Error handling in the PDR

Joop Ringelberg 08-01-21 Version: 1

Introduction

Purescript offers the possibility to **catch** runtime errors. Having caught an error, one must handle it. The aim of handling is to protect the end user from situations that he cannot handle himself. Ideally, this means that the end user can continue to use the program as if nothing had happened. This may for example be the case when the source of the error was unavailability of the network connection. The end user may than be advised to restore the connection and/or try later. In general we try to inform the user that the action he initiated could not be completed. We also try to inform him on how to behave to remedy the situation that has arisen.

In the future we may add functionality to send these errors to developers, if the end user chooses so.

We have a second objective when handling errors and that is to provide good debugging information. This means not too low in the function call stack (it is of little use to know that the error occurred on accessing the database), neither too high ('something went wrong when you tried to open this screen' is not very useful, either).

Sources of errors

An obvious source of errors is the programmer. The purescript compiler catches most of these errors before the program can run, thus preventing them from becoming runtime errors. Nevertheless, some programming mistakes may persist and cause runtime errors. By testing we try to eliminate them all. We do not try explicitly to handle such errors.

The program relies on secondary services for its normal operation, such as a database and an internet connection. These are a major source of runtime errors: the services may be unavailable, or may malfunction. In the case of the database, some artefacts may be missing, for example. We try to catch and handle all these potential problems.

Fetching type instances from the database

In particular, the PDR is written on the assumption that a resource identifier (such as e.g. a role or context) can always be exchanged for the resource itself at the database. This assumption may be violated by malfunctioning of the database or because someone has removed a resource by other means than the PDR. Consequently, we have protected each and every access to the database by error catching- and handling code.

Fetching models from the database

A special case of resources taken from the database are the documents that represent type information. A model file (written in ARC) is translated into a DomainFile (a JSON document). These resources are taken from the database just like context- and role instances. We do protect functions that access these resources by error handling code.

What if we cannot exchange the identifier of a model for a DomeinFile, in the database? When this happens on handling incoming deltas, or JSON structures that represent resources coming in as dropped files or as fragments sent in through the GUI, we try to retrieve the model from a repository.

All other cases constitute a serious error condition. It is an error of a type that we can only protect the PDR against on a high level: that of processes that rely on model reflection but can be skipped entirely without compromising the state of the PDR.

Probably this will go all the way up to the end user, or his bot. We may not be able to answer a request by the end user, or to perform a side effect ordered by him. Practically, this means catching the error in the API itself. Similarly, we may be unable to compile a bot rule. Obviously we should inform the end user about such a situation.

Providing good debugging information is difficult for these cases. The moment we signal the error is not the moment it was caused. Neither does it matter much to know why we were looking up a type definition; it is the fact that the model is missing, that matters.

Fetching a model when handling an incoming transaction

It turns out we only need to check for a missing model when a UniverseRoleDelta arrives. A UniverseContextDelta is always preceded by a delta for its external role and the type of the external role and the type of its context are invariably in the same model. A property delta refers to a role that must be present, hence a role delta always precedes a property delta that would require a new model. Even if the property was defined in an aspect, we would have imported the model that defines that aspect as a dependency of the model that defines the role.

Looking up type definitions in a model file

Should we check, in the source code, if we can find a type name in its model? Or do we assumethat type names can always be found?

To answer this question, we must first establish the source of the type names that are run through these lookup functions. There are four such sources:

1. They may be hard coded, i.e. be part of the program source code.
2. They may occur in model source texts.
3. They may occur in DomeinFiles.
4. They may be a type reference in a context- or role instance.

Hard coded type names are the sole responsibility of the programmer. We should not include error checking in code to catch such programming mistakes!

Handling unknown type names in model source texts is the job of the parsing system.

Type names should only be included in DomeinFiles if they are defined in the model source files. Again, this is taken care of during parsing a source file and translating it into a DomeinFile.

That leaves us with type references in instance data. We have addressed this problem in the text model versions and compatibility. To summarise the discussion there:

- by using persistent internal names in DomeinFiles and as type references in instance data, we protect ourselves against errors that would otherwise arise when type names change in successive model versions. In other words, even if the readable name of a type has changed in a model, we can still look up the type definition in that new model version by using the internal name of the type.
- by including version identifiers in type names, we can recognise incompatibility of incoming data (deltas) with the version of the model that is being used locally. Only type references that can be looked up in the locally available DomeinFiles will pass the screening at the gate performed when handling deltas.

So we can finally answer the question: Should we check, in the source code, if we can find a type name in its model? with a resounding: NO!

Authentication errors

The PDR relies on two services that require authentication: the (local) database and the message broker.

Couchdb has a notion of session with a time out period. We want the PDR to authenticate automatically if necessary, after the initial login by the end user (once the end user has authenticated himself with the PDR). This requires we catch messages returned by the database server to the effect that the session has expired.

The same holds for RabbitMQ, the message broker. However, the library we use to connect to RabbitMQ handles this by itself.