Dependency Field Guide

by

a softer space

Dependencies – can't live with them, can't live without them

A Softer Space DE, Inhaber Tom Moya Schiller | Heerstraße 3 b | 60488 Frankfurt am Main, Germany

Latest version of this document:
asofterspace.com/tools/dependencyFieldGuide.pdf

Phone: +49 176 511 27307
Email: info@asofterspace.com
Website: asofterspace.com
VAT Nr: DE319451065

DEPENDENCY FIELD GUIDE

8th August 2019

Software projects rise and fall with their dependencies.

Consciously using several key dependencies can speed up the creation of your application and even make it more secure and more maintainable by keeping the complexity within your own code to a minimum.

It is a fine line though between this – and utter chaos, in which code from who knows where suddenly pops up in the most unlikely places, and no one dares touching any parts of the code anymore as literally anything else might break. Needless to say maintainability goes right out the window: One necessary update to a dependency breaks the integration of some external library, and when fixing it that cool new component you just added yesterday stops working, and on and on and on…

Contents

It's All Relative

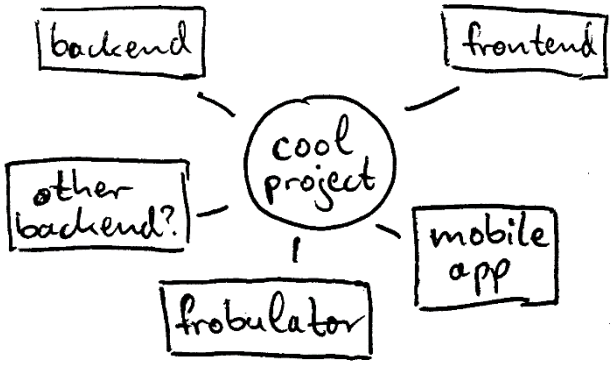
You can have more dependencies, making it quicker to get started – but possibly causing more trouble during updates.

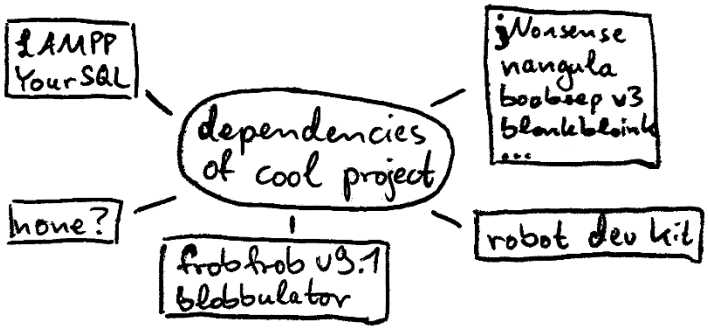
You can have no dependencies at all, avoiding version conflicts during updates – but all the responsibility will be on your shoulders, and it will take ages just to get a prototype out of the door, let alone a real product.

In the end, there is **no simple rule** that you can stick to – avoiding dependencies at all costs will lead to madness just as quickly as adding every single piece of code you can find on the internet to your project. Instead, for each and every dependency you have to **consciously check** whether it, individually, is a good fit for your project – and in addition to that keep the overall amount and layout of dependencies in mind.

You can draw the **layout** of your dependencies by drawing a map of your entire project's parts, and then drawing a second map in which you just write down the dependencies that are used by the corresponding parts of the project.

If you notice that one particular part is using a whole cluster of dependencies, then it might be worthwhile to look into it in particular to see if it can be simplified.





Maintainability, Maintainability, Maintainability!

We often think about a software project in terms of how quickly it can be created; we imagine a few weeks or months of coding to get the general product ready, then we do some A / B testing, tweak things a bit here and there, ship it – and we are done.

But if you actually want people to use your product for a while, let alone keep selling it over years, then the original **effort of creating it** in the first place quickly becomes negligible compared to the amount of work that goes into **maintaining it**. Source code is only written once but read many times. And sadly:

> *It's harder to read code than to write it.*
> *~ Joel Spolsky*

This includes updates of your external dependencies, where the migration to new interfaces is often harder than it was to just add it originally. It also includes going through existing code again and again – during that phase of tweaking the application just a little bit here and there, and when you are adding that hot new feature everyone is talking about, and when you are making changes in anticipation of that huge client who you hope might want to start using your product then.

Overall, great external dependencies can keep the amount of code small that you need to maintain and make your life easier – while bad ones will lead to myriad problems later on.

Focus on what will make your life **easier in the future**.

A year from now, will you be glad that this particular dependency is in your code, as it will simplify your work – or will it actually be in your way, require a lot of updates or even replacement?

Recognizing Great External Code

These are a few things to look out for when thinking about adding a new external dependency.

Likewise, when you are deciding whether to keep an existing one or to replace it, you can consult this list.

1. It should actually **benefit you** in some tangible way.
   Yes you can also add this and that and the other thing – but why would you? How does it help you achieve your goal?

2. It should be released under a **software license** that fits into your existing ecosystem.
   Are you building a proprietary piece of software and this dependency is copyleft? Or did you find some source code of a competitor and are planning of just copying a couple classes from them – what could possibly go wrong?[1]

A **copyleft** license commonly requires that software built with this dependency should be released under the exact same license, thereby keeping the software downstream free to use and open source.

If you cannot release your own software project under that same license, then you will have to check carefully whether including that dependency is allowed or not –a distinction may e.g. be made between linking into an external module as opposed to fully integrating code into your application, but this has to be examined on a case by case basis.

[1] The answer is "a lot." **A lot** can go wrong when you illegally take someone else's code – just don't do it.

3. It should be **actively maintained**, ideally by more than just one person.
   Without any active maintainers, it will be solely your responsibility to adapt it to new environments and to fix security issues.

4. It should be built on **mature technology**.
   If the technology is so cutting-edge that huge changes are expected on a daily basis, then you will have to reintegrate this dependency again and again.

5. It should **not drag in** a whole bunch of **other dependencies**.
   Several of your dependencies relying on the same transitive dependencies can be efficient if they both accept fairly large ranges of versions, such that you do not get into a deadlock.

6. It should come with **useful documentation**.
   Tutorials and examples by other people online can be a substitute for "official" documentation if the dependency is widely used and lots of these are available.

Useful documentation is especially important when **migrating** from one version of the dependency to another.

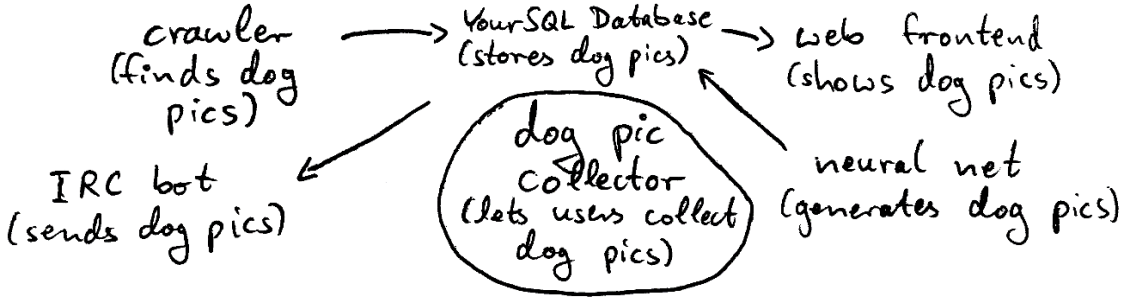
You can check if the dependency already has switched from one major version to another – maybe it is in version 3 now but was in version 2 for many years before. Look at how that switch was handled, how well the migration was documented and how many people were confused online.

This can give you confidence about how well future migrations will be handled that you might be affected by.

It All Went Wrong… Now what?

You wake up one day, take a look at your project, and realize it has all gotten out of hand. When showing it to someone new, you have a hard time even starting to describe what some of the parts are doing. You are afraid of important updates becoming available to your dependencies, as updating their versions is likely to leave you in an inconsistent state, requiring days of work to just get it to compile again.

The most immediate need is to **gain an overview** – try to write down in one sentence what your project is actually doing, at its core. Then, make a **map of all the parts** and for each in turn shortly write down how it helps your project achieve its goal.



Now the **same for the dependencies**: Note every single dependency that you have, how it got pulled into your project, why – and most importantly: Whether it is actually, in the long term, good to have it, or not.

Dependency	How?	Why?	Helpful?
jNonsense	random script file on server	I think we are using it in the user acquisition process... right?	meh...
org.foobar.supercool	Maven (parent pom)	displays cat GIFs	definitely!
leftpad	central repo	pads things left	not if we only pad right
random	copied from xkcd	returns 4	yes, if we need 4

Most likely the situation is not yet as bad as you might think – a few changes here and there, getting rid of a few really problematic dependencies, maybe redesigning or documenting a few confusing parts; and you will have a much clearer path ahead.

Dependency Cheat Sheet

Dependencies should make your life **easier in the future** – to ensure this, for each and every dependency **consciously check** whether it is a good fit.

It should:

1. Actually benefit you
2. Be released under a suitable software license
3. Be actively maintained
4. Be built on mature technology
5. Not drag in lots more dependencies
6. Have useful documentation

In case of a dependency emergency, **gain an overview**:

1. Write down in one sentence what your project is actually doing
2. Make a map of all the parts, and write down how they help your project achieve its goal
3. For each dependency write down:
   - How it got pulled in to your project
   - Why it is in your project
   - Whether it is helping you or standing in your way

If you need help, contact us at info@asofterspace.com – we are glad to look at your project, scratch our heads and shrug. 😉

Good judgment is
the result of experience.
Experience is usually
the result of bad judgement.
~ Uncle Zeke