Ginkgo 2.0 Proposal

The first commit to Ginkgo was on August 19th, 2013. Much fruitful community feedback - and code cruft - have accumulated over the years. Both can be addressed with a new major version of Ginkgo: Ginkgo 2.0.

Here's the original 2.0 proposal - this document is now considered stale as active development has begun. The 2.0 GitHub issue has the most up-to-date context and links. In addition to commenting on what's *in* the proposal. Please consider what might be *missing* and comment on that here. Private comments can be sent to onsijoe@gmail.com

Table of Contents

Rough Timeline

✅	~July 30	Private access to proposal for Ginkgo users/committers (k8s e2e, CF)
✅	~Aug 15	Public access to proposal + Ginkgo github issue

✅	~Sep 1	Align on 2.0 feature-set and start development
✅	~Sept 2021	First beta of 2.0
✅	~October	Ship 2.0 RC
	~October	Ship 2.0 GA

The Overarching Meta

Guiding Principles

1. Ginkgo 2.0 will **not** be a rewrite. We should be able to get to Ginkgo 2.0 in ~2-3 months.
2. As such, Ginkgo 2.0 will be an incremental iteration on Ginkgo 1.0. We will take the opportunity to break a few public interfaces to reduce code cruft and implement some much-asked for features...
3. ...but we won't break too many. The goal is for all users to migrate their tests from Ginkgo 1.0 to Ginkgo 2.0 within 6 months and for most migrations to either be a no-op or to take O(minutes) of developer effort.
4. We want to deliver on some of the most long-standing requests in the Ginkgo backlog...
5. ...and to remove functionality that is either little-used or dangerous.
6. We'll gather community feedback along the way, and ship a number of RCs before GAing 2.0...
7. ...but we'll err on the side of action and shipping over analysis-paralysis.

Non-Goals

1. This proposal and work is focused on Ginkgo. **Not** Gomega.
2. While the native `testing` framework has improved over time - and one can imagine rewriting aspects of Ginkgo on top of new functionality in `testing` - we will avoid doing so for a few reasons:
    a. This is intended to be an iteration, not a rewrite.
    b. Ginkgo exerts fine control and holds strong opinions over the test lifecycle, particularly around randomization and parallelization, and this could prove challenging to construct on top of `testing`.

c. We want to avoid fundamentally changing the model so as to limit developer effort to migrate from Ginkgo 1.0 to Ginkgo 2.0.

d. There are alternative frameworks out there that leverage `testing` more directly to provide BDD-style testing (eg https://github.com/sclevine/spec)

3. Boiling the ocean.

Releasing Ginkgo 2.0

The official Go Modules blog posts recommend placing major version bumps in separate subdirectories of a package (e.g. `/v2`). However this increases the maintenance burden on code maintainers who must maintain the original version alongside the new major version (in perpetuity?).

As outlined below the cost of migrating to Ginkgo 2.0 should be relatively low for the majority of developers. As such we will bump the master branch of `github.com/onsi/ginkgo` to 2.0 after a period of betas and release candidates. Developers who need access to v1.0 can pull a specific tag. Alternatively, for developers that cannot specify a specific version in their dependencies we may introduce a new package `github.com/onsi/ginkgo-v1-deprecated` which will be maintained for a few months while users migrate. We would appreciate community feedback on this approach.

Proposed Deletions

Ginkgo 2.0 will remove the following functionality from Ginkgo:

Deletion: Async Testing Functionality

Priority
High

Context
Ginkgo's support for asynchronous testing documented here is considered dangerous and should not be used.

In Ginkgo 1.0 developers can pass an anonymous function to an `It` block of the form:

```
It("does something asynchronous", func(done Done) {
    c := make(chan string, 0)
```

```
    go DoSomething(c)
    Expect(<-c).To(ContainSubstring("Done!"))

    close(done)
}, 0.2) // timeout is 0.2 seconds
```

Typically, Ginkgo runs the anonymous functions passed to `It` blocks on the main goroutine. When passed a function that takes a `Done` channel, however, Ginkgo launches the function in it's own goroutine and passes it a `Done` channel that it expects to close within the configured timeout. If the channel is not closed, the test is marked as failed due to a timeout.

This approach to asynchronous testing is naive and dangerous. In particular, it can result in test pollution as timed out goroutines cannot actually be terminated by Ginkgo and will leak past the lifecycle of the test. If the goroutine eventually gets unblocked while a *different* test is running all manner of surprising and poorly defined behavior can occur (e.g. it can mark a different test as failed; it could alter shared test state from underneath the currently running test).

The correct way to test for asynchronous behavior is with Gomega's `Eventually` construct. `Eventually` offers richer asynchronous testing semantics that significantly reduces the risk of test pollution.

Proposed Change

We propose removing support for asynchronous testing in Ginkgo completely. This will entail:

1. Removing the exported `Done` channel type.
2. Removing support for running `It`s asynchronously.

Migration Plan

Users will need to rewrite their asynchronous tests to correctly use Gomega. We expect that asynchronous tests are rarely used - however if sufficient evidence warrants it, we could deliver a migration tool that maintains the same (broken) behavior and gives developers time to more correctly refactor their tests. For example, our tool could rewrite the asynchronous test above to the following:

```
It("does something asynchronous", func() {
      done := make(chan interface{}, 0)
      go func() {
          c := make(chan string, 0)

          go DoSomething(c)
          Expect(<-c).To(ContainSubstring("Done!"))
```

```
        close(done)
    }()

    Eventually(done).Should(BeClosed(), 0.2)
})
```

This is functionally identical to the previous implementation but makes the asynchronicity more explicit. From here developers could manually massage their code to make it safer:

```
It("does something asynchronous", func() {
    c := make(chan string, 0)

    go DoSomething(c)

    Eventually(c).Should(Receive(ContainSubstring("Done!")), 0.2)
})
```

Deletion: `Measure` Functionality

Priority
High

Context
Ginkgo includes a facility to benchmark code. `Measure` nodes run as part of the test suite and invoke and measure aspects of their passed in function.

We believe this functionality:

1. Is infrequently used.
2. Is out of place - Ginkgo is a test framework particularly oriented around integration testing. Benchmarking is best accomplished via other means including, potentially, libraries that interoperate with Ginkgo.
3. Is thin and immature. It gives you just enough to leave you begging for much, much, more.
4. Adds complexity to the codebase.

Proposed Change
As such, we propose removing support for `Measure` nodes and benchmarking in Ginkgo.

Migration Plan

None. Users would need to rewrite their benchmarks. We do not anticipate that this feature is highly used but would love feedback from the community if folks disagree.

Deletion: `ginkgo convert`

Priority
Low

Context
`ginkgo convert` allows users to convert existing go test suites into Ginkgo test suites. It does this by somewhat naively transforming all `TestX` functions into top level `It`s. The value added by `ginkgo convert` is low and there is little evidence that it is widely used. Removing it will simplify the codebase.

Proposed Change
We propose removing support for `ginkgo convert`.

Migration Plan
None needed.

Deletion: `ginkgo -notify`

Priority
Low

Context
`ginkgo -notify` uses terminal-notifier to emit desktop notifications on MacOS. We anticipate this feature is rarely used.

Proposed Change
We propose removing `ginkgo -notify`.

Migration Plan
None needed.

Proposed Additions and Changes

Ginkgo 2.0 will add the following functionality

Ordered Contexts

Priority
High

Context
A long-standing GitHub issue discusses adding support for `BeforeAll` and `AfterAll` to Ginkgo. The intent behind `BeforeAll` and `AfterAll` is to support test setup and teardown for a set of tests that *only* runs once - the `BeforeAll` before any tests run and the `AfterAll` after all tests have run.

Historically adding support for `BeforeAll` and `AfterAll` has been considered an anti-pattern. Doing so would (a) promote sharing state between `It`s (and thereby risking test pollution) and (b) break Ginkgo's implicit contract that `It`s are completely independent and therefore can be run in parallel.

However, requests for `BeforeAll` and `AfterAll` continue to come in. The primary motivating factor is to enable expensive set-up and tear-down code to run just once for a collection of tests. While Ginkgo's philosophy will continue to center on clean separate `It`s that can be parallelized aggressively, it's time for pragmatism to prevail.

So, we propose adding support for `BeforeAll` and `AfterAll` to Ginkgo 2.0.

Proposed Change
We propose adding support for `BeforeAll` and `AfterAll` by introducing a new `ginkgo.Context` (henceforth, simply `Context`) construct, the `OrderedContext`.

`OrderedContexts` are collections of `It`s that make the following guarantees:

1. The `It`s in an `OrderedContext` **MUST** run in the order that they appear in the test.
2. The `It`s in an `OrderedContext` **MUST** run serially with respect to each other but **MAY** run in parallel to other tests (including tests that occur in other `OrderedContext`s)
3. `OrderedContexts` **MAY** include additional `Context`s and `OrderedContexts`. The `It`s defined in these nested containers always exhibit the same behavior outlined above in points #1 and #2.
4. `OrderedContexts` **MAY** be nested inside other containers (e.g. `Context`s, `Describe`s etc.)

`BeforeAll` is a new construct that **MUST** be defined within an `OrderedContext`. It makes the following guarantees:

1. The anonymous function passed to the `BeforeAll` (henceforth "function") is called at most once.
2. The function runs before any `It`s defined in the `OrderedContext`.
3. The function runs before any `BeforeEach`es defined in the `OrderedContext`.
4. The function runs before any `JustBeforeEach`es defined in the `OrderedContext`.
5. If multiple `BeforeAll`s are defined, they are run in the order that they appear.
6. If a `BeforeAll` fails any subsequent `BeforeAll`s, `JustBeforeEach`es, and `It`s are **NOT** run. The only subsequent nodes called if a `BeforeAll` fails are `AfterAll`s defined at the same level in the testing hierarchy or higher (this is to ensure any cleanup functionality implemented by the user is still called).

`AfterAll` is a new construct that **MUST** be defined within an `OrderedContext`. It makes the following guarantees:

1. The anonymous function passed to the `AfterAll` (henceforth "function") is called at most once.
2. The function runs after any `It`s defined in the `OrderedContext`.
3. The function runs after any `AfterEach`es defined in the `OrderedContext`.
4. The function runs after any `JustAfterEach`es defined in the `OrderedContext`.
5. If multiple `AfterAll`s are defined, they are run in the order that they appear.
6. If an `AfterAll` fails subsequent `AfterAll`s are still run (this is to ensure any cleanup functionality implemented by the user is still called).

Invoking `BeforeAll` or `AfterAll` outside of the immediate body of an `OrderedContext` generates a runtime error.

Note that an `OrderedContext` **MAY** include a `BeforeAll` or an `AfterAll` but does not require either.

Example

```
var _ = OrderedContext("order matters here", func() {
      BeforeAll(func() {
            fmt.Println("BeforeAll")
```

```
	})

	BeforeEach(func() {
		fmt.Println("BeforeEach")
	})

	It("A", func() {
		fmt.Println("A")
	})

	It("B", func() {
		fmt.Println("B")
	})

	Context("a nested context", func() {
		//it is a runtime error to call BeforeAll or AfterAll here

		BeforeEach(func() {
			fmt.Println("BeforeEach Nested #1")
		})

		It("C", func() {
			fmt.Println("C")
		})

		AfterEach(func() {
			fmt.Println("AfterEach Nested #1")
		})
	})

	OrderedContext("an ordered nested context", func() {
		BeforeAll(func() {
			fmt.Println("BeforeAll Nested")
		})

		BeforeEach(func() {
			fmt.Println("BeforeEach Nested #2")
		})

		It("D", func() {
			fmt.Println("D")
		})
```

```
            It("E", func() {
                  fmt.Println("E")
            })

            AfterEach(func() {
                  fmt.Println("AfterEach Nested #2")
            })

            AfterAll(func() {
                  fmt.Println("AfterAll Nested")
            })
      })

      It("F", func() {
            fmt.Println("F")
      })

      AfterEach(func() {
            fmt.Println("AfterEach")
      })

      AfterAll(func() {
            fmt.Println("AfterAll")
      })
})
```

When run this example will **ALWAYS** (regardless of parallelization or randomization) emit:

```
BeforeAll
BeforeEach
A
AfterEach
BeforeEach
B
AfterEach
BeforeEach
BeforeEach Nested #1
C
AfterEach Nested #1
```

```
AfterEach
BeforeEach
BeforeAll Nested
BeforeEach Nested #2
D
AfterEach Nested #2
AfterEach
BeforeEach
BeforeEach Nested #2
E
AfterEach Nested #2
AfterAll Nested
AfterEach
BeforeEach
F
AfterEach
AfterAll
```

Exclusive Contexts

Priority
High

Context
There are certain scenarios where some tests simply cannot be run in parallel with any other tests. Ginkgo 1.0 does not provide any clean mechanism for indicating that a test or set of tests **MUST NOT** be run in parallel with any other tests. Users are often forced to pull these out into separate packages and ensure they never invoke `ginkgo -p` on that package.

Proposed Change
We proposed adding a specialization of `Context` called `ExclusiveContext`.

`ExclusiveContexts` are collections of tests that meet the following contract:

1. The `It`s in an `ExclusiveContext` **MUST** run serially with respect to **ANY** test defined in the suite. (i.e. no other tests can run while the tests defined in an `ExclusiveContext` are running, but `It`s within an `ExclusiveContext` may run concurrently with each other)

2. `ExclusiveContexts` **MAY** include additional `Describe`s, `Context`s and `OrderedContexts`. The `It`s defined in these nested containers always exhibit the same behavior outlined above in point #1.

3. `ExclusiveContexts` **MUST** be defined at the top-level and **MUST NOT** be nested in other containers.

Per #3 it is a runtime error to define an `ExclusiveContext` anywhere but the top level.

From an implementation point of view, we expect the tests in an `ExclusiveContext` will always run on Parallel Node #1 (nodes are one-indexed) **after** all other tests have completed.

Test Labels

Priority
High

Context
It is common for large Ginkgo-based integration test suites (e.g. the Cloud Foundry Release Integration test suite, the Kubernetes End-to-End tests (e2e)) to grow significantly as the tested codebase expands. Organizing these suites and enabling users to precisely select which subset of tests to run has grown increasingly complex with time.

Today, users rely on `ginkgo`'s `--focus` flag to pass a carefully crafted regular expression that selects the right subset of tasks. Ginkgo matches this regular expression against the concatenated description strings (i.e. all the strings contained in any parent container nodes and the description string in the `It` itself) to determine whether or not a given test should run. This has proven error-prone and frustrating.

To alleviate this problem, we propose introducing test labels to Ginkgo.

Note: We're using the word `label` instead of `tag` to avoid confusing Ginkgo labels with Go build tags.

Proposed Change
First a high-level summary:

Test labels allow users to annotate container nodes (i.e. `Describe`, `Context`, etc.) and test nodes (i.e. `It`) with labels. The final set of labels associated with a given test is the union of all the labels in its parent containers, and the labels on the `It` node itself. Command line support enables users to precisely select which set of labels to run or skip during a test-run. And the set of labels is available at run-time via `CurrentGinkgoTestDescription`().

Labels are of type `Label`:

```
type Label string
```

and are passed into Ginkgo nodes as slices: `[]Label`. The convenience function `Labels()` allows users to efficiently construct these label slices.

Now the details:

Creating Labels

An individual Label is a string of type `Label`:

```
type Label string
```

To annotate tests, a set of labels represented as `[]Label` must be passed to a Ginkgo node. Users use the `Labels()` convenience function to to construct these sets:

```
func Labels(args ...interface{}) []Label
```

The `Labels` convenience function can take any number of `Label`, `string`, `[]string` or `[]Label` arguments. The output will be a deduped slice of `[]Label` that contains the union of all passed in labels.

Annotating Tests

To allow passing in test labels the signature for container nodes and test nodes will change to:

```
func Describe(text string, args ...interface{})

func Context(text string, args ...interface{})

func It(text string, args ...interface{})
```

While this loses some degree of type-checking at compile time (which we will mitigate with clear run-time feedback) it provides a flexible and productive user experience when writing tests and avoids breaking existing code. To annotate a test, users simply need to pass in a `[]Label` parameter as one of the function arguments. They typically construct this slice inline using the `Labels()` convenience function.

Here are some examples:

First - note that existing Ginkgo tests will compile just fine. There will be zero cost to migrate code given these public interface changes:

```
var _ = Describe("An existing Ginkgo test", func() {
      It("will run just fine", func() {
            Expect(CurrentGinkgoTestDescription().Labels).To(BeEmpty())
      })
})
```

Second - users can annotate a test using the `Labels` convenience function. For example:

```
var _ = Describe("An annotated Ginkgo test", func() {
    It("will be annotated", Labels("label 1", "label 2"), func() {

Expect(CurrentGinkgoTestDescription().Labels).To(ConsistOf(Label("label
1"), Label("label 2")))
    })
}
```

Will result in a test with the labels `["label 1", "label 2"]`.

Third - users will be able to reuse common labels with ease. For example:

```
var commonLabels = Labels("common label 1", "common label 2")

var _ = Describe("An annotated Ginkgo test", func() {
    It("will be annotated", Labels("label 1", "label 2", commonLabels),
func() {

Expect(CurrentGinkgoTestDescription().Labels).To(ConsistOf(Label("label
1"), Label("label 2"), Label("common label 1"), Label("common label 2")))
    })
}
```

will result in a test with the labels `["label 1", "label 2", "common label 1", "common label 2"]`. Note that we are relying on the `Labels` convenience function to take care of merging the `commonLabels` slice with the bare `"label 1"` and `"label 2"` strings.

Fourth - users can attach labels to containers. For example:

```
var _ = Describe("An annotated Ginkgo test", Labels("container label 1",
"container label 2"), func() {
    It("will be annotated", Labels("it label 1", "it label 2"), func() {
```

```
Expect(CurrentGinkgoTestDescription().Labels).To(ConsistOf(Label("container label 1"), Label("container label 2"), Label("it label 1"), Label("it label 2")))
    })
}
```

will result in a test with the labels `["container label 1", "container label 2", "it label 1", "it label 2"]`. This will allow tests within a shared describe/context to share the same set of parent labels. Note: it will not be possible for a child node to remove a label added by a parent node.

Lastly - the proposed Ginkgo node function signatures could potentially allow users to shoot themselves in the foot with poorly formed function invocations. We will apply the following rules when calling `Describe`, `Context`, and `It`:

1. There **MUST** be at least one variadic argument passed to the function.
2. The last variadic argument **MUST** be a niladic void function.
3. At most one other variadic argument is allowed and it **MUST** have type `[]Label`

Breaking any of these rules results in a run-time panic.

Listing Labels

The default Ginkgo reporter will be updated to emit the list of test labels associated with any test. This will appear any time the test description is emitted.

This means that the combination of `ginkgo -v -dryRun` will cause Ginkgo to emit the set of all tests including all their labels without running tests and can be used to view the set of all labels.

Selecting Tests by Label

Ginkgo currently supports `--focus` and `--skip` flags. These will remain with their existing behavior unchanged. i.e. Regular Expressions passed to these flags will be matched against the full test descriptions (i.e. the concatenated set of container and child description strings) to identify which tests to run and skip. These will **not** be extended to support matching against labels.

Instead, we introduce a new flag: `--labelFilter` which takes a label filter.

The label filter provides a simple grammar for selecting labels that supports:

- Boolean && operations for requiring the presence of multiple labels (AND).

- Boolean `||` operations for requiring the presence of one set of labels or the other (OR).
- Boolean `!` operations for negation (NOT).
- `(` and `)` for grouping labels.

Some examples:

```
> ginkgo --labelFilter="Label A"
// Requires that tests have Label A to run.

> ginkgo --labelFilter="!Label A"
// Runs any test that does not have Label A.

> ginkgo --labelFilter="Label A && Label B && Label C"
// Requires that tests have all three of Label A, Label B, and Label C to
run.

> ginkgo --labelFilter="Label A || Label B || Label C"
// Requires that tests have any of Label A, Label B, or Label C to run.

> ginkgo --labelFilter="(Label A || Label B) && !Label C"
// Requires that tests have either Label A or Label B but not Label C to
run.
```

Users can combine `--focus`, `--skip`, and `--labelFilter`. The three requirements will simply be ANDed together.

Make Table Testing a First Class Citizen and Adding Support for Labelled Table Tests

Priority
Low

Context
The table testing extension under [extensions/table] has seen broad adoption. We propose making it a first class part of the Ginkgo DSL.

Proposed Change

We propose moving `DescribeTable` (as well as `FDescribeTable`, `PDescribeTable`, and `XDescribeTable`) and `Entry` (as well as `FEntry`, `PEntry`, and `XEntry`) to the top-level exported Ginkgo DSL. We will also remove the `extensions/table` package.

In addition to promoting the table functionality we will add support for test labeling by allowing users to pass a single `[]Label` argument as the first variadic argument to `DescribeTable` and `Entry`. Such an argument would be detected and interpreted as a set of labels to assign to the generated Ginkgo nodes.

Migration Plan
Users will need to delete their import of the extensions/table package.

Improve `ginkgo` CLI Flag Documentation

Priority
Low

Context
The command-line documentation for the `ginkgo` CLI tool's flags has become crufty and difficult to parse. There are two problems:

1. Several `ginkgo` flags are simply pass-throughs down to `go test`. It's not clear which flags are tied to Ginkgo functionality and which are tied to the underlying `go test` tool.

2. The Ginkgo-specific flags fall naturally into categories of functionality (see, for example, the groupings presented in the online documentation) - the current flat list obscures this structure making it harder for users to reason about which arguments connect with which features.

Proposed Change
We will update the `ginkgo` CLI to emit more structured flag documentation. This will make it easier for users to understand which flags apply to `ginkgo` vs `go test` and to better understand which areas of functionality the different flags apply to.

We will not change any flag names so this will not be a breaking change.

Improve `GinkgoTInterface` to Ensure Broader Compatibility

Priority
Low

Context
As documented here Ginkgo provides a `GinkgoT` object that conforms to the `GinkgoTInterface`. The intent with this object is to enable compatibility with testing libraries that expect a `testing.T` object and can accept it through an interface. Over the years, both `testing.T` and (more recently)

`testing.TB` have evolved. It's time to update `GinkgoTInterface` to match the latest object definitions in `testing`.

Moreover, most of the methods in `GinkgoT` are unimplemented. However some *could* be implemented to improve compatibility (e.g. `Skip` and `Name`).

Proposed Change

We will extend `GinkgoTInterface` to provide as much of the `testing.TB` and `testing.T` definitions as possible. We will also better flesh out the `GinkgoT` methods where the `testing` semantics have Ginkgo equivalents. For example, calling `GinkgoTInterface.Skip` will invoke Ginkgo's own `Skip` functionality.

Note that much of this is inspired by this Ginkgo issue.