

Data Management Request Processing An MVC Pattern Language

White Paper produced by
Mphasis Corporation

April 2005
Version 1.0

© Mphasis. All rights reserved

Document Contact Information

Name	Bert Hooyman, Senior Architect Mphasis UK Ltd. Southbank House Black Prince Road London SE1 7SJ, United Kingdom
Email	bert.hooyman@mphasis.com
Business Phone	+31 618 780 559
Business Fax	+31 302 967 801

Table 1 **Contact Details**

Note From the Author

The request processing pattern described in this white paper is the result of exposure to numerous recurrences of the same data management requirements. We believe the pattern language covers all the essentials of data management. Having said that, we find that reality is always more complex than we are led to believe. Many of these 'reality checks' have found their way in this white paper. We invite all readers to actively contribute additional reality checks and contradictory findings, with the purpose of further enhancing the quality of this work.

Acknowledgements

I have enjoyed the valuable feedback from the following Mphasis engineers: Niranjan Sathe and Mohan Borwankar. I thank them both for helping put this white paper together.

Contents

Document Contact Information	i
Note From the Author	i
Acknowledgements	i
0 Executive Summary	1
Audience	1
Objectives	1
Prerequisites	1
0 Request Processing Patterns for Data Management	2
0 Manage a Whatever	3
0.0 Use Cases	3
0.0 Page Flows	3
0.0 Command Language	4
0.0 Repository Access	6
0 Common Processing Aspects	7
0.0 Processing Context	7
0.0 Data Exchange with Repository	8
0.0 Data Validation & Error Handling	8
0.0 Access Control	8
0.0 Dealing with Empty Result Sets	9
0.0 Dealing with Exceptions	9
0 Request Processors for Data Management	10
0.0 New Whatever	10
0.0 Copy Whatever	10
0.0 Update Whatever	11
0.0 Save Whatever	12
0.0 Cancel	12
0.0 Delete Whatever	13
0.0 View Whatever	14
0.0 List Whatever	14
0.0 Sort Whatever	15
0.0 Page Whatever	16
0.0 Search Whatever	16
0.0 Find Whatever	17
0.0 Filter Whatever	17
0.0 Match Attribute	18
0.0 Service a Whatever & Service Whatever	18
0 Extensions	20
0.0 Multi-Page Details & Edit	20
0.0 Interaction with Tree Viewers	20
0.0 A Service Layer	21
0.0 Customize Whatever Summary	21

0	Conclusions.....	22
Appendix -	References.....	23

1 Executive Summary

This MphasiS white paper addresses the management of data in web-enabled business applications. Data management features are typically light on behaviour and heavy on data. They are found in almost all web-enabled applications and in many cases a major part of the application is indeed dedicated to data management.

In an earlier whitepaper (see ref. [1]), a requirements analysis and generally applicable use cases were presented. These use cases together describe a pattern which is known as “Manage a Whatever”. In a subsequent whitepaper (see ref. [2]), *canonical web pages* and a *page flow pattern* have been identified that apply to Manage a Whatever. A recurring pattern of *user requests* emerges from the web page flows.

In this white paper, the processing of these user requests is further analyzed with the purpose of identifying a generic design for request processors in the data management domain.

Audience

This white paper is written for MphasiS clients. It describes a pattern language for defining the request processors of data management tasks. Using the pattern language, MphasiS architects and development leads can roll out streams of web-enabled data management modules for their client projects in an efficient manner.

Objectives

The goal of this white paper is to bring a practical implementation of reusable data management code closer to reality. The request processor design that is elaborated here is not directly tied to any technology, although it is clearly based on an object-oriented style of software design.

Prerequisites

This white paper assumes a basic understanding of the Manage a Whatever Use Cases as well as some familiarity with the page flows and user requests resulting from that pattern. References [1] and [2] provide the relevant details.

In this white paper, we assume a data access tier is available with an interface that follows the pattern described in ref. [3]. This is the *repository access pattern* that has been designed to co-exist with the request processing pattern.

2 Request Processing Patterns for Data Management

In web-enabled applications, the classical Model-View-Controller pattern or MVC is a well-established design pattern for the processing of *user requests*.

The responsibility of a request processor is to execute the business logic for serving the request, handle errors and navigate to an appropriate view based on the outcome of the action. For data management applications, many request processors will have to interact with a data access tier to retrieve or store data.

The Information Architecture for Data Management applications (see ref. [2]) has identified fifteen fundamental user requests that are typically used in data management applications. Consequently, fifteen distinct request processors will be needed to service these user requests.

In this white paper, reference is made to a *repository access pattern* that is described in another white paper in this series (see ref. [3]). This pattern describes the access methods and data structures typically needed to support efficient data access.

The aim of this white paper is to identify the business logic pattern, i.e. the steps taken by each controller as parts of its action. This includes how a request processor and a repository cooperate and how the request processor selects an appropriate view based on the outcome of its action. These views are web pages from the page flow pattern (ref. [2]).

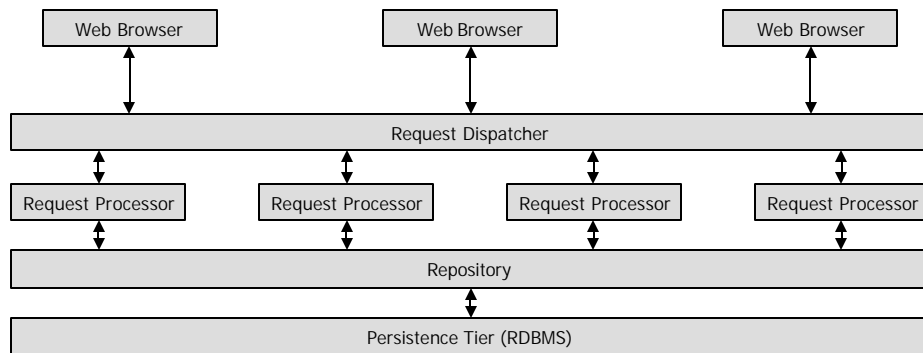


Figure 1 Generic System Diagram for Web-enabled Data Management

3 Manage a Whatever

As a quick summary, the core elements of the Manage a Whatever pattern language are repeated here.

3.1 Use Cases

Use cases for data management focus on the basic CRUD activities; Create, Retrieve, Update and Delete. The notion of a *Whatever* is introduced to indicate that this pattern of use cases applies to *any* business object. The pattern is not intended to be applied at the level of individual database tables. Instead, it is expected that the pattern is applied to compound data models with rich associations.

Create a Whatever and **Copy a Whatever** are the two basic use cases to create new instances of a Whatever; **Update a Whatever** and **Delete a Whatever** complement these to form the C, U and D of CRUD.

To retrieve whatevers, users may **Find a Whatever**, **Filter Whatever** or **List Whatever**, and drill down to a single Whatever using **View Whatever Details**. Users can scroll through subsequent Whatever of a list using **Next Whatever** and sort lists of Whatever using **Sort Whatever**.

Other domain-specific operations that apply to Whatever (e.g. Flag Emails, Validate Addresses, Export Data...) are all covered by the **Service a Whatever** use case, which is the extension point for the pattern.

3.2 Page Flows

Web pages for data management can be very complex with a multitude of features on each page, or very simple with only a single feature per page, or any level in between. In any case, the basic structure is the same.

Create, Copy and Update a Whatever share the **Edit Page** which is a data-entry form. Find and Filter results are displayed on either the **List Page** or the **View Details Page**, depending on the number of matching Whatever. The Find a Whatever use case requires a **Search Page**, which is another data-entry form. Filter Whatever uses a **Filter Page** which uses selection menus rather than data entry to steer the user's task. These five pages are sufficient to implement all the use cases of Manage a Whatever. Use cases not directly supported by a page find their way into the pattern as *commands*. These are realized either as buttons on the page(s) or as hyperlinked elements of the page(s).

The flow between the pattern pages is shown in Figure 2 below. This diagram also includes the *commands* or *user requests* that lead from page to page; these are described in more detail in section 3.3.

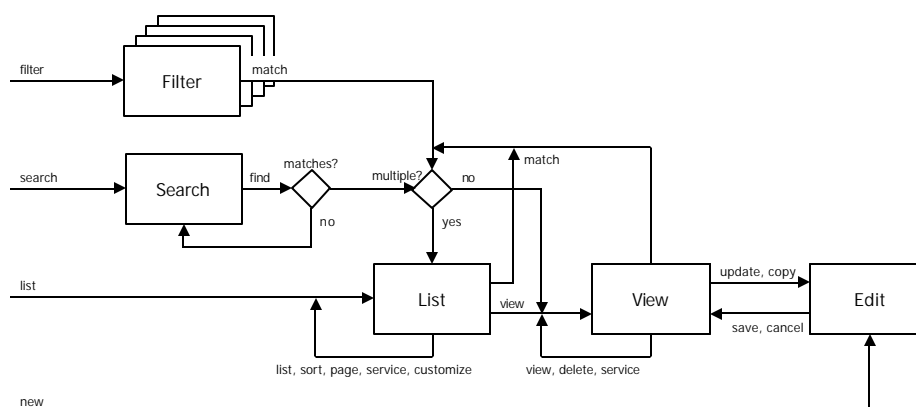


Figure 2 Standard Page Flow for Manage a Whatever (includes user requests)

3.3 Command Language

The command language of Manage a Whatever gets materialized through the user requests that flow from the web pages to an application server. These user requests are the entry points for the request processors described in this white paper.

All pages of the pattern typically provide a navigational link to the Search Page, the List Page and the Filter Page. Command buttons are one way to realize these links; **Find Whatever**, **List Whatever** and **Filter Whatever**. Another globally accessible link leads to the Edit Page for the creation of a new Whatever, in case this is a button it is typically labeled as **New Whatever**.

On the List Page, summary information of Whatsers is displayed in a tabular format. The column headers will often be used to direct a sort order; these are hyper-linked headers that trigger the **Sort Whatsers** use case.

Long lists of Whatsers are typically displayed using some form of pagination; a subset of matching entities is displayed on each page. Additional navigation support is provided to let the user move from one page to another. Many different interaction styles for pagination are known, including **Prev/Next/First/Last** command buttons and hyperlinked page number sequences. Pagination is dealt with as part of the List Whatsers use case.

To drill down from the List Page to the **View Details** Page, one or more attributes in the table rows may be hyperlinked. Candidate attributes for hyper-linking are those that uniquely identify an instance of a Whatever. Attributes that are non-unique may also be hyperlinked; these trigger a Filter Whatsers use case.

On the View Details Page, users may scroll through a collection of Whatsers, displaying the details of each instance one by one. As per the List Page, this can be implemented using buttons or instance number sequences. This command triggers **Next Whatever**.

From the View Details Page, additional command buttons lead to the **Delete Whatever**, **Update Whatever** and **Copy Whatever** use cases. As per the List Page, selected

attributes of a Whatever may be hyperlinked to trigger the Filter Whatever use case. This is of course restricted to non-unique attributes.

On the Edit Page, users can create or update instances of Whatever. Ultimately, either the **Save** or the **Cancel** button is used to complete the activity.

On the Search Page, a **Find** button is used to trigger the Find Whatever use case. On the Filter Page, selection of an attribute value triggers the **Filter Whatever** use case. Depending on the display style, attribute values may be selected from a popup menu or from a list that is displayed as a table.

The **Service a Whatever** use case finds its way into the command language by way of more command buttons: Print this Whatever and Download Whatever Details are the most common examples of this command. Service Whatever can also be implemented on the List Page; this requires a mechanism to select multiple rows in the list and subsequently apply a command to all selected rows. This is known as **Do With Selected Whatever**.

To summarize, the command language for data management includes the following fifteen user requests, which are discussed in much more details in the remainder of this white paper.

Request	Description
new whatever	display an empty edit page in create mode.
copy whatever	display a pre-populated edit page in create mode. Requires identifier of the whatever to copy.
update whatever	display a pre-populated edit page in update mode. Requires identifier of the whatever.
save whatever	process form data as either update or insert and revert to view whatever page.
cancel	cancel edit task and revert to view whatever or list whatever page.
delete whatever	delete current whatever from persistent storage and revert to list whatever page. Requires identifier of a whatever.
view whatever	view details of selected whatever. Requires identifier of the whatever.
list whatever	display list page with a overview of all whatever. In practice, often interpreted as 'list my whatever' (i.e. for any user a predefined notion of ownership applies).
sort whatever	re-display the list page, applying the indicated sort order and column. Begin on page 1 in case of a paginated list.
page whatever	re-display the list page, showing the selected page (subset) of whatever. Maintain existing sort order.
search whatever	display the search page.
find whatever	process form data as a query by example and revert to either list or view details page with search results, or to search page in case no matching whatever were found.

filter whatever	display a filter page.
match attribute	process selected attribute value as a query and revert to either list or view details page with query results (the case of no matching whatever's should be an exception).
print whatever details, print whatever list, export whatever's	these are examples of services that apply to whatever. These may or may not be part of the application. All these services require an identifier of the whatever(s).

Table 2 Command Language Summary

3.4 Repository Access

The requests processors described in this white paper will in most cases need to access the underlying data storage. A repository access pattern is specified to align with the user requests in an optimal way. This pattern is the subject of a subsequent white paper. Here, a summary of the repository access methods is provided.

Method	Description
insert	accept an instance of a Whatever and persist it. Return whatever instance with added synthetic data.
update	accept an instance of Whatever and persist any changes. Return whatever instance.
delete	accept a Whatever identifier and remove it from persistent storage.
get	accept a Whatever identifier and return instance details.
match	apply criteria to identify a new collection of whatever's. This is the 'current collection' going forward. Return a count of matching records.
page	apply current criteria, including a sort attribute and sort order, and a pagination setup to identify a subset of whatever's to return. Ensure a <i>minimally</i> sufficient subset of attributes is provided.
scroll	apply current criteria, including a sort attribute and sort order, and an offset passed in via a pagination setup and return instance details. Ignore page size and always return <i>full</i> details of a single instance.
domain	accept an attribute identifier and return an ordered collection of all <i>possible</i> key-value pairs of the attribute (only applies to enumerated values).
range	accept an attribute identifier and return an ordered collection of <i>used</i> key-value pairs of the attribute.
model	return a formal description of the Whatever domain model (this is an optional method)

Table 3 Repository Access Methods

4 Common Processing Aspects

Before turning to the detailed analysis of request processors, the 'common ground' that all such processors will share is identified.

Firstly, the concept of a processing context which represents the current state of a user's interaction with a repository is determined. State is needed mainly because of the need to cancel inserts and update operations. Also, whilst scrolling through a collection of *Whatevers* in detailed mode, the context must maintain a notion of the current criteria that define the collection.

Other common elements of request processing include ways to exchange information with the repository, validation and error handling, access control as well as empty collections and other exceptional situations.

4.1 Processing Context

There are many situations that call for the display of a known, stable state. In the middle of an update, a cancel would imply a return to the View page showing the original, unedited data. Many exception conditions likewise require the redisplay of an earlier stable context.

It appears that the most effective way to accomplish this is by holding a *processing context* as part of the user's session data. Such a context would either be the most recent List page or the most recent View page, as appropriate. Those two page types are definitely considered as stable contexts, although the actual data that was displayed may no longer be available.

Four specific items of information are required to completely determine the processing context; these include

- the criteria that define the current collection,
- the size of the current collection, the current offset in the collection and the current page size,
- the current sort attribute(s) and sort order(s),
- the display mode (list or detail page).

4.1.1 Maintaining a Context

As stated, a processing context must be maintained for the duration of a user session. In addition, an application may allow users to persist a context as a preference item, or simply persist the processing context when a session times out or the user logs out.

Session data may either be held at the client side or at the server side, the design of which is not part of the request processing pattern. Either way, all elements of the context must be Serializable; when held at the client side it will all go into an HTTP cookie.

4.2 Data Exchange with Repository

When talking to a repository, the request processors must pass along some standard pieces of information. The most important one no doubt is a Criteria object to specify an interest in a subset of Whatever's. `Repository.match()` accepts a Criteria object; `Repository.page()` and `Repository.scroll()` both return results that include a Criteria object as well.

Other relevant information is passed via a Pagination object or as a List of Sorting objects. Results from the repository are returned using an `OrderedResultPage`; this includes the List of Whatever's, the current Criteria, the current Pagination and the current List of Sorting specifications. Note that this constitutes an almost complete processing context as identified earlier.

Finally, lists of key/value pairs are returned by both `Repository.domain()` and `Repository.range()`. These are not to be implemented as (Java) Map types because the keys will not be known by the request processors. Instead, these methods will return a List of `KeyValuePair` objects, possibly extended with additional attributes (such as a description).

4.3 Data Validation & Error Handling

All request processors are responsible for validating the request arguments that are passed to them.

Request processors that detect validation errors have a responsibility to inform the user of the issue at hand. Wherever possible, the user must be given an opportunity to correct any error that is present. The way this should be done is by forwarding to the same page that triggered the validation error.

4.4 Access Control

Many request processors will be faced with access control issues. Depending on the granularity of access control, users may be allowed access to a page and all of its contents, or some sensitive items on a page must be suppressed based on a user's role. Likewise, users may be entitled to modify Whatever's or only selected attributes of each Whatever, based on some application-specific criteria.

There are no general patterns for implementing an access control strategy that works in all situations. However, some effective strategies apply almost universally.

One is to verify that the user indeed has access to the feature implemented through a certain request (e.g. is this user entitled to delete Whatever's?). If not, the user must be diverted to a generic error page.

Another generic strategy is to prevent a user from ever navigating to a page she or he is not entitled to work with. As an example, when a user has read-only access to a certain instance of a Whatever, the View Details page should display the "Update Whatever" command in a disabled state or perhaps completely hide it. As the user has no entitlements to modify the Whatever, it is of no use to let her/him navigate to the Edit page in the first place. Likewise, the "New Whatever" command should probably also be disabled.

4.5 Dealing with Empty Result Sets

List, Filter and View pages must be designed to deal with empty result sets. In View mode, a user can click on Delete repeatedly until all instances are deleted. Likewise, the List page (which is very often used as a 'home page' in data management) may be empty at initial deployment or following removal of all entities. As the Filter page is populated from currently used attribute values, it will be empty when there are no entity instances.

4.6 Dealing with Exceptions

Data validation was discussed earlier, in section 4.3. That discussion also talked about the communication of validation errors to the user. The same style of communication applies to errors found during business logic validation. It may be that the business logic states that entity X cannot be deleted as long as it is the parent in a parent-child relationship. This type of business logic is not regarded as data validation but instead as business logic validation. Errors of this nature should be communicated to the users in the same way as data validation errors.

A completely different class of problems is what is referred to as exceptions. These are conditions that the user has no control over. She/he never was the cause of the problem, and the user cannot be expected to correct this type of problem in any way. At best, the system can inform the user of a critical situation, with a suggestion on what operational procedure to follow in order to get it resolved.

5 Request Processors for Data Management

In this chapter, a high-level design for each of the fifteen request processors of Manage a Whatever is presented. This design is completely independent of the business domain which is a *good* thing.

5.1 New Whatever

new – display an empty edit page in create mode.

Description

The new request is the entry point for the Create a Whatever use case. The goal is to display the Edit Page.

Responsibilities

To populate the enumerated attributes of the edit page with selection elements. For this, the request processor must query the repository about the valid domains of all enumerated attributes using `Repository.domain()`.

This request should leave any existing context undisturbed. Only when the newly created Whatever is saved will the context change.

Inputs

None.

Forward

The request processor must forward to the Edit Page flagging this operation as an *Edit for Insert*, as opposed to an Edit for Update. As the Edit page may lead to a cancel request, this request processor must ensure a valid context is maintained.

When fine-grained access control is applied, the Edit page must be processed so that any non-editable elements are disabled before the page is shown to the user.

5.2 Copy Whatever

copy – display a pre-populated edit page in create mode.

Description

The copy request is the entry point for the Copy a Whatever use case. The goal is to display the Edit Page with as much pre-populated data elements as possible, using a source instance to copy values from.

Responsibilities

To populate the enumerated attributes of the edit page with selection elements. For this, the request processor must query the repository about the valid domains of all enumerated attributes.

This request processor must retrieve the details of the source instance from the repository (using `Repository.get()`) and apply its understanding of the business do-

main before the source data is used to pre-populate the Edit page. Some attributes of the source instance, such as the instance identifier as well as any other identifying properties, are inherently private to the source instance and should not be copied.

This request should leave any existing context undisturbed. Only when the newly created Whatever is saved will the context change.

Inputs

Requires identifier of the whatever to copy (the source instance).

Forward

The request processor must forward to the Edit Page flagging this operation as an *Edit for Insert*, as opposed to an Edit for Update. As the Edit page may lead to a cancel request, this request processor must ensure a valid context is maintained.

When fine-grained access control is applied, the Edit page must be processed so that any non-editable elements are disabled before the page is shown to the user.

5.3 Update Whatever

update – display a pre-populated edit page in update mode.

Description

The update request is the entry point for the Update a Whatever use case. The goal is to display the Edit Page pre-populated with the original data of the source instance.

Responsibilities

To populate the enumerated attributes of the edit page with selection elements. For this, the request processor must query the repository about the valid domains of all enumerated attributes.

This request processor must retrieve the details of the source instance from the repository (using `Repository.get()`).

This request should leave any existing context undisturbed.

Inputs

Requires identifier of the whatever to update (the source instance).

Forward

The request processor must forward to the Edit Page flagging this operation as an *Edit for Update*. As the Edit page may lead to a cancel request, this request processor must ensure a valid context is maintained.

When fine-grained access control is applied, the Edit page must be processed so that any non-editable elements are disabled before the page is shown to the user.

5.4 Save Whatever

save – process form data as either update or insert and revert to view whatever page.

Description

This request is issued when the user clicks on the “Save” button of a form. The goal is to persist the modified data in the repository.

Responsibilities

When fine-grained access control is applied, all form elements must be inspected to ensure that the user is entitled to make modifications to them.

All form data must be validated before it is persisted. This includes simple data validation as well as business logic validation.

When this is a ‘save for update’, the request processor must retrieve the instance details from the repository and modify the instance according to the form elements provided. Subsequently, `Repository.update()` is applied and the updated instance of Whatever is displayed on the View page within the processing context (i.e. the scrolling interface may be applied).

For a ‘save for insert’, the request processor must populate a new instance of Whatever with form data and apply `Repository.insert()`. After an insert, the processing context is reset and the newly created instance is shown on a View page without scrolling interface.

Inputs

This request requires form data and, in the case of a save for update, the instance identifier for the source instance.

Forward

This request processor must forward to the View Details Page. After a ‘save for update’, any existing context may still be considered valid, although the sort order may be disturbed. After a ‘save for insert’, there is no longer a processing context so the new instance is displayed in View mode without any scrolling interface.

5.5 Cancel

cancel – cancel edit task and revert to view whatever or list whatever page.

Description

The goal of a cancel request is to discontinue the current task and revert to the most recent ‘steady state’, which is encoded as the *processing context*.

Responsibilities

To use the processing context details to determine whether the most recent steady state was either a List or a View Details page, and what instance(s) was displayed on it. If the most recent steady state was a List page, then this request must for-

ward to that same List page, and when it was a View Details page, this request must forward to that View Details page.

Inputs

None.

Forward

This processor must forward either to the List or to the View Details page. When there is no processing context, some generic fallback ('list all') must be chosen.

When the page offset parameter is no longer valid (out of bounds), some sensible fallback value must be chosen by the processor. This may also be delegated to the repository, as the repository will signal such issues before the request processor will.

5.6 Delete Whatever

delete – delete current whatever from persistent storage and revert to most recent steady state.

Description

The goal is to permanently remove an instance of Whatever. After a successful delete by the repository, the request processor must execute a strategy to determine what should be displayed next.

In case a current collection is known, and the deleted instance was not the last instance of the collection, a satisfying strategy is to forward to the *next* instance, which now happens to exist at the same offset that was used by the deleted instance earlier. When the deleted instance was the last of the collection, that offset is no longer valid and the best instance to display is probably the *previous* instance.

When this request was issued from a List page (using 'With selected Whatever's do: Delete'), the request processor should forward to the List page, showing a page of instances starting at the same offset as before. When there are no more instances at that offset, the List page should be populated with whatever constitutes the last page of the collection. `Repository.match()` provides all the details of the current collection following the delete, so the processor can apply this information in its strategy.

Responsibilities

To permanently remove an instance of Whatever from the repository.

Inputs

Requires identifier of a whatever. May also be applied to a sequence of whatever's, which are processed one by one.

Forward

This request processor forwards to either the View page or the List page, depending on the processing context.

5.7 View Whatever

view – view details of selected whatever.

Description

The goal of this request is to present the full details of a single instance of a Whatever.

There are two entry points; one is when a drilldown from a list of Whatever is taking place, the other entry point is when a reference to a unique Whatever is made somewhere else in an application (typically in a totally unrelated module). The difference is that during drilldown, one would want to ensure that the processing context is maintained and made available. In the other situation, there is no processing context as the launching point is in an unrelated area of the application.

Responsibilities

For a view as drilldown, the processor must recall the context and apply `Repository.match()`. It must adjust the offset before applying `Repository.scroll()`. It must use the details of `OrderedResultSet` to set the new context. It must forward to the View Page with a scrolling interface.

For a context-free view, the processor must apply `Repository.get()`, clear the processing context and display a single instance in View mode without scrolling interface.

Inputs

Requires identifier of the whatever. A view as drilldown should pass an offset into the current collection; a context-free view should pass an instance identifier instead.

Forward

This request processor forwards to the View page, either with or without context.

5.8 List Whatever

list – display list page with a summary of whatever.

Description

The list request resets any current criteria in order to display the full list of Whatever. Any current sorting details as well as page size and attribute ordering details still apply.

The goal is to display a page of matching instances, beginning at offset 0.

In practice, applications often introduce 'List My Whatever' which suggests a pre-defined notion of ownership or relevance applies. This is implemented using the match request (see section 5.14 below), not using the list request.

Responsibilities

To reset any current criteria and page offset, and to recall any existing sorting details, page size and attribute ordering. When none such details are available, the request processor must make a sensible choice.

This processor must apply `Repository.match()` and `Repository.page()` to retrieve a page full of summary information.

The request process must install a new context so any subsequent operations have a known steady state to revert to.

Inputs

None.

Forward

This processor must forward to the List page.

5.9 Sort Whatever

sort – re-display the list page, applying the indicated sort attribute and order.

Description

The sort request applies the current criteria and resets any current offset in order to display the first page of Whatever in the new sort order. Any current page size and attribute ordering details still apply.

Responsibilities

To reset any current page offset, and to recall the current criteria as well as any existing sorting details, page size and attribute ordering. Depending on the business requirements, the new sort attribute and order either extend or replace any existing sort details.

When no criteria or page size details are available, the request processor must make a sensible choice.

This processor must apply `Repository.match()` and `Repository.page()` to retrieve a page full of summary information.

The request process must install a new context so any subsequent operations have a known steady state to revert to.

Note that when new sort details *extend* an existing sorting specification, an additional request option must be defined to flag that the new parameters replace an existing sort order (explicit override)

Inputs

An attribute to sort on as well as a sort order (ascending/descending). An optional 'override' flag when multi-attribute sorts are implemented and the user requests to suppress any earlier sort order.

Forward

This processor must forward to the List page.

5.10 Page Whatever

page – re-display the list page, showing the selected page (subset) of whatever.

Description

The page request applies the current criteria, page size, attribute ordering and sorting details. An offset into the current collection is applied to obtain a page of Whatever summaries.

When user-defined page sizes are applied, this request carries the new target page size as an additional parameter.

Responsibilities

To combine a new offset (and page size) with the criteria and sorting details from the processing context. To retrieve a page of summary details from the repository and display those on the List page.

Inputs

The offset into the current collection and optionally a page size.

Forward

This processor must forward to the List page.

5.11 Search Whatever

search – display the search page.

Description

The search request is the entry point for the Find a Whatever use case. The goal is to display the Search Page.

Responsibilities

To populate the enumerated attributes of the search page with selection elements. For this, the request processor must query the repository about the valid domains or ranges of all enumerated attributes using either `Repository.domain()` or `Repository.range()`. The choice between `domain()` and `range()` is largely a matter of business requirements versus performance optimizations. The `domain()` method is expected to be faster; the `range()` method is more effective as it ensures that there will indeed be matches against the selected value.

This request should leave any existing context undisturbed.

Refine Search

As an extension, refine may be used which will re-install the search parameters that are available in the current criteria. In this way, the user can refine her/his search by adding more criteria. The original criteria may either be displayed as pre-selected form fields or as non-editable values.

Inputs

None; for refine search a parameter that signals the re-use of earlier criteria.

Forward

This processor must forward to the Search page.

5.12 Find Whatever

find – process form data as a query by example and revert to either list or view details page with search results, or to search page in case no matching whatever were found.

Description

The find request is supposed to execute a search for matching instances; the goal is to present multiple matches on a List page; a single match can be displayed on the View Details page. When no matching instances are found, the Search page should be re-displayed.

Responsibilities

To create a new Criteria object that reflects the search values submitted via the search form. This processor must execute `Repository.match()` with the new criteria and forward to the appropriate results page depending on the match count.

This processor is responsible for defining a new context in case the match returns one or more matching instances.

Inputs

Form data with search values.

Forward

This processor forwards to the List page, the View Details page or the Search page, depending on the match count.

5.13 Filter Whatever

filter – display a filter page.

Description

The filter request is the entry point for the Filter Whatever use case. The goal is to display the Filter Page. In earlier releases of the Use Case pattern and the Page Flow pattern, this was known as Browse Whatever.

On a filter page, one or more enumerated attributes are presented as list of values *that are actually used in at least one Whatever*. There are various presentation alternatives for filters, including popup menus and tabular displays. A filter page may contain more than one such attribute value list to choose from.

Responsibilities

To apply `Repository.range()` for as many attributes as are needed on the Filter page.

Inputs

When each filter page provides all values of just one single attribute, this request comes with an attribute name so the processor will know which range to ask for. When there is only one filter page, no further inputs are needed.

Forward

This request forwards to the Filter page.

5.14 Match Attribute

match – process selected attribute value as a query and revert to either list or view details page with query results.

Description

For the request processor, the match request is almost identical to a find request. The only difference is that the search form is replaced by a single attribute name/value pair. Because the attribute values were provided earlier through a `Repository.range()` call, the case of no matching whatever should be an exception.

Responsibilities

To create a new Criteria object that reflects the attribute value filter. This processor must execute `Repository.match()` with the new criteria and forward to the appropriate results page depending on the match count.

This processor is responsible for defining a new context in case the match returns one or more matching instances.

Inputs

An attribute name and value to match.

Forward

This processor forwards to the List page or the View Details.

5.15 Service a Whatever & Service Whatever

service – these are services that apply to whatever.

Description

This is the extension point of the Manage a Whatever pattern. Depending on the application requirements, a single request processor for all extensions may be used, or alternatively an extra processor may be provided for each additional service.

In general, assuming a service does not add or remove any instances of whatever, the generic approach upon completion of the service execution should be to display a list or detailed view as per the processing context.

Responsibilities

To execute the requested service on the instances of Whatever that are passed in.

Upon completion, the processing context is used to redisplay either a List page or a View Details page.

Inputs

This request requires a (list of) instance identifier(s) and when a single request processor is used to dispatch multiple services, the service name must also be provided.

Forward

This request should forward to either the List page or the View Details page depending on the processing context.

6 Extensions

6.1 Multi-Page Details & Edit

In many cases, the complexity of a Whatever is such that it is no longer feasible to provide all details of a Whatever on a single page. Often, a tabbed GUI style is used to organize independent groups of attributes on sub-pages. Typically, one group of attributes is visible in the top half of the page, and the bottom half of the pages is used to display a tabbed section. The top half is used to ensure that end users never lose the focus on the specific instance they are dealing with. Rather than repeating the attributes on each tab, they are simply displayed outside the tabbed GUI.

As the details of a Whatever are now spread across multiple distinct pages, there will also be distinct Edit pages; one each for every tab that contains editable data.

For the request processing pattern the implication is that there will be as many view requests as there are tab pages, and also multiple update and save requests. A complication may be when a valid entity update transaction in the back-end requires data on multiple tabs to be present. In this situation, the request processors must co-operate to ensure that a transaction is only started when a minimally sufficient subset of attributes is accumulated across the tabs. This means that there will be much more state to carry along from page to page; which in turn may lead to an alternative design that keeps the various tab displays as Dynamic HTML on the client side and limits the client/server request to a single save across all tabs.

6.2 Interaction with Tree Viewers

Hierarchical tree viewers have become a popular navigation paradigm for complex information structures. In simple hierarchies, all nodes are of the same type; in more complex hierarchies, there can be different node types (compare files and folders in a hierarchical folder viewer). Even more complex hierarchies include the concept of 'multiple parent nodes' which means a node may have more than just a single parent.

In general, the display of and interaction with a hierarchical tree viewer should not be considered part of the Manage a Whatever pattern, as it is not limited to a single business domain (different node types). The volume and detail of information in a tree view is dynamic and depends on the number of opened nodes in the tree. This style of interaction requires more specific attention from the information architects and the software designers.

Ultimately, though, users navigate a hierarchical tree browser with the purpose of identifying a single node in the tree and starting a task from there. So, all leaf nodes and most likely also all other nodes of the tree are starting points for Manage a Whatever patterns. Due to the nature of a tree browser, the pattern that starts at a node no longer includes "List Whatever's", but instead it is constrained to tasks that refer to a single instance of a Whatever, such as View, Update, De-

lete, Copy, Print and Download. Popup menus with applicable tasks are generally used to start a standard MaW task.

A complicating side-effect of not having the List Whatever use case implemented is that it is difficult to combine hierarchical tree viewers with "Find a Whatever", as the Find use case typically displays its matches in a list view. Search results now have to be inserted into the tree as a new node.

6.3 A Service Layer

In the discussion so far, it has been assumed that request processor access the Repository directly. In reality, the *Service Layer* pattern (ref. [10], pp 133-142) is often used to decouple the two.

The responsibility of the service layer is to process a command and apply all business logic (validation, exception handling, auditing, logging, etc.). The request processor becomes a thin protocol transformation layer. The benefit of this approach is that there is no duplication of business logic across similar request processors, and a potential reuse of the service layer, e.g. for use as a Web Services provider interface.

6.4 Customize Whatever Summary

This is a refinement of the display of summary information. The idea is to allow the user to specify the data columns of interest and their relative ordering. This customization subsequently becomes part of the processing context. As a new request, the responsibility of the request processor is to make the new attribute ordering part of the processing context and to forward to the List page with the most recently used context (essentially redisplaying the previous List page with reordered columns).

7 Conclusions

A standard set of fifteen user requests suffices to describe the full interaction of an end user with the server-side implementation of Manage a Whatever. In turn, the fifteen request processors rely on only nine methods of the Repository Pattern for the interaction with an underlying data storage tier. Of these nine methods, one each is needed for insert, update and delete; the other six are all related to information retrieval.

The most common access pattern to the repository is a two-step approach; first `Repository.match()` is applied to obtain a count of matching instance. This is followed by a subsequent `Repository.page()` or `Repository.scroll()` to obtain actual instance data.

A certain amount of context is required to maintain a flow of interaction between a user and the application; this context must be maintained as user session state, either at the server side or at the client side.

Appendix A - References

- [1] "Data Management Requirements – A Use Case Pattern Language", Version 1.1, White Paper by Bert Hooyman, MphasiS UK Ltd., March 2005
- [2] "Data Management Information Architecture – A Page Flow Pattern Language", Version 1.1, White Paper by Bert Hooyman, MphasiS UK Ltd., March 2005
- [3] "Repositories for Data Management – A Data Access Pattern Language", Version 1.0, White Paper by Bert Hooyman, MphasiS UK Ltd., April 2005
- [4] "A visual vocabulary for describing information architecture and interaction design", Jesse James Garrett, version 1.1b, March 6, 2002, on the web at <http://www.jjg.net/ia/visvocab/>
- [5] "Writing Effective Use Cases", Alistair Cockburn, Addison-Wesley, 2001. ISBN 0-201-70225-8
- [6] "Designing Easy-to-use Websites", Vanessa Donnelly, Addison-Wesley, 2001. ISBN 0-201-67468-8
- [7] "Designing Data-Intensive Web Applications", Stefano Ceri, Piero Fraternali, Aldo Bongio, Marco Brambilla, Sara Comai, Maristella Matera, Morgan Kaufmann Publishers, 2003. ISBN 1-558-60843-5
- [8] "Web Database Applications with PHP & MySQL – Building Effective Database-Driven Web Sites", Hugh E. Williams and Davis Lane, O'Reilly 2002. ISBN 0-596-00041-3
- [9] "Web Application Design Handbook : Best Practices for Web-Based Software", Susan L. Fowler and Victor R. Stanwick, Morgan Kaufmann Publishers, 2004. ISBN 1-558-60752-8
- [10] "Patterns of Enterprise Application Architecture", Martin Fowler, Addison-Wesley, 2003. ISBN 0-321-12742-0
- [11] "Domain-Driven Design", Eric Evans, Addison-Wesley 2004. ISBN 0-321-12521-5