Database - Bug #7504

Export data tool from FWD into .d files

07/11/2023 08:55 AM - Alexandru Lungu

Status:	WIP	**Start date:**	
Priority:	Normal	**Due date:**	
Assignee:	Alexandru Lungu	**% Done:**	0%
Category:		**Estimated time:**	0.00 hour
Target version:			
billable:	No	**case_num:**	
vendor_id:	GCD	**version:**	

Description

History

#1 - 07/11/2023 08:59 AM - Alexandru Lungu

This is a task inspired by #7241: the need of exporting the data from FWD in .d files to do a schema upgrade on import.

The goal here is to have a Java native fast internal tool that allows the extraction of persistent data into .d files. There files should be "importable" in the sense that they can be used for a further database import back in a newer FWD. This is fundamental to schema upgrades and data migration.

#3 - 07/24/2023 03:24 AM - Alexandru Lungu

- Status changed from New to WIP

- Assignee set to Ștefan Roman

#4 - 07/28/2023 07:51 AM - Alexandru Lungu

Plan

We need an ant option for this. To be consistent, I suggest something like ant export.db that relies on the build.properties configuration (database HOST, PORT, USERNAME, PASSWORD, ENGINE, etc.). The target should be a light-weight brand new com.goldencode.p2j.schema.ExportWorker main class.

The database can be spoiled with different kind of extra tables / fields we don't actually want to export, right? Also, I don't think that we actually want to rely on an existing .df for that matter, so we need to stick to a complete solution that works exactly like the OpenEdge export tool.

- We need to infer what tables are to be exported and what are their legacy name / order. If we simply select all tables from the database, we may end up with incorrect names, order or tables that we don't actually support in FWD (we don't have specialized DMO classes). The same goes for the table fields and indexes. **The most reliable meta-data source is our own DMO annotations**. Thus, I suggest making use of:
  - DmoMetadataManager to actually parse our own DMO classes.
  - DatabaseManager.getDatabaseTables to trigger the registering of all DMO classes we have.
- Once we have all DmoMeta for each table to export, we need to do the actual data dumping. This is not quite a challenge; we can easily use Persistence.executeSQLQuery to retrieve a list of the results we intend to use. Don't forget here to make use of sever-side cursors: no auto-commit, set fetch-size and FORWARD-ONLY.
- Once we have the hydrated records, we can basically use Stream to do the dumping. However, how do we actually take the data out of the DMO?
  - Maybe we can use something similar to TableMapper.getLegacyOrderedList and DmoMeta.getLegacyGetterMap to actually retrieve the underlying BDT for each legacy field **in the right order**!
  - For each list of BDT (a table row basically), we can simply call Stream.putWorker with ExportField.
- AFAIK, lobs are dumped in a separate LOBS folder, so we will need to support this as well.
- .d files also have a footer, so we may want to generate it as well.

Challenges

- We need to double check the export for each kind of type we have. Simple: int, char, logical, etc., but most important we need to pay extra attention to edge cases like memptr, blob, clob, etc.
- We need to ensure the output is consistent with the .df that we used for the initial import, but most important the output should be "equal" to the initial dump we use for testing. This implies order of fields.
- Is the cpstream relevant in this whole process? If no, I guess we can set a standard cp-stream (UTF-8) and just specify in the footer whatever we

used. Otherwise, we can make it configurable from Java args.

Dump definition

I think we have all we need to actually regenerate the df file, but I guess this is not a priority here. Usually, this tool will be used to reimport the database, so the df would still be there from the very initial import. However, if the df is to be lost in the process, we can actually integrate a tool that also export the data definition based on the existing code (dmo package).

#5 - 07/28/2023 02:52 PM - Greg Shah

I agree our ant build should have an option to use this, but I do want our utility code (Java) to be a complete standalone command line tool with all needed parameters passed on the command line. We are moving away from the build.properties approach in the near future, so only the ant scripting should depend on it.

> The database can be spoiled with different kind of extra tables / fields we don't actually want to export, right?

I'm not sure about this. In the 4GL, the code would usually be written with an outer loop of for each _file: which would pick up all tables. If we don't do that, then we can't really migrate the existing database.

> .d files also have a footer, so we may want to generate it as well.

Yes

> We need to double check the export for each kind of type we have. Simple: int, char, logical, etc., but most important we need to pay extra attention to edge cases like memptr, blob, clob, etc.

We should match the behavior of the EXPORT statement.

> Is the cpstream relevant in this whole process?

Yes, our import process is very dependent upon the encoding.

#6 - 07/28/2023 03:22 PM - Ovidiu Maxiniuc

Greg Shah wrote:

> The database can be spoiled with different kind of extra tables / fields we don't actually want to export, right?

> I'm not sure about this. In the 4GL, the code would usually be written with an outer loop of for each _file: which would pick up all tables. If we don't do that, then we can't really migrate the existing database.

If this utility/script will be run as the import counterpart then the full persistence support might not be available (the _meta database and _file populated). But, as for import, we can use the .p2o xml, which is normally present.

Alternatively, the dmo/ of a specified schema folder can be scanned for @Table annotated interfaces and iterate from there. Registering the DMO with DmoMetadataManager, which is anyway required, will return the DmoInfo with all needed metadata. I would recommend staying away from TableMapper with this utility.

#7 - 08/08/2023 08:39 AM - Greg Shah

From Stefan via email:

> I created an ExportWorker class for #7504 and now I am trying to understand ant import.db and ImportWorker approach, so I can try a similar one for the export tool.

I actually would like to move away from the current ant import.db and ImportWorker approach. These rely upon TRPL and there is not a good reason to use TRPL for this purpose. The ASTs we are "traversing" are highly regular and using an AST for this "structure" is overkill. It complicates the solution without adding any value. I prefer if we read the stored metadata dynamically and then just processed the tables as needed.

For the export, please avoid TRPL completely.

#8 - 08/17/2023 06:49 AM - Alexandru Lungu

> I actually would like to move away from the current ant import.db and ImportWorker approach. These rely upon TRPL and there is not a good reason to use TRPL for this purpose. The ASTs we are "traversing" are highly regular and using an AST for this "structure" is overkill. It

> complicates the solution without adding any value. I prefer if we read the stored metadata dynamically and then just processed the tables as needed.

Greg, by stored metadata dynamically you mean the beans we can find in the dmo package (read annotations, hydrate from database and export) or the ones we find in .p2o xml (query the database based on that information). Stefan already has some attempts with both solutions (the second one being more advanced).

- If we rely on the dmo package, we are bound to the generated jar and its annotations. I guess this is OK and clean as we have a lot of logic with DMO already in our persistence layer (i.e. hydration). The good part here is that we have the hydration for free (including extent fields resolution). Also, all of our persistence layer is based on Record. The bad part here is that we depend on the customer application compiler solution (I don't think we ever do that in our ant commands).
- If we rely on the p2o solution, we can execute such routine without having the application compiled. The good part is that this was already achieved until some point and it was quite accessible to retrieve and parse the p2o. The bad part is that we don't have a clean solution of "hydrating" (going from Java type -> FWD type -> ExportField). AFAIK, there is no FWD type factory solely based on Java types (without record/DMO meta). At best, I think we can retrieve the field type from the p2o, but this will require a

Please advice on the approach here.

> If this utility/script will be run as the import counterpart then the full persistence support might not be available (the _meta database and _file populated).

I don't think that a script is the way to go here. A Java utility will be the solution.
Is there any gain into having such information (_meta / _file)? Of course we won't have the full support, but using FWD persistence layer is still the way to go. There are easier solutions as simple JDBC metadata inquiry, but these are too "vanilla" and may loose some FWD inner functionality.

> I agree our ant build should have an option to use this, but I do want our utility code (Java) to be a complete standalone command line tool with all needed parameters passed on the command line. We are moving away from the build.properties approach in the near future, so only the ant scripting should depend on it.

An ant command is exactly what I was thinking of as an interface. Note that we have already documented build_db.xml as "commands for import, export, drop, create", even though there is no export yet. Of course, most of the commands will be the same as the import's one (database, url, user, password, etc.), so I can't see how this **can** be unrelated to an eventual build.properties. I mean, we can remove build.properties and hard-code the parameters into the command-line, but I don't see the advantage here.

#9 - 08/17/2023 07:11 AM - Greg Shah

> Greg, by stored metadata dynamically you mean the beans we can find in the dmo package (read annotations, hydrate from database and export)

Yes, this one.

> or the ones we find in .p2o xml (query the database based on that information).

I want to avoid this usage as this brings more dependencies on TRPL.

- If we rely on the dmo package, we are bound to the generated jar and its annotations.

We already have this dependency for ImportWorker since we use the DMOs to actually process each record.

> An ant command is exactly what I was thinking of as an interface. Note that we have already documented build_db.xml as "commands for import, export, drop, create", even though there is no export yet. Of course, most of the commands will be the same as the import's one (database, url, user, password, etc.), so I can't see how this **can** be unrelated to an eventual build.properties. I mean, we can remove build.properties and hard-code the parameters into the command-line, but I don't see the advantage here.

The current ant approach has many limitations:

- It requires ant to be installed in addition to FWD.
- It is pretty old and doesn't handle things like dependency management, so most customers would prefer moving to gradle or maven. We ourselves prefer gradle for project builds but we have development involved to make the shift. If I recall correctly, gradle still may have some problems with aspectj.
- It is something that is outside of FWD and so we have to maintain independent copies of this scripting for every customer project. Anything built into FWD is **standard** and does not have to be maintained separately.

Yes, any such utilities will often have some requirement for configuration. But we don't want them to be hard coded to a specific project build approach that we have been using (build.properties, build.xml and the build_db.xml). That stuff was created as a quick and dirty way to script the import and was never meant to be the utimate solution.

#10 - 08/17/2023 07:19 AM - Alexandru Lungu

> Yes, any such utilities will often have some requirement for configuration. But we don't want them to be hard coded to a specific project build approach that we have been using (build.properties, build.xml and the build_db.xml). That stuff was created as a quick and dirty way to script the import and was never meant to be the utimate solution.

Well, the ant script is ultimately calling a Java command under the hood right, but with some "easy to set" parameters. How do you think of having this integrated? From the Java side, all these options will come a args right? So it makes sense to have an ant (or anything else, including command-line) in place to avoid long command lines. Or maybe we should have a export.sh?

Anyway Stefan, please fall-back to the dmo package approach. This means having <customer-application>.jar in path to do the proper Class look-up.

#11 - 08/17/2023 07:33 AM - Greg Shah

We might want to consider reading the configuration from the directory. In that case it will require little or no parameters.

#12 - 08/17/2023 07:41 AM - Greg Shah

Also I should point out that export will be used on production systems that won't have the standard application cfg project installed.

#13 - 08/17/2023 08:21 AM - Alexandru Lungu

Greg Shah wrote:

> We might want to consider reading the configuration from the directory. In that case it will require little or no parameters.

An abstract approach is suitable then. Stefan, please pass down the parameters through the args. Later on, we can adapt this routine to use DirectoryService identically. The point here is that we shall use an abstraction on the way we retrieve the parameters. Consider using an ExportConfiguration instance that is built from main (for now) and passed down to the ExportWorker. Do your development independently from ant / customer project structure. Just add the customer-application jar into the classpath and move on.

I think everything is clear now.

#14 - 08/17/2023 09:42 AM - Eric Faulhaber

Another configuration option to consider is to allow a dump of only a subset of tables (perhaps an inclusive or exclusive filter), instead of every table for which a DMO exists.

This poses special challenges, in that:

- it can produce an extremely long command line parameter, compared to other, simpler arguments;
- it should support ad-hoc use, making the directory an inconvenient mechanism.

This is only an issue for the export. The import is naturally filtered by which *.d files are present in the dump directory.

This is a secondary concern to getting the base mechanics of an export working, but it is something to consider now, so we don't make it harder to implement later.

#15 - 08/22/2023 04:57 AM - Alexandru Lungu

- Assignee changed from Ștefan Roman to Alexandru Lungu