


# 10 Things You're Doing Wrong in Talend

- 
- [Sam Hopper](#)
- December 05, 2019
- Datalytx.

## ...and how to fix them!

We've asked our team of Talend experts to compile this top ten list of their biggest bugbears when it comes to jobs they see in the wild – and here it is!

### 10. Size does matter

Kicking off our list is a common problem – the size of individual Talend jobs. Whilst it is often convenient to contain all similar logic and data in a single job, you can soon run into problems when building or deploying a huge job, not to mention trying to debug a niggling nasty in the midst of all those tMaps!

Also, big jobs often contain big sub-jobs (those blue-shaded blocks that flows form into), and you will eventually come up against a Java error telling you some code is “exceeding the 65535 bytes limit”. This is a fundamental limitation of Java that limits the size of a method's code, and Talend generates a method for each sub-job.

Our best advice is to break down big jobs into smaller, more manageable (and testable) units, which can then be combined together using your favourite orchestration technique.



## 9. Joblets that take on too much responsibility

Joblets are great! They provide a simple way to encapsulate standard routines and reuse code across jobs, whilst maintaining some transparency into their inner workings. However, problems can arise when joblets are given tasks that operate outside the scope of the code itself, such as reading and writing files, databases, etc.

This usage will require additional complexities when reusing joblets across different jobs and contexts, and can lead to unexpected side effects and uncertain testability.

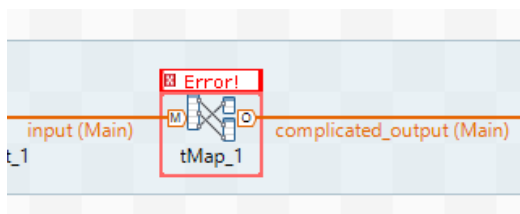
We have found that you can get the best from joblets when treating them as the equivalent of [pure functions](#), limiting their scope to only the inputs and outputs defined by the joblet's connections and variables.

## 8. “I’m not a coder, so I’m not looking at the code”

Ever seen a `NullPointerException`? If you’re a Talend engineer, then of course you have! However, this kind of run-time error can be tricky to pin down, especially if it lives in the heart of a busy `tMap`. Inexperienced Talend engineers will often spend hours picking apart a job to find the source of this kind of bug, when a faster method is available.

If you’ve worked with Talend for a while, or if you have Java experience, you will recognise the stack trace, the nested list of exceptions that are bubbled up through the job as it falls over. You can pick out the line number of the first error in the list (sometimes it’s not the first, but it’ll be near the top), switch to the job’s code view and go to that line (`ctrl-L`).

Even if the resulting splodge of Java code doesn’t mean much, Eclipse (the technology that Talend Studio is built on) will helpfully point out where the problem is, and in the case of a `tMap` it’s then clear which mapping or variable is at fault.



```
complicated_output_tmp.field3197 = "blah";  
complicated_output_tmp.field3198 = "blah";  
complicated_output_tmp.field3199 = "blah";  
complicated_output_tmp.field3200 = "blah";  
complicated_output_tmp.field3201 = "blah";  
complicated_output_tmp.field3202 = "blah";  
complicated_output_tmp.field3203 = "blah";
```

## 7. No version of this makes sense

Talend, Git (and SVN) and Nexus all provide great methods to control, increment, freeze and roll back versions of code – so why don’t people use them! Too often we encounter a Talend project that uses just a single, master branch in source control, has all the jobs and metadata still on version 0.1 in Studio, and no clear policy on deployment to Nexus.

Without versioning, you’ll miss out on being able to clearly track changes across common artefacts, struggle to trace back through the history of part of a project, and maybe get into a pickle when rolling back deployed code.

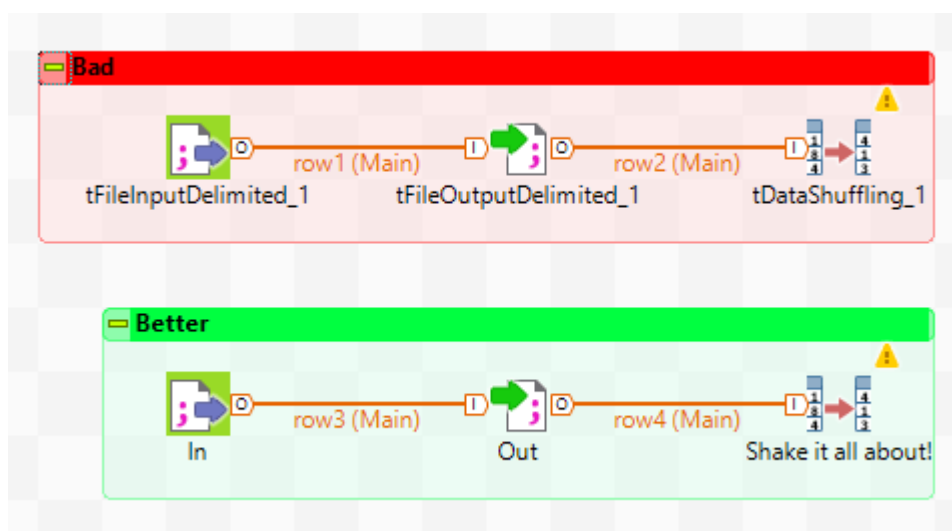
It’s too huge a topic to go into here, but our advice is to learn how your source control system works – Git is a fantastic bit of software once you know what it can do, and come up with a workable policy for versioning projects sources and artefacts.

## 6. What's in a name?

Whilst we're on the topic of not having policies for things, what about naming? A busy project gets in a mess really quickly if there's no coherent approach to the naming of jobs, context groups/variables, and other metadata items.

It can be daunting to approach the topic of naming conventions, as we've all seen policy documents that would put Tolstoy to shame, but it doesn't need to be exhaustive. Just putting together a one-pager to cover the basics, and making sure all engineers can find, read and understand it, will go a long way.

Also, while you're about it, think about routinely renaming components and flows within jobs to give them more meaningful names. Few of us would disagree that `tFileInputDelimited405` is not quite as clear and informative as `LoadCustomerCSV`, so why don't we do it more? And renaming flows, particularly those leading into `tMaps`, will make everyday chores a breeze!



## 5. Don't touch the contexts!

So often we see context variables being updated at different points in a job – it's so easy to do as they're right there at `context.whatever`, with the right data type and everything. But then how can you refer back to the original parameter values? And what if you then want to pass the original context into a child job, or log it to a permanent record?

If you need a place to store and update variables within a job, the `globalMap` is your friend. Just be careful to cast the correct type when reading values back, or you may get Java errors. For example, if you put an integer into the map, make sure you read it back as `((Integer)globalMap.get("my_variable"))`, as auto-generated code will often put `(String)` as the cast by default.

## 4. But I like all these columns, don't make me choose!

Something that adds to the unnecessary complexity and memory requirements of a job is all those columns that are pulled from the database or read from the file and not needed. Add to that, all the rows of data which get read from sources, only to be filtered out or discarded

much later in the job, and no wonder the DevOps team keep complaining there's no memory left on the job server!

We find that a good practice is to only read the columns and rows from the source that the job will actually need. That does mean you may need to override the metadata schema and choose your own set of columns (although that can, and often should, then be tracked and controlled as a generic schema), and while you're about it, maybe throw a WHERE clause into that tDbInput's SQL statement. If that's sometimes impossible, or impractical to do, then at least drop in a tFilterColumns and/or tFilterRow at the earliest possible point in the flow.

### 3. Hey, lets keep all these variables, just in case

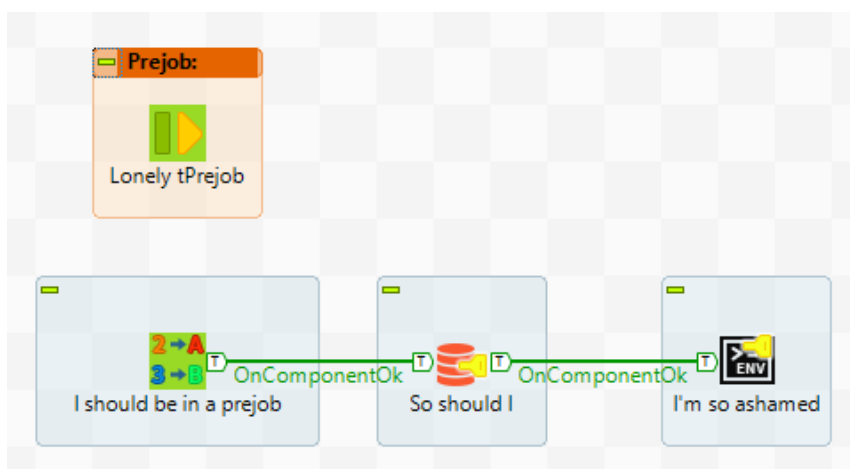
Context groups are great – just sprinkle them liberally on a job and you suddenly have access to all the variables you'll ever need. But it can soon get to the point where, especially for smaller jobs, you find that most of them are irrelevant. There can be also cases where the purpose of certain context groups overlap leading to uncertainty about which variable should actually be set to achieve a given result.

A good approach is first to keep context groups small and focussed – limit the scope to a specific purpose, such as managing a database connection. Combine this with a clear naming convention, and you'll reduce the risk of overlapping other sets of variables. Also, feel free to throw away variables you don't need when adding a group to a job – if you only need to know the S3 bucket name then do away with the other variables! They can always be added back in later if needed.

## 2. Jumping straight in

By missing out a tPrejob and tPostjob, you're missing out on some simple but effective flow control and error handling. These orchestration components give developers the opportunity to specify code to run at the start and end of the job.

It could be argued that you can achieve the same effect by controlling flow with onSubjobOk triggers, but code linked to tPrejob and tPostjob is executed separately from the main flow, and, importantly, the tPostJob code in particular will execute even if there's an exception in the code above.

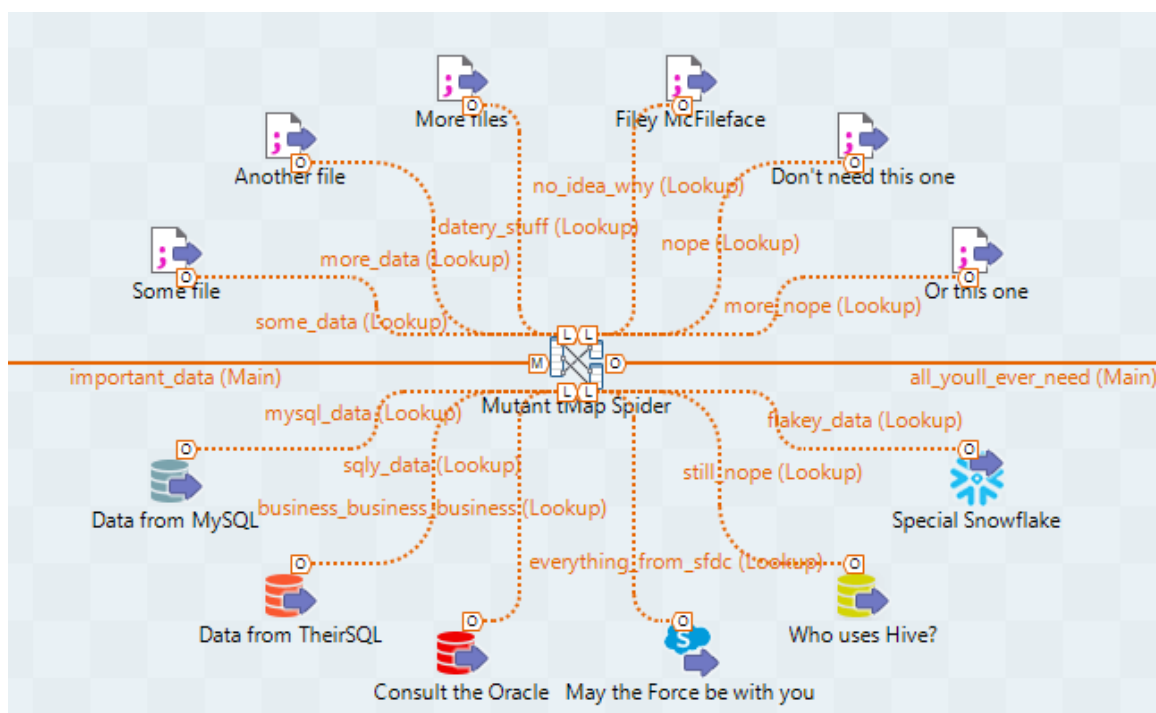


# 1. Mutant tMap spiders

Too many lookup inputs in a tMap, often coming in from all points of the compass, can make a Talend job look like something pulled from a shower drain. And just as nice to deal with. Add to this a multitude of variables, joins and mappings, and your humble tMap will quickly go from looking busy and important to a pile of unmanageable spaghetti.

A common solution, and one we advocate, is to enforce a separation of concerns between several serial tMaps, rather than doing everything at once. Maybe have one tMap to transform to a standard schema, one for data type clean-up, one to bring in some reference tables, etc. Bear in mind though that there's a performance overhead that comes with each tMap, so don't go overboard and have 67 all in a line!

At the end of the day, each job, sub-job, and component should be a well-defined, well-scoped, readable, manageable and testable unit of code. That's the dream, anyway!



*Special thanks to our Talend experts Kevin, Wael, Ian, Duncan, Hemal, Andreas and Harsha for their invaluable contributions to this article.*

This article was originally published on [Datalytx](https://www.datalytx.com).

<https://www.talend.com/blog/2019/12/05/10-things-youre-doing-wrong-in-talend/>