Composer package version strategy

This is a discussion about how to use Composer

Situation: As of 2017-08-26, composer.json has a mixed usage of the Version Constraints for Composer dependencies. For example, jQuery was updated from 3.1 to 3.2 without anyone really noticing (and all went well). Yet, for others like PivotTable.js, we are locked at v2.7.0, while upstream is at 2.14.0 A jQuery upgrade is arguably more risky than for PivotTable.js

What should be our strategy? Recommended background reading: Why your software project will slowly die without continuous updating

Goals

Some of these goals conflict, so it will be about balancing trade-offs

- Keep trunk always up to date with libs
- Reduce workload (please do keep in mind we have over 100 dependencies!)
- A testing process to detect issues with upgrades
- Tiki release script should indicate what libs were updated since last release
- Get updates from upstream
- Have only minimal changes to Tiki while in the last years of the Long Term Support period. (ex: year 3, 4 and 5: only get security updates
- Avoid dependency issues
- Take advantage of tools like https://www.versioneye.com/

Potential strategies

Strategy #1

- Trunk always up to date within version (5.5 -> 5.x)
- Major upgrades done manually (ex.: 5 -> 6)
- After x years, LTS branches are locked to current version
  - Only security fixes are done, so put in place a process to update packages for security fixes

Pros:

- Constantly staying up to date

Cons:

- Things can break in the background while no one is looking. Could be hard to debug.
Strategy #2

- No automatic upgrades
- Developers upgrade packages after having tested them
- Put in place a process to update packages for security fixes

Pros:

- Less chance of an upgrade breaking something

Cons

- Not taking advantage of enhancements

Strategy #3

- Trunk always up to date within minor version (5.5.2 -> 5.5.x)
- Major (5 -> 6) and medium (5.2 -> 5.3) upgrades done manually
- After x years, LTS branches are locked to current version
  - Only security fixes are done, so put in place a process to update packages for security fixes

Pros:

- Less chance of an upgrade breaking something

Cons

- Not taking advantage of enhancements
- Doesn't apply for some libs, because they don't have 5.5.2 -> 5.5.x, but only 5.5 -> 5.x

Strategy #4

- No global strategy. It's a case by case choice by developers. Even if this is the choice, it should be more explicit and some global pros and cons should be listed to help developers choose.

Pros:

- Provides flexibility to choose according to the context (although exceptions can be made with the other proposed strategies)

Cons

- Inconsistency
• Developers don't know what is appropriate (not great for new developers)