Effective Web Dynpro - Adaptive RFC Models

Bertram Ganz, NWF Web Dynpro Foundation for Java

Overview

- In many Web Dynpro applications, backend access is based on RFC modules in SAP systems. The Web Dynpro for Java Foundation provides a cutting-edge technology for accessing RFCs in the *Adaptive RFC Adapter.*

This article reviews some basic principles and guidelines ("do's" and "don'ts") for the effective use of Adaptive RFC models in Web Dynpro. Starting with a general description of adaptive RFC features and benefits, two important performance issues related to RFC parameters are discussed. Another important topic is the relationship between RFCs, Adaptive RFC models, and Web Dynpro development components. We will provide some rules of thumb for an optimal partitioning of these entities inside your Web Dynpro application. Finally, the article describes restrictions when utilizing the adaptability features of an Adaptive RFC adapter.

Contents

Keywords

Adaptive RFC Adapter, Adaptive RFC models, RFC, JCo, performance, export-parameters, import-parameters, table-parameters, deployment granularity of Adaptive RFC models, adaptability, adaptability restrictions, closed namespace

Icons in Body Text

The do's and don'ts are divided into two categories: *performance relevant* and *general* recommendations. Each category is marked with a different icon.

Icon	Meaning
◔	Performance relevant recommendation
●	General recommendation

Related Tutorials

Some specific Web Dynpro tutorials dealing with Adaptive RFC models can be found on the *SAP Developer Network (SDN)* at http://sdn.sap.com. The tutorials and project ZIP files are available for download under the category *Home* → *Developer Areas* → *Web Application Server* → *Web Dynpro*.

- Accessing SAP Backend in Web Dynpro
- Handling Transactions with BAPIs in Web Dynpro

1. Recommended Do's

Do (1): Apply Adaptive RFC Models for Accessing RFC Modules in SAP Systems

With the *Adaptive RFC Adapter,* the Web Dynpro for Java Foundation provides its own technology for accessing Remote Function Call (RFC) modules that are exposed in an SAP system.

SAP strongly recommends using the *Adaptive RFC Adapter* instead of implementing another technique for accessing RFCs based on the JCo communication layer. The Adaptive RFC Adapter yields the following benefits:

- **Low Memory Consumption:** The Adaptive RFC Adapter was optimized for typical user interface scenarios. Therefore, even if large amounts of data are returned by an RFC call, only the data directly required by and displayed on the user interface is unmarshalled, which yields lower memory consumption.

  In this respect you should obey the following rule, which is explained at the end of this article: Don't (3): Do not implement data-intensive operations on the UI layer.
- **Automatic Model Generation**: The Adaptive RFC Model importer provided by the Web Dynpro tools automates the generation of a complete model access layer based on a given RFC module. This access layer allows the calling of any RFC from within Web Dynpro with minimal hand-written code. In some cases, no hand-written code is necessary at all because wizards exist for generating the necessary code automatically.
- **Dictionary Integration:** The Adaptive RFC Adapter is a *dictionary*-based interface. At design time, all structures, fields, and simple data types used by the imported RFC module are replicated in a *logical dictionary.* The generated model classes can be bound to structures in the logical dictionary so that the model attributes represent structure fields or model class properties can reference dictionary simple types.

  At design time, the snapshot of all dictionary types belonging to an Adaptive RFC model can be used for various declarations, like binding context nodes to dictionary structures (*context-to-structure binding*) or typing context attributes using dictionary simple types.

  At runtime, the model dictionary uses a provider infrastructure to get all types of information online from the SAP backend system. This runtime access of dictionary metadata provides the basis for type-specific services, like displaying *field labels*, *column headings*, *tool tips*, populating *value sets;* or, formatting *lengths*, *currencies,* and *units*).
- **Adaptability:** Based on the *Java Dictionary Runtime,* the Adaptive RFC Adapter can handle the arrival of new fields within an existing structure at runtime. It can adapt to modifications such as the addition of new fields (*extension fields*) to *append* structures in standard SAP tables or changes of *field lengths*, *label texts,* and *value sets*. There is no application coding required for getting all dictionary-based runtime services.

  The user interface support of the Adaptive RFC Adapter is not yet adaptive, but dynamic. This means that the application developer has to manage the visibility of extension fields based on a dynamic view layout modification.
- **Destination Configuration:** The Adaptive RFC Adapter provides a mechanism for mapping logical system names defined at design time to physical systems not known until runtime. This includes some advanced mechanisms, e.g. for mapping a logical system to a specific physical system on a per-user, per-role, or per-session basis. This allows integrating Adaptive RFC-based applications in many different system landscapes.

- **Model Reimport:** The *model reimport* functionality provided by the Web Dynpro tools allows re-synchronizing a model to the current RFC module version in the backend. New fields, structures, and datatypes are then also available at design time.
- **Value-Added Features:** The Adaptive RFC Adapter provides many value-added features for accessing RFC modules from within your Web Dynpro application. Besides dictionary integration it includes a sophisticated connection and lifecycle management, automated data transport, backend type conversions, and generic services based on the *Common Model Interface API* (*CMI*) or the Web Dynpro Runtime API like the `WDCopyService`, `IWDModelClassChangeTracking` or `IWDModelRegistry`.

Do (2): Put as Many RFCs in a Single Adaptive RFC Model as Possible

Do: Principally, you should *put as many RFCs in a single Adaptive RFC Model as possible.*

Consider this rule of thumb with regard to using Adaptive RFC Models in Web Dynpro for the following reasons:

1. **Closed Namespace**: A model is imported into a closed namespace; therefore, any commonly used datatypes or structures can be shared only within a single model. The term *closed namespace* means that the definition of dependencies between Adaptive RFC models is not supported. In this respect you should strictly adhere to the rule that two models must be put into two different namespaces or Java packages.

   The best example for this restriction is the *Return* structure of BAPIs (Bapiret2). If you import two RFCs (BAPIs) in different models, then you will receive two different classes representing Bapiret2. You can therefore no longer build any common functionality for parsing and interpreting the Return value of multiple BAPIs. You cannot share references, which means you cannot pass data by reference from one RFC to another. Instead, you would be forced to copy, e.g. via the `WDCopyService`.

2. **Connection**: A model represents a number of services that are required at runtime. The most important one is the connection (`JCO.Client`). If you import two RFCs in two different models, then each model will require, by default, a connection to be assigned at runtime. In most cases though, you might want to share the connection between multiple RFC calls, therefore it is wise to keep those RFCs in a single model.

   It is possible to share a connection between multiple models (see Method `IWDDynamicRfcModel.setConnectionProvider()`), but it is still easier if you are not forced to do this.

3. **Reimport**: Reimport functionality exists, so you are able to update an existing model and add further RFCs as needed. In the past, this was often the reason for using multiple models. This problem no longer exists.

Use Cases for Separate Models

You may still wish to separate models, and there are use cases for this. Here are some situations in which you would do this:

1. **Different Backend Systems**: You need to access different RFCs in different backend systems, e.g. RFC A accesses System A and RFC B accesses System B. This requires using different connections; therefore, you *must* use different models.
2. **Different RFC Lifecycles**: E.g. RFC A belongs to a different business package than RFC B. This could lead to updates in RFC A that would not affect RFC B. If this is known at development time, then it is wise to also keep these RFCs independent of each other by

importing them into different models. This is usually only the case in larger development projects involving many developers.

3. **Componentization**: If you make strong use of Web Dynpro components that are contained in different DCs, then you may also want to split up RFC models to prevent them from getting too big, and to avoid unnecessary dependencies. E.g. Web Dynpro Component A (part of DC A) uses RFC A and Web Dynpro Component B (part of DC B) uses RFC B. But the two components have nothing to do with each other. In this case, you have the option of either importing a single common model and placing it in a different DC (lets say DC X) to prevent DC A and DC B from getting dependent on each other. Alternatively, you can import two different models and place each model in the DC in which it is used: RFC A is imported into Model A and placed in DC A. RFC B is imported into Model B and placed in DC B.

Do (3): Place Web Dynpro Models into Separate DCs for Optimizing the Development Performance

For optimizing the performance of a single edit-deploy-run cycle, the deployment granularity plays a decisive role. Consider the following rule of thumb regarding the separation of Web Dynpro models.

Do: Principally, Web Dynpro models should be separated into separate (model) DCs (one model per DC). These models can be referenced in other Web Dynpro components based on a defined DC usage.

It is a better strategy to separate models into another DC and expose them as public parts. Typically, models contain classes that do not change during development, especially if the interfaces have been defined thoroughly. Since it is optional to build dependant components during design and build time, the development component that holds the models can be excluded from the build, thereby reducing build time as well as deploy time.

Since Different DCs are not allowed to share the same namespace, placing each model in a different DC enforces the following rule:

Do: Put each model in its own namespace or Java package.

Do (4): Consider Given Restrictions when Utilizing Adaptability

When utilizing the adaptability features of an Adaptive RFC model, you have to consider some restrictions in SAP NetWeaver ‘04.

Custom Fields Do Not Show Up in the Context Node at Design Time

When adding custom fields, NO new code is generated. Therefore, these new fields are not accessible via typed APIs. But they are available at runtime using generic APIs. The following code can therefore be used for accessing a new field called *CustomField*.

```
wdContext.currentRFCStructureElement().getAttributeValue("CustomField");
```

For accessing runtime information about all fields in a given dictionary structure, you can access the `IStructure`-API:

```
IWDNodeInfo nodeInfo = wdContext.nodeAddressData().getNodeInfo();
IStructure structure = nodeInfo.getStructureType();
```

Adding New Tables or Complex Structures

It is *not* possible in SAP NetWeaver '04 to add new complex structures or tables as custom extensions, and also be able to access these via Adaptive RFC. Only existing structures and tables used within your RFC function modules can be extended (with scalar fields). Doing this will possibly cause runtime errors and may also lead to inconsistencies in your application, because the additional data (customer extensions) will not be available.

Adding New Scalar Fields Directly as RFC Input or Output Parameters

Due to a restriction in the RFC layer, it was not possible to dynamically identify new scalar parameters added *directly* to the input/output section of an RFC function module. Therefore, adding fields can only be done if the structure was defined using a dictionary structure. You can identify a model class, which can be extended by the customer, by checking if the model class has a dictionary structure binding. Simply open (double-click) the model class of your choice, and in the *Overview* tab you will see whether a dictionary structure binding exists. Notice that *Input* and *Output* model classes DO NOT have such a structure binding. Therefore, adding new fields to these structures will not work.

It is not wise to directly manipulate the input and output structures of RFCs directly anyway, as this will lead to merge conflicts. By design, SAP suggests using *Append Structures* in combination with BAPIs for doing custom extensions. Changes in Append Structures are fully supported by the Adaptive RFC framework.

2. Recommended Don'ts

Don't (1): Do not Transfer Table Values as Export or Import Parameters

In contrast to the recommended RFC design, the transmission of tables via RFC in import/export or changing parameters (which should never be used) is not recommended. In this case, tables are transported marshalled in XML as *blob* (binary large object). The transportation of XML is performance as well as memory intensive. The problem is based in RFC-communication and there is no generic performance optimization for this issue available in the current release.

The solution for the problem is easy and not very work-intensive:

Don't: Do not transfer table values as export or import parameters.

Do: Change your Adaptive RFC interface so that all table values are transferred as table parameters and not within export/import/changing parameters.

A *model-reimport* is only needed at design-time when a table that was defined as an import or export parameter before is then defined as a table parameter. In this case, some model classes must be extended by an additional relation.

The described solution is only recommended if the SAP-datatype **`XString`** and complex structures are not used within the table, otherwise no solution exists yet. So please be aware of this known limitation in the design phase of your project and consequently use, if possible, a length-limited datatype, like **`char[10]`** instead of a **`XString`**.

Don't (2): Do Not Embed Tables within RFC Import, Export or Table Parameters

In addition to the problem with tables in export, input, or changing parameters, the existence of **complex structures** or **tables in tables** results in a performance-critical overhead of XML metadata to be transferred. The reason is that the RFC implementation can not calculate the complete length of these structures. Consequently, all kinds of metadata for each row and column is transferred via XML: the amount of data for these tables is very large, which can possibly lead to a substantial decrease in performance.

Don't: Do not embed, or at least try to avoid embedding tables within RFC import, export, or table parameters. Instead flatten all complex structures in the given RFC parameters.

The recommended solution to the problem can only be seen as a workaround because it often results in time consuming reimplementations and adaptations on the client side (Web Dynpro Adaptive RFC model generation, context-to-model-binding).

Don't (3): Do Not Implement Data-Intensive Operations on the UI Layer

As already mentioned, the implementation of the Adaptive RFC adapter was optimized for typical user interface scenarios. Based on the defined binding mechanisms between the UI and the Adaptive RFC model (*data binding*, *context mapping*, and *context-to-model binding*) only the data directly required by and displayed on the user interface is unmarshalled, which yields lower memory consumption.

Because the unmarshalling process of RFC-based data can be quite memory intensive, it is important to reduce it to a minimum. But this feature can be undermined if data-intensive operations like filtering or sorting take place on the UI layer (within Web Dynpro). Therefore you should obey the following rules:

Don't: Do not implement data-intensive operations on the UI layer, for example sorting and filtering of context model node elements.

Do: Instead, these data-intensive operations (such as sorting, filtering) should already take place in the backend implementation (the RFCs) itself.

Don't (4): Do Not Put Different Models into the Same Java Package

Don't: Do not put different models into the same Java package. Instead, every model should reside in its own separate namespace.

The explanation of this recommendation is given in the following two sections:

- Do (2): Put as Many RFCs in a Single Adaptive RFC Model as Possible
- Do (3): Place Web Dynpro Models into Separate DCs for Optimizing the Development Performance

Acknowledgements

Many thanks to Markus Cherdron, Frank Weigel and Jörg Singler for their valuable feedback and helpful suggestions on the given recommendations.