Semantic Versioning

Telling others how much your product has changed by defining an API

Ben Christel, Oleksandr Diachenko, C.J. Jameson
Palo Alto Tech Talk
August 29, 2017

Pivotal

We’re going to talk about...

- The motivation for semantic versioning: Dependency Hell
- Semver in a nutshell
- Illustration: The “Fruit Tree” API
- How to use semver in db development
- Semver is not a silver bullet - use with caution

The Two Types of Dependency Hell

Dependency specifications are too loose

or

Dependency specifications are too strict

http://semver.org

Semantic Versioning is a scheme for telling others what has changed about your product.

Semantic Versioning is a scheme for expressing to others what has changed about your product.

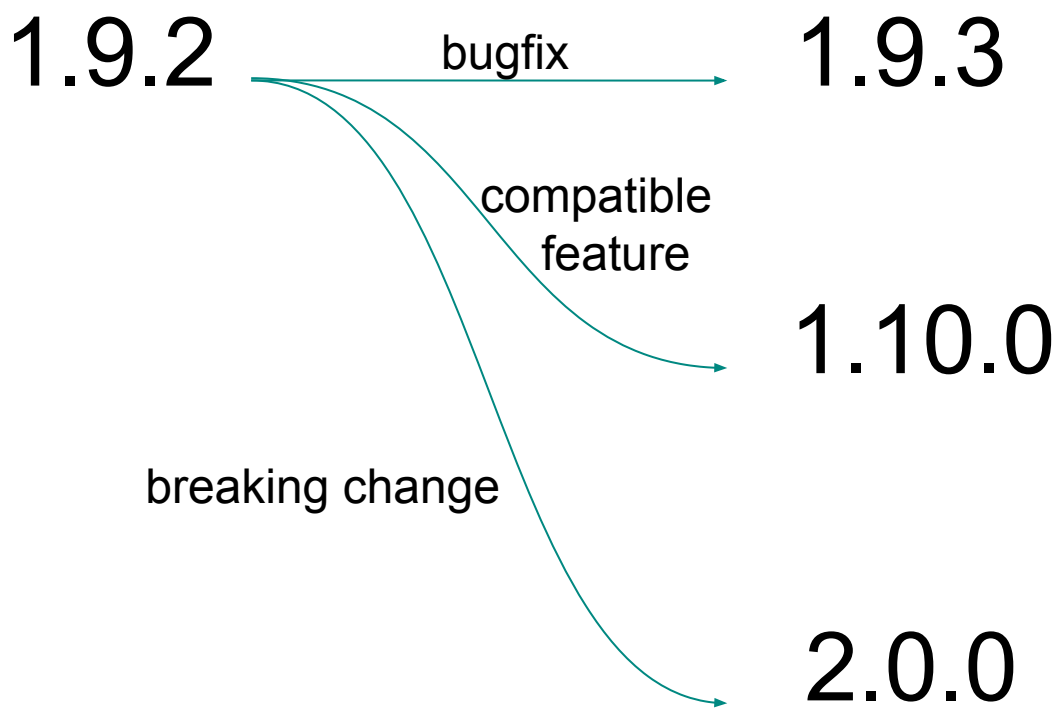
- Semantic versioning is machine-friendly. Package managers can automatically install "safe" updates while avoiding updates that introduce breaking changes.
- It's human-grokkable: in most cases, it is easy to intuit the degree and kind of changes that occurred between two versions.

Semantic Versioning looks a lot like 3 numbers separated by dots

1.9.2

MAJOR MINOR PATCH

Semantic Versioning assigns meaning to *changes* in each part of the version string



Major version 0 means the API is not yet stable

0.1.2

After the three digits, additional text can describe release or build process information

1.9.2-beta4

Semantic Versioning is not:

- automatic on the package author's side. You must know what changed since your last release in order to appropriately increment the version number.
- a replacement for release notes. You will need to document your API and changes to it so your users understand what they can rely on and what they can't.

The “Fruit Tree” API

Create, Read, Update, and Delete some apples, oranges, etc.

Step 1: defining our public API for an initial version of a product

Method/path	Input	Output	
		200	500
GET api/v1/apples		[{“id”: “id1”, “name”: “Big Apple”, “color”: “red”}, ...]	[{“error_code” “code”, “error_message” : “There are no apples“}]
GET api/v1/apples/id		{“id”: “id1”, “name”: “Big Apple”, “color”: “red”}	[{“error_code” “code”, “error_message” : “There are no apple with such id“}]
PUT api/v1/apples	[{“name”: “Big Apple”, “color”: “red”}, ...]	[{“id” : “id1”, “name”: “Big Apple”, “color”: “red”}, ...]	[{“error_code” : “code” “Unable to create an apple”}]
POST api/v1/apples	[{“id” : “id1”, “color”: “green”}, ...]	[{“id” : “id1”, “color”: “green”}, ...]	[{“error_code” : “code” “Unable to update an apple”}]
DELETE api/v1/apples		[{“id”:”id1”}, {“id”: “id2”},...]	[{“error_code” : “code” “Unable to delete apples”}]
DELETE api/v1/apples/id		{“id” : “id1”}	[{“error_code” : “code” “Unable to delete an apple”}]

Step 2: release first stable version

1.0.0

Step 3: add optional attribute to some of the fruits, defined in API

{“id”: “id1”,
“name”: “Big Apple”,
“color”: “red”,
“weight”: “0.5lb” }

Required attributes
Optional attributes

Step 4: release new version, bumping up minor version

1.1.0

Step 5: add new required attributes, add support for a new fruits

Method/path	Input	Output	
		200	500
GET api/v1/oranges		[{“id”: “id1”, “name”: “Clockwork Orange”, “color”: “red”}, ...]	[{“error_code” “code”, “error_message” : “There are no oranges“}]
GET api/v1/oranges/id		{“id”: “id1”, “name”: “Clockwork Orange”, “color”: “green”}	[{“error_code” “code”, “error_message” : “There are no orange with such id“}]
PUT api/v1/oranges	[{“name”: “Big Apple”, “color”: “red”}, ...]		[{“error_code” : “code” “Unable to create an orange”}]
POST api/v1/oranges	[{“id” : “id1”, “color”: “green”}, ...]		[{“error_code” : “code” “Unable to update an orange”}]
DELETE api/v1/oranges		[{“id”:”id1”}, {“id”: “id2”},...]	[{“error_code” : “code” “Unable to delete oranges”}]
DELETE api/v1/oranges/id		{“id” : “id1”}	[{“error_code” : “code” “Unable to delete an orange”}]

Step 6: releasing next version

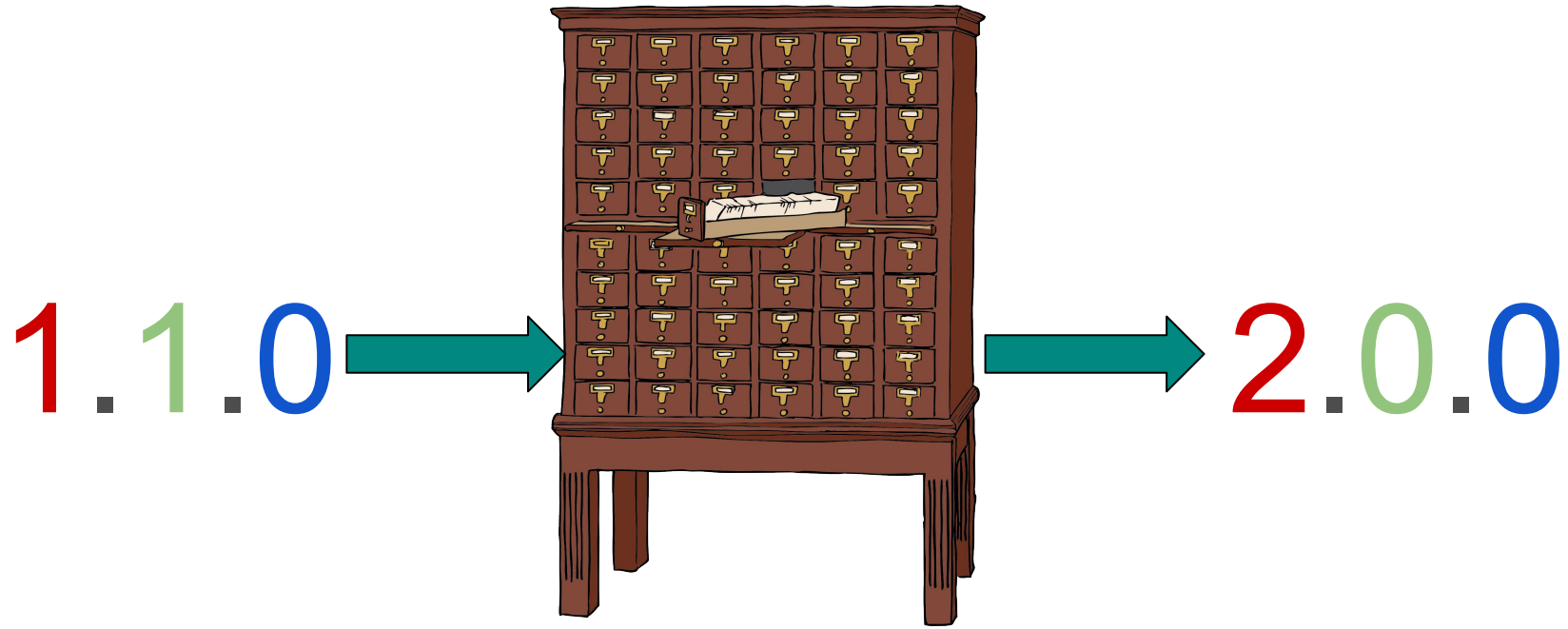
- Option 1: keep two API paths, for a backward compatibility - release 1.2.0 version
- Option 2: migrate to v2 API completely so it will be breaking change - release 2.0.0 version

To use semantic versioning for a Database product, we include things like the catalog in our definition of the API

What defines your API?

For GPDB, the catalog will only change if the major version changes

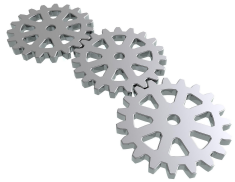
Catalog change



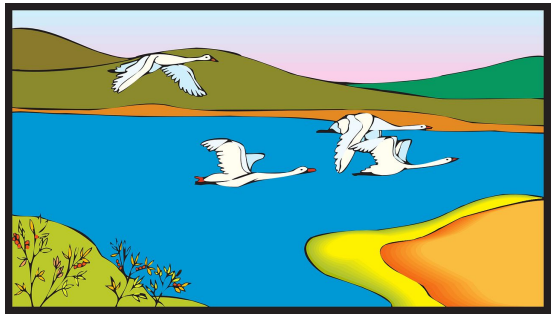
For databases, a major version bump often would require data migration or more intense processes to avoid data loss

1.1.0

System data

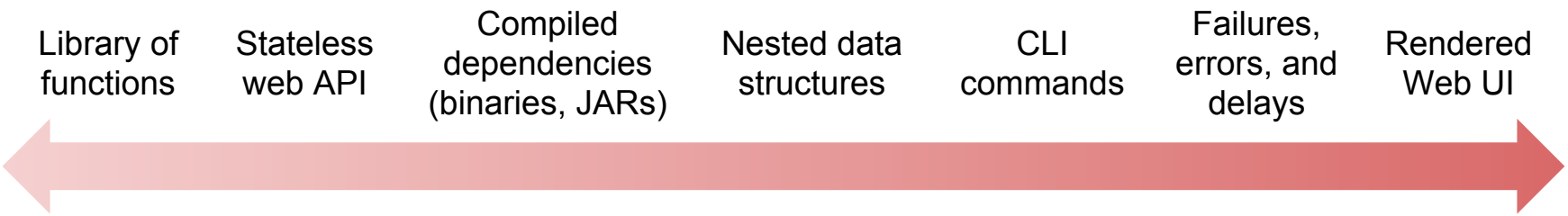


User data



2.0.0

But we also think more broadly about the user interface and user experience… can semver account for this?



Golly, it can be difficult

- Acceptance-regression tests
- Standards and conventions like HTTP status codes
- Consolidated output and logging streams

What can you document? What would you document?

Semver is not a silver bullet - use with caution

Ideal: Users' tools (operating systems, package managers, etc.) can automatically apply updates with confidence based on the change in version

Ideal: Behavior of your software can be clearly described because the code-level interface is the only interface, such as a code library

Ideal: Description of your interface is concise; users don't depend on coincidental or undocumented behavior

Semver is not a silver bullet - use with caution

Ideal: Users' tools (operating systems, package managers, etc.) can automatically apply updates with confidence based on the change in version
Reality: Tradeoff between desired updates and healthy skepticism of dependencies; can pin your dependencies in your build system if so

Ideal: Behavior of your software can be clearly described because the code-level interface is the only interface, such as a code library
Reality: External interfaces are not always APIs… UI elements are more complex; even terminal/CLI interfaces are tricky to settle on

Ideal: Description of your interface is concise; users don't depend on coincidental or undocumented behavior
Reality: API definitions might communicate the happy path features, but any feature can fail in many ways. Users may come to depend on failure mode. Consistent HTTP status codes and consolidated logging might help

Some build system tools rely on semver or take advantage of it when gathering dependencies

- Apache Maven (not strict - semver is supported, but other formats are allowed)
- Rubygems/Bundler (not strict - semver is supported, but other formats are allowed)
- Swift package manager (SPM)
- Node package manager (NPM) (strict - you must use semver)
- Elm package manager (very strict - you must use a subset of semver AND the package repository verifies through Elm's type system that you only changed the stuff your version number says you did)

Thanks! Questions?