Introducing the GenIL Component

2

IF_GENIL_CONT_ROOT_OBJECT~CHECK_RELS_REQUESTED returns TRUE. If so, we need to retrieve the children of our container objects, as Step 5 from the data flow.

IF LV_ROOT->CHECK_RELS_REQUESTED( ) = ABAP_TRUE.
* process the directly dependent objects
  PROCESS_CHILDREN( IT_REQUEST_OBJECTS = IT_REQUEST_OBJECTS
                      IV_ROOT           = LV_ROOT ).
ENDIF.

Step 5 is implemented in a separate helper method of our GenIL container class, namely PROCESS_CHILDREN (it is not part of the GenIL API). In our data-flow diagram we have indicated that inside Step 5 we will need to repeat all the previously discussed steps, but for the child object. This is exactly what our PROCESS_CHILDREN method does. According to the above snippet we are passing to it the IT_REQUEST_OBJECTS and the current root object LV_ROOT. But before we go into the details let’s finalize our GET_OBJECTS. What is left is to continue looping through all the passed container objects and apply Steps 2 to 5 to each one.

LV_ROOT = IV_ROOT_LIST->GET_NEXT( ).

Finally we close the WHILE loop and complete the method implementation.

2.5.4 Bookings Backend API

It is time to go into the details of ZCL_BOOKINGS_BACKEND_API=>READ_BOOKINGS and the PROCESS_CHILDREN method of the component class.

Listing 2.6 shows the code of READ_BOOKINGS.

DATA LS_BOOK_ATT_N LIKE LINE OF GT_BUFFER_BOOKINGS.
FIELD-SYMBOLS <LINE> LIKE LINE OF GT_BUFFER_BOOKINGS.

IF IS_BOOK_KEY IS NOT INITIAL.
* Try to read from the buffer
  READ TABLE GT_BUFFER_BOOKINGS
  WITH KEY GUEST_ID = IS_BOOK_KEY-GUEST_ID
  HOTEL_NAME = IS_BOOK_KEY-HOTEL_NAME
  LASTCHECKIN = IS_BOOK_KEY-LASTCHECKIN
  ASSIGNING <LINE>.
ELSE.
* Late key assignment case
* Get created booking => Only one object creation possible at a time
  READ TABLE GT_BUFFER_BOOKINGS WITH KEY NEW = 'C'
ENDIF.

ASSIGNING <LINE>.

if sy-subrc <> 0.
  RETURN.
endif.

* Check if the read from the buffer was successful
IF SY-SUBRC = 0.
* If the object is new or modified, return from the buffer
  IF <LINE>-NEW EQ 'C' OR <LINE>-NEW EQ 'M'.
    MOVE-CORRESPONDING <LINE> TO ES_BOOK_ATT.
    RETURN.
  ENDIF.
ENDIF.

* Read from the DB
SELECT SINGLE * FROM zbooking_reserv INTO CORRESPONDING FIELDS OF ES_BOOK_ATT
  WHERE guest_id EQ IS_BOOK_KEY-guest_id
  AND hotel_name EQ IS_BOOK_KEY-hotel_name
  AND lastcheckin EQ IS_BOOK_KEY-lastcheckin.
CHECK SY-SUBRC IS INITIAL.
* return the data read
  MOVE-CORRESPONDING ES_BOOK_ATT TO LS_BOOK_ATT_N.

Listing 2.6 READ_BOOKINGS Method

For now, do not spend too much time on the first 26 lines of READ_BOOKINGS code. Their purpose will become crystal clear when we discuss the transaction behavior (LOCK_OBJECTS, SAVE_OBJECTS, CREATE_OBJECTS, and MODIFY_OBJECTS). Remember GET_OBJECTS is not only called when the consumer (BOL) explicitly requests to read business object data. At this point, you need only know that at the beginning of the method we are checking the GenIL buffer (our custom implementation) to see if the requested bookings entity is a new one or is being modified, but is still not persistent in the database. If we find matching data in the buffer (new or in the process of being modified), we return it from there (no database read). We buffer the data when we lock it before we start a new transaction. We will explain this topic further when we go through the data modification procedure.

In the read-business-data scenario, there will be no modifications and no new entities. Therefore, what is important for us is the database read. It is quite straightforward:
We use the object's key attributes to find a matching record via a select statement. Once we find one we return it via the export structure \texttt{ES\_BOOK\_ATT}:

\begin{verbatim}
SELECT SINGLE * FROM zbooking_reserv INTO CORRESPONDING FIELDS OF ES_BOOK_ATT
WHERE guest_id EQ IS_BOOK_KEY-guest_id
AND hotel_name EQ IS_BOOK_KEY-hotel_name
AND lastcheckin EQ IS_BOOK_KEY-lastcheckin.
CHECK SY-SUBRC IS INITIAL.
* return the data read
MOVE-CORRESPONDING ES_BOOK_ATT TO LS_BOOK_ATT_N.
\end{verbatim}

For now, ignore the rest of \texttt{READ\_BOOKINGS} and focus on the last snippet. The effect is that we return a fresh snapshot of the data in our database. We could have performed optimizations related to data caching but this is the safest bet when it comes to avoiding data inconsistencies.

### 2.5.5 Reading Dependent Object Data

Listing 2.7 shows the implementation of the \texttt{PROCESS\_CHILDREN} method (our helper method). Note that we will stick to Steps 2 to 5 from the \texttt{GET\_OBJECTS} data flow, but this time we will apply to it the children of the \texttt{Bookings} object, \texttt{UsedProducts}.

```
data: LV_CHILDREN type ref to IF_GENIL\_CONTAINER\_OBJECTLIST,
  LV_PRODUCT type ref to IF_GENIL\_CONTAINER\_OBJECT,
  LV_BOOKING type ref to IF_GENIL\_CONTAINER\_OBJECT,
  LV_NAME type CRMT\_EXT\_OBJ\_NAME,
  LT\_FOREIGN\_RELS type CRMT\_RELATION\_NAME\_TAB,
  LT\_REQ\_RELS type CRMT\_RELATION\_NAME\_TAB,
  LV\_PRODUCT\_KEY type ZBOL\_BOOK\_PRODUCTS\_KEY,
  LV\_PRODUCT\_ATT type ZBOL\_BOOK\_PRODUCTS\_ATT,
  LT\_PRODUCT\_KEYS type ZBOL\_BOOK\_PRODUCTS\_T\_KEY,
  LV\_BOOKING\_KEY type ZBOL\_BOOKINGS\_KEY.
field-symbols: <REL> type CRMT\_RELATION\_NAME.

* check which relations to dependent objects are requested
  LT\_REQ\_RELS = IV_ROOT\_GET\_RELATION( ).
  LOOP AT LT\_REQ\_RELS ASSIGNING <REL>.
```

\begin{verbatim}
CASE <REL>.
  WHEN 'BookingProductsRel'.
    * get the list of all directly dependent objects (children)
      LV\_CHILDREN = IV_ROOT\_GET\_RELATION( 
        IV\_RELATION\_NAME = <REL>,
        IV\_AS\_COPY = ABAP\_FALSE ).
    * Note: We set the IV\_AS\_COPY flag to FALSE so that our
      list gets updated if new children were created.
    * Loop over the list
      LV\_PRODUCT = LV\_CHILDREN\_GET\_FIRST( ).
      WHILE LV\_PRODUCT IS BOUND.
        * get the key of the child object
          CALL METHOD LV\_PRODUCT\_GET\_KEY
            IMPORTING
              ES\_KEY = LV\_PRODUCT\_KEY.
        * check if child's key is given
          IF LV\_PRODUCT\_KEY IS INITIAL.
            * no key was given, object is just place holder -> read all
              get the parent object and its key first
              LV\_BOOKING = LV\_PRODUCT\_GET\_PARENT( ).
              CALL METHOD LV\_BOOKING\_GET\_KEY
                IMPORTING
                  ES\_KEY = LV\_BOOKING\_KEY.
              CALL METHOD ZCL\_BOOKINGS\_BACKEND\_API\_READ\_PRODUCT\_KEYS( 
                EXPORTING
                  IS\_BOOKING\_KEY = LV\_BOOKING\_KEY,
                  ET\_PRODUCT\_KEYS = LT\_PRODUCT\_KEYS ).
            * create new objects for the other products (except the
              first) by copying the place holder
              LOOP AT LT\_PRODUCT\_KEYS INTO LV\_PRODUCT\_KEY FROM 2.
                TRY.
                  LV\_PRODUCT\_COPY\_SELF\_WITH\_STRUCTURE( 
                    IS\_OBJECT\_KEY = LV\_PRODUCT\_KEY ).
                CATCH CX\_CRM\_GENIL\_MODEL\_ERROR.
                  * this can only happen if the optional parameter IV\_RELATION\_NAME
                    was given
                  but did not fit to the model.
                  CALL CX\_CRM\_CIC\_DUPLICATE\_ENTRY.
                CATCH CX\_CRM\_CIC\_DUPLICATE\_ENTRY.
                  * this can only happen if the method was called on an
                    object which has a
                    1:1 relation to its parent and no other relation name
                    was given.
            ENDTRY.
        * return the data read
          MOVE-CORRESPONDING ES\_BOOK\_ATT TO LS\_BOOK\_ATT_N.
        ENDLOOP.
  END_CASE.
\end{verbatim}
Introducing the GenIL Component

ENDLOOP.

* set new key for place holder object from the first product
  READ TABLE LT_PRODUCTS_KEYS INTO LV_PRODUCT_KEY
  INDEX 1.
  IF SY-SUBRC = 0.
    TRY.
      LV_PRODUCT->SET_KEY( LV_PRODUCT_KEY ).
      CATCH CX_CRM_GENIL_DUPLICATE_KEY.
        * This happens if an object with the same key already exists in the container.
      ENDTRY.
      * now we have created an object for each found used product, but only the keys were set.
      * normally we would also set the attributes directly, but here we want to demonstrate
      * how to work with the automatically updated child list. So we proceed only
      * with the current product, because all new objects were appended to the list and will normally
      * processed later on.
      ENDIF.
    * proceed only if at least one product was found
    IF LV_PRODUCT_KEY IS NOT INITIAL.
      * check if attributes should be read
      IF LV_PRODUCT->CHECK_ATTR_REQUESTED( ) = ABAP_TRUE.
        CALL METHOD ZCL_BOOKINGS_BACKEND_API=>READ_PRODUCT
          EXPORTING IS_PRODUCT_KEY = LV_PRODUCT_KEY
          IMPORTING ES_PRODUCT_ATT = LV_PRODUCT_ATT).
        IF SY-SUBRC = 0.
          put attributes to the container
          LV_PRODUCT->SET_ATTRIBUTES( LV_PRODUCT_ATT ).
        ENDIF.
      ENDIF.
      * You could check if dependent objects should be read
      * but as we know that such do not exist we will stop here
      LV_PRODUCT = LV_CHILDREN->GET_NEXT( ).
  ENDIF.
  * If there were relationships to an access object we could have explored the FOREIGN relations.

Listing 2.7 PROCESS_CHILDREN Method

One BO can have many relations. With this in mind, we have to loop through the requested relations and assess them individually. In our case, there is only one relation, BookingProductsRel.

LT_REQ_RELS = IV_ROOT->GET_RELS_REQUESTED( ).
LOOP AT LT_REQ_RELS ASSIGNING <REL>.
  CASE <REL>.
    WHEN 'BookingProductsRel'.
      * (steps 1-5)
      ENDWHILE.
  ENDCASE.
ENDLOOP.

For each found relationship, you will need to implement specific logic. Therefore, it is best if you move that in a separate method/class. For our implementation, we will sacrifice best practices for the sake of simplicity and we will put all our code in one method. Once we find a BookingProductsRel we need to retrieve all the container objects associated with it and loop through them.

LV_CHILDREN = IV_ROOT->GET_RELATION( IV_RELATION_NAME = <REL>
  IV_AS_COPY  = ABAP_FALSE ).
* Note: WE set the IV_AS_COPY flag to FALSE so that our list
* gets updated if new children were created.
* loop over the list
LV_PRODUCT = LV_CHILDREN->GET_FIRST( ).
WHILE LV_PRODUCT IS BOUND.
  * (steps 2-5)
  ENDWHILE.

The above snippet corresponds to Step 1 of our data-flow algorithm. Next, we need to check whether the retrieved container objects (within UsedProducts) have key values. If the key is initial, then we need to read all the objects of this type. This is something missing from our data-flow diagram, but it is an important part of the
Introducing the GenIL Component

GenIL protocol. Passing an empty child container object (you can think of it as a dummy object)—as shown below—is the way to signal to the GenIL container that all the children associated with the requested relation will be read and returned.

```plaintext
CALL METHOD LV_PRODUCT->GET_KEY
  IMPORTING ES_KEY = LV_PRODUCT_KEY.
* check if child's key is given
IF LV_PRODUCT_KEY IS INITIAL.
  ... (read all children)
ENDIF.
```

Inside IF LV_PRODUCT_KEY IS INITIAL, we will refer to the parent and get its key. Once we have it, we will read the related UsedProducts keys and populate our container with objects containing those keys. In essence, we need to create a new UsedProduct instance (container object) for each key found.

```plaintext
LV_BOOKING = LV_PRODUCT->GET_PARENT( ).
CALL METHOD LV_BOOKING->GET_KEY
  IMPORTING ES_KEY = LV_BOOKING_KEY.
CALL METHOD ZCL_BOOKINGS_BACKEND_API=>READ_PRODUCT_KEYS(
  EXPORTING IS_BOOKING_KEY = LV_BOOKING_KEY
  IMPORTING ET_PRODUCT_KEYS = LT_PRODUCTS_KEYS ).
```

* create new objects for the other products (except the first) by copying the place holder
  LOOP AT LT_PRODUCTS_KEYS INTO LV_PRODUCT_KEY FROM 2.
  TRY.
    LV_PRODUCT->COPY_SELF_WITH_STRUCTURE( IS_OBJECT_KEY = LV_PRODUCT_KEY ).
    CATCH CX_CRM_GENIL_MODEL_ERROR.
    CATCH CX_CRM_CIC_DUPLICATE_ENTRY.
    ENDD TRY.
  ENDF LOOP.

* set new key for place holder object from the first partner
  READ TABLE LT_PRODUCTS_KEYS INTO LV_PRODUCT_KEY INDEX 1.
  IF SY-SUBRC = 0.
    TRY.
      LV_PRODUCT->SET_KEY( LV_PRODUCT_KEY ).
      CATCH CX_CRM_GENIL_DUPLICATE_KEY.
      ENDD TRY.
    ELSE.
      ... (handle error)
    ENDD IF.
```

None of this is as complicated as it might seem at this point. First, for the provided Bookings key, we read the product keys via the backend API (ZCL_BOOKINGS_BACKEND_API=>READ_PRODUCT_KEYS). We will look at this method later. For now, keep in mind that it returns a table of product keys (there could be many products for one booking). Then we loop through that table and create product container objects starting from the second entry. In the previous snippet, we created a new object by calling COPY_SELF_WITH_STRUCTURE on the LV/booking container object. By doing so, we replicated the whole container structure (its model) associated with the empty/dummy container object. The keys of this object might be INITIAL, but there could be relationships associated with the object and we want to replicate these in each object that we are about to create. This is why we started from the second product in the table, rather than from the first. Once done creating (via copying) new container objects, we will come back to the first (empty) object and set its key attributes. This is exactly what happens after the loop. We read the first key found and call SET_KEY on the first container object (variable LV_PRODUCT).

The previous code snippet contains several exception-processing statements. They are empty, but we want to show what you should expect from the container object interface. It is sufficient to say that our implementation will not lead to an exception. We encourage you to look at the code and the comments in Listing 2.7 for additional information.

If our model had been more complex, there might have been relations from UsedProducts to lower-level dependent objects. We would have to repeat the same process for them as well. For example, we could have done this for each newly created UsedProduct container object. But a freshly created object does not contain any links to its dependencies (by default). The only object that had information about the model beneath was that first “dummy”: UsedProducts. That is why we did not populate it immediately, but used it as a template to create all the other product container objects.

So far, we have created a container object for each key found in the products table. However no attributes were set. Normally, we would also retrieve and set all the attributes, rather than just the keys, but here we want to demonstrate how to work with the automatically updated child list and stick to our data flow from Figure 2.17.

We set the attributes of the current product in the same way that the other objects (if any) will be processed later on by the same loop.
IF LV_PRODUCT_KEY IS NOT INITIAL.
* check if attributes should be read
  IF LV_PRODUCT->CHECK_ATTR_REQUESTED( ) = ABAP_TRUE.
    CALL METHOD ZCL_BOOKINGS_BACKEND_API=>READ_PRODUCT(
      EXPORTING IS_PRODUCT_KEY = LV_PRODUCT_KEY
      IMPORTING ES_PRODUCT_ATT = LV_PRODUCT_ATT ).
    IF SY-SUBRC = 0.
* put attributes to the container
      LV_PRODUCT->SET_ATTRIBUTES( LV_PRODUCT_ATT ).
    ENDIF.
  ENDIF.
ENDIF.

The previous snippet completes the realization of Steps 2 and 3 from the GET_OBJECTS data flow. We rely on ZCL_BOOKINGS_BACKEND_API=>READ_PRODUCT to fetch the UsedProduct’s attributes based on the key values. We will discuss all the backend APIs very soon.

In our sample implementation, we will not check whether we need to read the children of UsedProducts (Step 4), as we know very well they do not exist. By following our code in GET_OBJECTS, however, it is easy to imagine what needs to be done. If the product had children objects and they were requested by the consumer, the corresponding relationships could be found in each product container object that we created earlier. This is guaranteed by the fact that we copied the empty (or also called placeholder) object with COPY_SELF_WITH_STRUCTURE.

Before we test our implementation, we will examine the backend API methods that we saw earlier. Let’s start with READ_PRODUCTS (see Listing 2.8).

```
DATA: LT_PROD TYPE TABLE OF ZBOOK_PRODUCTS,
  LS_PROD_ATT_N TYPE ZBOL_BOOK_PRODUCTS_ATT_N,
  DB_READ_FLAG TYPE ABAP_BOOL.

FIELD-SYMBOLS: <LINE> TYPE ZBOL_BOOK_PRODUCTS_ATT_N,
  <FS_PROD> LIKE LINE OF LT_PROD,
  <PROD_ATT> TYPE ZBOL_BOOK_PRODUCTS_ATT.

REFRESH ET_PROD_ATT.
* check if there is something in the buffer
READ TABLE GT_BUFFER_PRODUCTS
  WITH KEY GUEST_ID = IS_BOOK_KEY-GUEST_ID
  HOTEL_NAME = IS_BOOK_KEY-HOTEL_NAME
  LASTCHECKIN = IS_BOOK_KEY-LASTCHECKIN
  TRANSPORTING NO FIELDS.
  IF sy-subrc EQ 0.
* There is data in the buffer
    LOOP AT GT_BUFFER_PRODUCTS ASSIGNING <LINE>
      WHERE GUEST_ID = IS_BOOK_KEY-GUEST_ID
      AND HOTEL_NAME = IS_BOOK_KEY-HOTEL_NAME
      AND LASTCHECKIN = IS_BOOK_KEY-LASTCHECKIN
      AND NEW <> 'D'.
      APPEND INITIAL LINE TO ET_PROD_ATT ASSIGNING <PROD_ATT>.
      MOVE-CORRESPONDING <LINE> TO <PROD_ATT>.
    ENDLOOP.
  ELSE.
* Read from the DB
    SELECT * FROM ZBOOK_PRODUCTS INTO TABLE LT_PROD
      WHERE GUEST_ID EQ IS_BOOK_KEY-GUEST_ID
      AND HOTEL_NAME EQ IS_BOOK_KEY-HOTEL_NAME
      AND LASTCHECKIN EQ IS_BOOK_KEY-LASTCHECKIN.
      IF SY-SUBRC EQ 0.
        LOOP AT LT_PROD ASSIGNING <FS_PROD>.
          * We need to return the data read....
          APPEND INITIAL LINE TO ET_PROD_ATT ASSIGNING <PROD_ATT>.
          MOVE-CORRESPONDING <FS_PROD> TO <PROD_ATT>.
        ENDLOOP.
      ENDIF.
  ENDIF.
```

Listing 2.8 READ_PRODUCTS Backend API Method

As we did when reading Bookings, we first check our GenIL component buffer and then read from the database. The difference here is that the database read might augment (rather than simply replace) the data stored already in the buffer. You can easily imagine a scenario in which we have inserted a new product but have not yet made the data persistent. Such a new product will remain in the buffer until we commit the transaction. In this case, reading only from the database will not be sufficient as we also need the new entry from the buffer. Again, we ask you to keep in mind that GET_OBJECTS is used not just to read the data during queries but also
plays a role in transactions. We will talk about the buffering when we implement the transaction behavior.

For now, please focus on the `SELECT * FROM ZBOOK_PRODUCTS` statement, as it is the one relevant for simply displaying the data in a read-only mode. We are selecting all the products for the key of the parent booking and are appending each record to the exporting table.

```
SELECT * FROM ZBOOK_PRODUCTS INTO TABLE LT_PROD
WHERE GUEST_ID EQ IS_BOOK_KEY-GUEST_ID
AND HOTEL_NAME EQ IS_BOOK_KEY-HOTEL_NAME
AND LASTCHECKIN EQ IS_BOOK_KEY-LASTCHECKIN.
IF SY-SUBRC EQ 0.
  LOOP AT LT_PROD ASSIGNING <FS_PROD>.
    * We need to return the data read....
    APPEND INITIAL LINE TO ET_PROD_ATT ASSIGNING <PROD_ATT>.
    MOVE-CORRESPONDING <FS_PROD> TO <PROD_ATT>.
  ENDMETHOD.
  ENDIF.
```

You might be wondering why we are discussing `READ_PRODUCTS`, given that it is nowhere to be found in the `PROCESS_CHILDREN` method. Both `READ_PRODUCT_KEYS` and `READ_PRODUCT` rely on `GET_PRODUCTS` to do all the heavy lifting. For example, retrieving the product keys requires the code in Listing 2.9:

```
call method ZCL_BOOKINGS_BACKEND_API=>READ_PRODUCTS(
  EXPORTING IS_BOOK_KEY = IS_BOOKING_KEY
  IMPORTING ET_PROD_ATT = LT_PRODS )
LOOP AT LT_PRODS ASSIGNING <PROD_ATT>.
  MOVE-CORRESPONDING <PROD_ATT> TO LS_PROD_KEYS.
  INSERT LS_PROD_KEYS INTO TABLE ET_PRODUCT_KEYS.
ENDLOOP.
```

Listing 2.9 READ PRODUCT_KEYS Backend API Method

As we explained, you will normally pull all the attributes at one time, but we wanted to implement the data flow because sometimes the model you are coding against can be quite complex. That is why we split the product retrieval: reading the keys first and then reading the rest of the attributes. For simplicity, we are relying on `READ_PRODUCTS` in both scenarios.

Based on the prior listing, we can see that all `READ_PRODUCT` does is to get a matching entry from the internal table imported from `GET_PRODUCTS` and return it as a result (see Listing 2.10):

```
DATA: LT_PRODS TYPE ZBOL_BOOK_PRODUCTS_T_ATT,
  LS_BOOK_KEY TYPE ZBOL_BOOKINGS_KEY.
MOVE-CORRESPONDING IS_PRODUCT_KEY TO LS_BOOK_KEY.

CALL METHOD ZCL_BOOKINGS_BACKEND_API=>READ_PRODUCTS(
  EXPORTING IS_BOOK_KEY = LS_BOOK_KEY
  IMPORTING ET_PROD_ATT = LT_PRODS )

READ TABLE LT_PRODS
  WITH KEY GUEST_ID = IS_PRODUCT_KEY-GUEST_ID
  HOTEL_NAME = IS_PRODUCT_KEY-HOTEL_NAME
  LASTCHECKIN = IS_PRODUCT_KEY-LASTCHECKIN
  PRODUCT_ID = IS_PRODUCT_KEY-PRODUCT_ID
  USAGE_DATE = IS_PRODUCT_KEY-USAGE_DATE
  INTO ES_PRODUCT_ATT.
```

Listing 2.10 READ PRODUCT Backend API Method

### 2.5.6 Testing the Implementation

It is time to test our implementation in the BOL browser. Remember to add the call to the `GET_OBJECTS` to your query implementation in case you have removed it. Start the `GENIL_BOL_BROWSER` transaction and enter your component set (`ZBOOK` in our example). Then execute the `SearchBookings` query as explained in the previous section.

If you have placed a breakpoint in `GET_OBJECTS`, you see that the method is invoked and also that all the attributes of the found objects are requested and set by our implementation (`LV_ROOT->CHECK_ATTR_REQUESTED` is true). The BOL has its own buffer, so it would store the data. As a result, the next BO read will hit the buffer rather than the database.

In the List Browser section (that shows the search result) double-click on an entry. You should get the attributes as shown in Figure 2.18. Note that `GET_OBJECTS` will not be called, as the data is already in the BOL buffer.
In the button toolbar above the Object Browser (right-hand side) you can click on the Children button. This will refresh the left-hand side, and instead of the list of queries you will see the list of relationships pertinent to the selected object. In our example, Bookings has only one relationship: BookingProductsRel. Go ahead and double-click on it. GET_OBJECTS is called again. Not surprisingly, LV_ROOT->CHECK_ATTR_REQUESTED is false (the Bookings attributes were populated during the first call). However, LV_ROOT->CHECK_RELS_REQUESTED is true and PROCESS_CHILDREN is executed. As expected, we see that the requested relationship is Booking-ProductsRel. Inside the container from that relationship, there is only one child object and its key is initial (a placeholder for UsedProducts). Consequently, our implementation reads all the products for the current booking from the database and fills up the container list.

The BOL browser will show all the related UsedProducts in the List Browser below the relationships. If you double-click on one of them you will see a screen similar to Figure 2.19. GET_OBJECTS will not be called because the framework just read all the relevant products, set their attributes, and stored the result in the BOL buffer.

If you click on the Parent button in the Object Browser toolbar, the tool will show the attributes of the parent object. If you want to execute a new search, click on the New Search button at the very top of the screen.

We will continue with the editing of our objects in Chapter 3. For now, we encourage you to experiment with the BOL browser, debug and even modify the GET_OBJECTS method.

### 2.6 Handling Container Lists and Objects

We have introduced and discussed the container object, and we now need to formalize and summarize all the information. The BOL/GenIL uses the container object to pass and request business objects. Semantically, the business objects are represented via their GenIL model and together with their relations form an object hierarchy. The container API facilitates the traversal of such a hierarchy. Figure 2.20 shows a class diagram of the container objects.

The root objects are always placed in a container list object of type IF_GENIL_CONT_ROOT_OBJECTLIST. The other container list objects are of type IF_GENIL_CONT_OBJECTLIST. You can think of IF_GENIL_CONT_ROOT_OBJECTLIST as a specialization of IF_GENIL_CONTAINER_OBJECTLIST for root objects.
Introducing the GenIL Component

For example, both our query and GET_OBJECTS methods import IF_GENIL_CONT_ROOT_OBJECTLIST. This is normal. Remember that because of the BOL splitting, the requested object hierarchy always starts with a root (or access) object.

The most important methods of your container list objects are:

- GET_FIRST
- GET_NEXT
- GET_GLOBAL_MESSAGE_CONTAINER
- GET_LAST
- REWIND
- ADD_OBJECT
- SIZE
- ADD_CHILDREN

The GET* methods allow you to traverse the container list; they all return IF_GENIL_CONT_OBJECT. In our query implementation, we used GET_GLOBAL_MESSAGE_CONTAINER to store the messages resulting from the business logic processing. We also used ADD_OBJECT to create a new object in the container list.

Always remember that a container list object only can be empty (GET_FIRST is not bound) inside a query method. In any other case it will always contain something.

If a container list holds only one container object with an initial key, you should consider this an indication to retrieve all the valid objects (all the root objects or all the objects referred to by the current relationship).

Every container object exposes, among others, the following methods:

- GET_CHILDREN
- GET_PARENT
- GET_ROOT
- ADD_CHILD

GET_CHILDREN will allow you to access any related container list object. The rest of the methods are self-explanatory.

Before we conclude this chapter, we want to discuss one more topic, namely filling the container list with objects. In the previous sections, we used ADD_OBJECT and COPY_SELF_WITH_STRUCTURE. You can also use the ADD_CHILD method from the container object.

The key difference between COPY_SELF_WITH_STRUCTURE and the other methods is that it preserves the object hierarchy associated with the container object on which it is invoked. You will need to traverse the full hierarchy in order to understand what has been requested by the framework (which dependent objects to return, for example). Remember that we retrieved UsedProducts by evaluating the relationships of Bookings. By doing so we found out that there is one UsedProducts object in the container list from BookingProductsRel. The key of this UsedProducts was initial, and this made us pull all the UsedProducts related to the current Bookings parent object.

Now, let’s look at a more complex scenario. In the case of an order (sales or service), we typically have a header that contains items, which in turn could be linked to information about a ship-to party. Imagine a situation in which you receive a container list with one header object that has an item relation. The latter reveals a...
single item container object with initial key. Imagine also that from that dummy item, there is a relationship to a ship-to party. The party container object also has an initial key. Such a setup automatically means that you need to retrieve all the items along with their ship-to party data. Therefore, you would first read the keys of all the items and for each key you would create a new item container object. What happens if you add the new item to the item container list via ADD_OBJECT? Figure 2.21 depicts this scenario.

It is obvious that by using ADD_OBJECT (or ADD_CHILD) we will lose any related containers. As a result, we would never pull the ship-to party for the items we just added to the container (the ones with keys 2115 and 2116 in Figure 2.21). This is obviously not what we want. For this reason, use methods like ADD_OBJECT with care; for example, when populating a container list with leaves.

Let us look at the effects of using COPY_SELF_WITH_STRUCTURE (Figure 2.22 reflects the outcome of using the object with key initial as a template).

As one would expect, the container is populated with item objects that all point to a container list filled with placeholders of the ship-to-party type. As a result you can pull all the items and their related ship-to parties. Based on the above scenario, we recommend that your first choice for filling a container list is COPY_SELF_WITH_STRUCTURE.

In this chapter, we introduced two of the most challenging topics in this book, namely the BOL and GenIL layers. We saw that they can be used to implement and access the business logic of your application in a uniform way. The GenIL layer provides the implementation of the business logic, while the BOL layer is its consumer. The BOL/GenIL layers provide several services to the implementing applications, including message containers, transaction handling, and data buffering.

In this chapter we implemented querying and reading data: IF_GENIL_APPL_INTLAY~GET_DYNAMIC_QUERY_RESULT and IF_GENIL_APPL_INTLAY~GET_OBJECTS methods. Let’s recap what we did.

The code inside GET_DYNAMIC_QUERY_RESULT accepts query parameters and search attributes. Based on these we tried to find matching bookings records in the database. For each match, we retrieved the keys and used them to create Bookings objects. We added these newly created objects (container) to a container list object.
We then passed the container list object to the `GET_OBJECTS` to do the complete reading and populating of the container objects. The container list was returned to the consumer. This concluded the implementation of `GET_DYNAMIC_QUERY_RESULT`.

`GET_OBJECTS` accepts a container list object. Each object inside that list is a root object (container). For each root container we checked whether the consumer wants us to get all the attributes. If so, we pulled all the related entities. We got another container list, but this one was holding the children of the current root object. We looped through that list of child objects.

If a child container contained initial key attributes (place holder) we treated it as a signal that the consumer wants us to read all the children of that type (connected to the root container object via the same relation). We queried the database and pulled all the keys for the matching dependent objects. For each key, we created a new child container object and added it to the container list of the related entities (child list).

We then continued with our loop and for each child object we checked whether we had to read the attributes, and if so we populated the container. We could also have checked for any related entities, but our GenIL model hierarchy has only two levels and the `UsedProducts` are leaf nodes. This concluded our `GET_OBJECTS` implementation.

While we tested our code in the BOL browser we saw that `GET_OBJECTS` is called once within `GET_DYNAMIC_QUERY_RESULT` and called again when we requested the details of the child entities (`UsedProducts`). The rest of the time, the data came from the BOL buffer.

If your custom application has to support only a read-only scenario, then you are finished at this point. However, if you want to edit our data you will have to do more coding. This is what we will discuss in Chapter 3.

We saw how to define the GenIL model via the GenIL Model Editor (for static models). If you would rather take care of the model persistency and definition yourself, you can study some of the existing CRM standard models (for example `BT` or even `UIF_FL`). SAP SDN also offers materials on this subject. One read that is easy to follow is the Create Your Own BOL Object blog by Harel Gilor (http://www.sdn.sap.com/irj/scn/weblogs?blog=/pub/wlg/19914).
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>15</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>21</td>
</tr>
<tr>
<td>1 Introducing the Web Client UI Framework</td>
<td>23</td>
</tr>
<tr>
<td>1.1 Main Components of the Web Client UI</td>
<td>24</td>
</tr>
<tr>
<td>1.1.1 Header Area</td>
<td>25</td>
</tr>
<tr>
<td>1.1.2 Navigation Area</td>
<td>28</td>
</tr>
<tr>
<td>1.1.3 Entry Page</td>
<td>30</td>
</tr>
<tr>
<td>1.1.4 Search Page</td>
<td>36</td>
</tr>
<tr>
<td>1.1.5 Overview Page (OVP)</td>
<td>39</td>
</tr>
<tr>
<td>1.1.6 Assignment Blocks</td>
<td>41</td>
</tr>
<tr>
<td>1.1.7 Interaction Concept</td>
<td>42</td>
</tr>
<tr>
<td>1.2 Architecture</td>
<td>43</td>
</tr>
<tr>
<td>1.3 Tools in Web Client UI</td>
<td>46</td>
</tr>
<tr>
<td>1.3.1 Component Work Bench</td>
<td>48</td>
</tr>
<tr>
<td>1.3.2 Skins and Skins Work Bench</td>
<td>49</td>
</tr>
<tr>
<td>1.3.3 Configuration and Personalization</td>
<td>53</td>
</tr>
<tr>
<td>1.4 Extensibility</td>
<td>58</td>
</tr>
<tr>
<td>1.4.1 Overview</td>
<td>58</td>
</tr>
<tr>
<td>1.4.2 Design Layer</td>
<td>64</td>
</tr>
<tr>
<td>1.4.3 Web Service Tool</td>
<td>67</td>
</tr>
<tr>
<td>1.4.4 Groupware</td>
<td>68</td>
</tr>
<tr>
<td>1.4.5 Office and Adobe Integration</td>
<td>69</td>
</tr>
<tr>
<td>1.5 Enterprise Search</td>
<td>69</td>
</tr>
<tr>
<td>1.5.1 Overview</td>
<td>69</td>
</tr>
<tr>
<td>1.5.2 Search Modeling Workbench</td>
<td>71</td>
</tr>
<tr>
<td>1.6 Collaboration</td>
<td>72</td>
</tr>
<tr>
<td>1.6.1 RSS Feeds</td>
<td>72</td>
</tr>
<tr>
<td>1.6.2 Favorites and Tags</td>
<td>72</td>
</tr>
<tr>
<td>1.6.3 Central Sharing Tool</td>
<td>73</td>
</tr>
<tr>
<td>1.7 Sample Business Scenario</td>
<td>74</td>
</tr>
<tr>
<td>1.8 Summary</td>
<td>76</td>
</tr>
</tbody>
</table>
8.1.4 Modifying the Generated UI Component ...................... 388
8.2 Configuration and Personalization ................................. 397
8.3 Attaching a Design Layer Object ..................................... 402
8.3.1 Design Object Assignments ....................................... 403
8.3.2 Creating a Design Object .......................................... 404
8.3.3 Creating DL Assignment .......................................... 406
8.3.4 Testing the DL Assignment ...................................... 407
8.4 Reusing Other Components within a Pop-Up .................... 408
8.4.1 Analyzing the Reusable Component ............................. 409
8.4.2 Value Help Pop-up ................................................ 410
8.4.3 Analyzing the Results ............................................. 413
8.5 Summary .................................................................... 414

9 Enhancing Applications with Fields and Tables .................. 417
9.1 Easy Enhancement Workbench ........................................ 418
9.2 Application Enhancement Tool ....................................... 420
9.3 Creating New Fields and Tables with the Application Enhancement Tool ........................................... 423
9.3.1 Starting the Application Enhancement Tool (AET) .......... 424
9.3.2 Creating a New Field .............................................. 426
9.3.3 Creating New Tables with the AET ............................. 435
9.4 Field Types ................................................................ 440
9.5 Navigation Fields .......................................................... 442
9.6 Extending SAP CRM Interactive Reports ......................... 444
9.7 Extending the Design Layer .......................................... 451
9.8 Summary .................................................................... 455

10 Behavioral Extensibility ............................................... 457
10.1 Behavior Extensibility via BAdIs ................................. 457
10.2 Adding Logic to Table Enhancements .............................. 460
10.3 Creating a Calculated Field .......................................... 463
10.4 Introducing Global Attribute Tags .................................. 471
10.5 Adding Custom BRF+ Operations .................................. 475
10.6 Summary .................................................................... 480

11 Rapid Applications and Mash-ups ................................ 481
11.1 Rapid Applications ..................................................... 481
11.1.1 Managing Rapid Applications ................................. 482
11.1.2 Creating a Rapid Application .................................... 484
11.1.3 Using and Configuring Your Rapid Application ......... 492
11.1.4 Extending the Default Implementation ..................... 496
11.1.5 Web Service-Based Applications ............................. 500
11.2 Application Mash-ups .................................................. 504
11.2.1 Mash-ups Overview .............................................. 504
11.2.2 Creating a Mash-up ............................................... 504
11.2.3 Enabling and Using a Mash-up ................................. 507
11.3 Web Mash-ups ......................................................... 508
11.3.1 A Mash-up Example ............................................. 509
11.3.2 Adding a Web Mash-up ......................................... 509
11.3.3 Testing and Enabling a Web Mash-up ....................... 511
11.4 Summary .................................................................... 512

12 Web Service Tool .......................................................... 515
12.1 Using the Web Service Tool ......................................... 516
12.2 Enabling the Web Service Tool ..................................... 523
12.2.1 Implementing the GenIL Adaptations ....................... 524
12.2.2 Creating a Web Service in WST ............................... 525
12.3 Summary .................................................................... 528

13 SAP CRM Integration with Microsoft Word and Adobe .......... 529
13.1 Integration Scenarios ................................................... 530
13.2 Technologies Involved in Integration ............................ 532
13.3 Design Time and Run time .......................................... 533
13.3.1 Design Time ....................................................... 534
13.3.2 Run Time .......................................................... 535
13.4 Creating a Template ................................................... 536
13.5 Adobe Integration ...................................................... 545
13.5.1 Creating the Template in Adobe .............................. 546
13.5.2 When to Use Adobe ............................................. 551
13.6 Summary .................................................................... 552
## Index

**A**
- ABAP, 200, 515
- ABAP search help, 311
- Able view context node, 228
- Abstract GenIL objects, 86
- Abstract object, 86
- Access, 85
- Access object, 85
- Account management, 553
- AccountSearchResults, 562
- Action menu, 588, 589, 590, 592
- Action provider, 590, 591, 592
- ActionScript, 593
- Active enhancement set, 344
- ActiveX, 556
- Adding a new field to a DL object, 454
- Adding the target ID to the navigation profile, 366
- AdditionalCheckforModifiedOutlookItem, 562
- Add mash-up, 509
- ADD_OBJECT, 128
- Add object to changed object list, 150
- Add UI element, 379
- Administrator, 47
- Adobe, 529
- Adobe Flex, 593, 597
- Adobe integration, 69
- Adobe LiveCycle Designer, 545, 546, 550
- Adobe templates, 69
- ADS, 545
- Advanced search context node, 228
- Advanced search controller, 230
- Advanced search page, 270
- Advanced search value help, 488
- Advance Search Page, 37
- AET (Application Enhancement Tool), 59, 374, 420, 464
- as a standalone application, 424
- comparison with EEW, 420
- exit class, 476
- from an application, 426
- from the UI Configuration Tool, 424
- registry, 444
- Aggregation, 85
- AJAX areas, 596
- Alias
  - BOL alias, 377
- AlwaysShowConflict, 562
- Application API, 457
- Application controller, 229, 606
- Application details, 484
- Application exit class, 476
- Application mash-ups, 504
- Application reference, 442, 487
- Application references
  - navigation field, 442
- Appointments and tasks, 553, 554
- AppAttachment, 562
- Architecture, 43, 80
- Web Client UI, 80
- Assignment block, 41, 280, 288, 290, 292
- Assignment to design layer, 406
- Association, 85
- Attribute, 82
  - properties, 94
  - structure, 90, 94
- Attribute metadata, 149
- Attribute setter method, 392
- Attribute structure, 396
- Attribute tag, 471
- Authorization profile, 335
- Authorization role, 332, 335, 336
- Authorizations, 423, 445, 485
- Automatic Delta Handling, 609
- Automatic UI generation, 378
- AutoSyncConflictOption, 562
- Available fields, 269
- AXTREG, 430

**B**
- Backend API, 97, 98, 112, 141, 172
- BAdI, 59, 344, 458, 476, 499, 570
  - create implementation, 460
Index

CRMClien, 564
CRM Email Inbox, 35
CRM Enterprise Search Modeling Workbench, 71
CRM_GW_HACTIVITY_REQUEST, 570
CRM_GW_CONTACT_REQUEST, 572
CRM interactive reports, 444, 445
CRM main software components, 45
CRM Middleware, 75
CRM_OFFICE_INT_ENHANCEMENT, 544
CRM_OFFICE_TEMPLATE_BADI, 536
CRM_OI_WEBSERVICE_BADI, 544
CRMServer, 564
CRMV_WS_METHODS, 524, 525
Cross-component navigation, 297, 342, 361
CSS, 603, 604, 609
CT-DTD-SR, 538
CT_RETURN, 572
CT-WS-SR, 516
Custom business logic, 498
Custom configuration keys, 359
Custom controller, 223, 229, 270, 281, 284
Custom data type, 442
Custom includes, 419
Custom GenIL handler class, 177
Custom validation logic, 460

D
Database model, 485
Data binding, 200, 353, 579, 594
Data container, 102
Data flow of GET_OBJECTS, 109
Data loss pop-up, 297, 300
Data persistency logic, 160
Data sources, 481
Data type, 440
Data warehouse extractors, 430
DDIC search help, 315
Debugger, 369
Deep table view context node, 228
DEFAULT, 403
Default GenIL attribute values, 388
DEFAULT_TASK_CANCELED, 560
DEFAULT_TASK_COMPLETED, 560
DEFAULT_TASK_INPROCESS, 561
DEFAULT_TASK_OPEN, 561
DEFAULT_TASK_TEXTTYPE, 560
Design application properties, 375
Define a search page, 380
Define BOL model, 376
Define central search, 70
Define design objects, 65
Define UI model, 378
Delegate outbound plugs, 368, 370
Delete, 174, 564
DELETE_ACTIVITY_REQ, 570
Delete an enhancement set, 345
DELETE_OBJECTS, 168, 170, 171
DeleteRemainingAccounts, 564
Deleting of dependent objects, 168
Delta flag, 144
Delta handling, 604
Dependent object, 85, 93
data, 114
Deploy, 435
Dequeue function, 174
Descriptor object, 365, 366, 369
Design layer, 64, 589, 590
attribute, 454
attribute information, 454
object, 451
Design layer assignment, 66, 403, 451, 453
for context node, 453
in the Workbench, 453
Design object, 65, 403, 453, 475, 590, 591
Design time, 533
Direct link groups, 336
Disabled features, 304
DO(Handle)IN, 203
DL assignment, 406
DO_HANDLE_DATA, 202, 203, 204, 219
DO_HANDLE_EVENT, 203, 204, 219
DO_INIT, 202, 205, 208, 213, 217
DO_INIT_ATTRIBUTE, 205
DO_INITATTRIBUTES, 202
DOM, 603
DO_PREPARE_OUTPUT, 354
DO_REQUEST, 202, 203, 204, 209, 213, 217
DownloadContactsFromCRM, 564
Dropdown list, 307
Dropdown list block, 431
Dropdown list box, 407, 487, 494
Dropdown list values, 468
DTD, 224
Dynamic cross component navigation, 365
Dynamic models, 87
Dynamic navigation, 590
Dynamic navigation event, 367
dynamic query, 94, 97, 98
object, 85

E
Easy Enhancement Workbench, 59
Edit mode, 290, 291, 294, 310, 312
Edit page, 43
Edit view, 42
EEW, 373, 418
migration, 423
EEWB, 418
Embeddable search view, 505
Embeddable search, 379
Embeddable Object, 509
Embedded component, 367
Embedded search, 70
Embedded views, 484
page, 506
Embedding component, 367
Embedding view, 62
Embed rapid applications, 504
Embed view, 504
EmployeeSearchResults, 564
Empty line insertion, 494
EnableAutoSync, 564
Enable configuration mode, 426
EnableLog, 564
EnableSSOAuthentication, 565
Enhance applications, 379
Enhance a view, 352
Enhance component, 346
Enhance design layer objects, 451
Enhanced object, 428, 429
Enhancement, 58
Enhancement analyzer, 384, 385
Enhancement definitions, 344
Enhancement implementation, 460
Enhancement information, 348
Enhancement object, 428
Enhancement overview, 432
Enhancement place, 430, 431
Enhancement search page, 424
Enhancement set, 58, 344, 463
Enhancement set definition, 347
Enhancement spot, 344
Enhancement tool features, 61
Enhancing a window element, 348
Enqueue, 142
function, 137
Enterprise Portal, 44
Enterprise search, 69, 70
Enterprise Search Administration Cockpit, 71
Entity collections, 237
Entry pages, 28
Event handler, 265, 267, 291, 292, 293, 298,
306, 317, 318, 363, 412, 586, 590, 595
Excel, 529
Exchange skin logo, 52
Execute query, 105
Existing table, 485
Expert mode, 442
Extending middleware, 444
Extending the functionality offered by cGWI,
569
Extensibility, 58
concepts, 64
Extensibility registry, 430
Extension include, 430
External session ID, 606

F
F2 button, 347
F4 help, 310, 312, 338, 410, 413
Fact sheets, 55
Favorites, 72
Field details, 431, 437, 485
Field sub-type, 442
Field type property, 442
FILL_STRUCT_FROM_NVP_TAB, 141
Filler value, 461
Fixed value list, 487
Flash Island, 593, 594, 595, 596, 610
Logical link, 328, 329
Logical system, 598
LogonSyncEnable, 565
Logo text, 53
L-Shape, 286, 287, 301, 319, 323, 338, 490, 604

**M**

Main BOL entity, 474
Main BOL object, 406
main.do, 208, 215
Main entity, 348
main.htm, 208
Maintain attributes, 526
MainWindow.do, 256
Manage rapid applications, 483
Mandatory checkbox, 398
Mandatory fields, 493
Mapped logical system, 598
Mapping application attributes, 506
Mash-ups, 59
Web 2.0, 63
Mash-up script, 510
MaxNoOfContactToBeSynced, 565
MAX_SEARCH_HITS_ACCOUNT, 561
MAX_SEARCH_HITS_CONTACT, 561
MAX_SEARCH_HITS_EMPLOYEE, 561
MAX_SEARCH_HITS_REFDOC, 561
Message class, 171
Message container, 101, 171
Message service, 171
Messages in the BOL browser, 189
Microsoft Exchange Server, 556
Microsoft Outlook, 553
Microsoft Word, 529
MIME, 609
MIME content, 463
MIME repository, 594, 597
Model, 198, 201, 213, 218, 227, 230
Model editor, 90
Model node, 278
Model View Controller (MVC), 44, 197, 205, 222, 223, 233, 257, 609
Modifications

flag, 144
to child objects, 152
Modified attributes, 150, 151
MODIFY_ACTIVITY_REPLY_REQ, 570
MODIFY_ACTIVITY_REQ, 570
Modify database records, 166
MODIFY_OBJECTS, 133, 144, 146
in handlers, 179
Multiple profiles, 561
MVC design pattern, 198, 205, 222, 224, 232

**N**

Navigation, 281
Navigational event, 283
Navigational link, 263, 267, 284, 321
Navigation bar, 28
Navigation bar profile, 328, 331, 333, 337
Navigation field, 442
Navigation history, 387
Navigation link, 230, 363, 380, 490, 494
Navigation link field, 487
Navigation profile, 366, 491
Navigation target, 484
NetWeaver, 516
New assignment block, 440
New empty row, 397
New table, 487
NextDays, 565
No direct root deletion, 169
Note, 573
NUM, 543
NW Web Service Navigator, 521

**O**

Object action, 601
Object browser, 195
Object foreign relations, 193
Object parts, 428, 429
Object type, 539, 601
Office integration, 69, 529, 530
OLTP reporting, 445
onClick, 219, 220

One click action column, 494
One-click actions, 316, 388, 400
OCA, 400
OneNote, 529
Open/close navigation bar, 28
Operand, 475
Operation, 475
Outbound plug, 264, 283, 284, 365, 367, 410, 411
Outbound plug mappings, 331
Outlook, 529
Output processing, 203
Overview page, 39, 254, 279, 283, 284, 285, 289, 292, 296, 434
toolbar, 40
Overview page controller, 230

Produce a UI component, 373
Proxy, 523
Proxy class, 501

Q

Queries, 83
Query criteria, 382
Query handler, 176
Query name parameter, 104
Query object, 85, 185
Query operations, 519
Query result, 85
Query result object, 85
Query service, 235
Quick create links, 28
Quote, 532

R

Rapid applications, 59, 61, 460
Reading dependent object data, 114
Reading object data, 187
Recent items, 28
ReferenceDocSearchResultsController, 566
Regenerate enhancements, 385
Register a GenIL component, 183
Register handler classes, 177
Relations, 82, 93
Relationships, 82, 117
REL_NAME_MY_ACCOUNTS, 561
REL_NAME_MY_CONTACTS, 561
Remind, 566
Remove from the buffer, 159
Render/Validate As, 465
Reports, 36
Report search page, 446
Repository information system, 458
Repository.xml, 230, 257
Request processing, 265
Restart the session, 432
REST service, 508
result.do, 217
result.htm, 218
Result object, 85
If you've ever had any questions about working with SAP's interactive forms, this comprehensive book will be a valuable addition to your library. Whether you are a beginning or advanced technical consultant developer, or form designer, you will learn everything you need to know about working with SAP Interactive Forms by Adobe. The second edition is updated for SAP NetWeaver 7.20, and includes new coverage of ABAP Offline Infrastructure, XDC Editor, JobProfiles Editor, parallelization of print jobs, and more.

approx. 790 pp., 2. edition, 89,95 Euro / US$ 89.95

www.sap-press.com

Find everything you need to know in this comprehensive guide to creating forms in SAP
Learn how to solve real-life problems that occur when working with interactive forms
Expand your knowledge with new information on ABAP Offline Infrastructure, XDC Editor, UI Configuration Tool, 356, 467, 585 UI framework session, 606 UI object type, 65, 261, 287, 319, 404, 406, 471, 474 UI runtime, 606 Update modified attributes, 157 URL, 559 User profile attributes, 475 Utility method, 394

V
Value context node, 463 Value help, 486 Value help From ABAP Dictionary, 405 Value help pop-up, 408 Value nodes, 271 V-getter, 411 View, 199, 264 View configuration, 398, 433, 443, 445, 495 pop-up, 427 View controller, 229, 595 View details, 380 View group context, 290, 292 View object, 86 View objects GenML, 86 View set, 258, 324, 587 View sets, 227, 230 View steps repository, 577 View variants, 400 Visible rows before paging, 269 Visible rows before scrolling, 269

W

X
XBCML, 594 XML, 530, 609 XML Schema, 547 XSD, 532, 534, 547 XUL, 533

Z
ZBT_CTR_QUOTE_READ, 536
Tzanko Stefanov is a director of product management at SAP. Previously he worked in several different leadership roles in the areas of SAP NetWeaver and SAP CRM.

Armand Sezikeye is a Development Manager for the Web UI Tag Library implementation team. Over the last 5 years he has acquired extensive know-how in various aspects of Web Client UI.

Sanjeet Mall is a chief architect for Mobile Applications in the Mobile Application Unit at SAP. Previously, he worked in the SAP CRM area at SAP.

Tzanko Stefanov, Armand Sezikeye, Sanjeet Mall

SAP Web Client

624 Pages, 2011, $79.95

www.sap-press.com/2847

We hope you have enjoyed this reading sample. You may recommend or pass it on to others, but only in its entirety, including all pages. This reading sample and all its parts are protected by copyright law. All usage and exploitation rights are reserved by the author and the publisher.