# Dependency Heaven

- **Speaker: Alex Miller**
- **Meeting: EuroClojure - July 2017**
- **Video: https://www.youtube.com/watch?v=sStlTye-Kjk**

[Time 0:00:00]

```
slide title: Dependency Heaven

Alex Miller, Cognitect
```

So I am Alex Miller, and I have been working with Clojure for about seven years, and working *on* Clojure for about four years at Cognitect. And I work on the core team and help develop new features, and things like that.

This is actually the first time I have done a technical talk at a Clojure conference, because usually I am MCing [Master of Ceremonies] them. So you get the first one.

[audience applause]

So this talk is called "Dependency Heaven", which maybe is not a place that we will actually reach. That may be an unattainable goal.

[Time 0:00:37]

```
slide title: Dependency Hell

Entire slide background is covered with yellow and orange flames.
```

But it is really in opposition to dependency hell. This was really just a fun opportunity to make the slide.

[Time 0:00:43]

```
slide title: What are the problems?

+ Version resolution
+ Artifact-focus
+ Getting started
```

Which is something that a lot of developers describe as sort of the nuts and bolts of their daily existence of managing dependencies on their project, and using dependencies, and things like that.

And this is something that we have actually been working on, the core team, some different things, off and on for probably eight months or more. More seriously for the last month or two, really, because it is in the line of trying to get Clojure 1.9 out the door. And I will get to why that is here in a little bit.

But what are the problems? There are actually several clusters of problems related to dependency management. And I have sort of bucketed these into version resolution, our artifact focus, and getting started.

[Time 0:01:33]

```
slide title: Version management and resolution

+ See: Rich's Spec-ulation keynote
tbd link
+ "Change" is really either
  + Growth - accretion, relaxation, fixation
  + Breakage - requiring more or providing less
+ If software only grows, the newer version is always ok to use
+ Avoid breakage by making new names (-> accretion)
+ Use specs to make commitments
```

[ transcript of Rich Hickey's Spec-ulation talk]

So this is a one slide summary of Rich's Spec-ulation keynote from the Conj last year. It is well worth your time to go watch the hour plus talk version of Rich talking about these subjects, much better than I will be able to. But this is an important background, so I wanted to at least talk about a few different things.

One of the key things in his talk is really about this notion of change, which is a word that we throw around a lot in software. And he wanted to distinguish between two different kinds of change: growth and breakage.

Where growth is really accretion, which is adding new things that you can use. Relaxation, which is where you either require less from people that call you, or provide more to people that call you. Those are ways that do not break existing code. Or fixation, just actually fixing the implementation of something without changing its interface or things like that.

And then breakage is something that people also call change. And that is when you require more from your consumers. Either they are calling new APIs, or you require them to call a new API, or you require them to pass you more stuff, or you provide less on the return. Those are all kinds of changes that basically break an existing consumer's use of your code.

And the key insight here is that if software only uses growth, only uses those techniques, it is always OK to use a newer version. There is never any problem with sort of hitting a breaking version, because your software does not break. And this is something that we strive for in the Clojure API, which is one of the reasons that it has a lot of stability. For a long time, that has been sort of implicit. This is sort of making that more explicit, and something that we want to be more serious about, and would like other people to be more serious about, too. Because breaking software sucks, and so we should just stop doing that.

[Time: 0:03:32]

So one way to avoid breakage is by making new names. So we should not be afraid to make a new name for an existing function, give it some new name, and leave the old one there. Existing consumers continue to use it. New consumers can use the new one. If you need to remove a function, or significantly modify things, put it in a new namespace. That is fine. So there are ways to do this at sort of a collection level as well.

And one tool that we have in [Clojure] 1.9 now is spec. Specs allow us to make commitments about our code. And specs are really about what you *can* do, not about what you *cannot* do. And some of the things you cannot do in spec are specifically designed around avoiding prohibiting things. Prohibiting is a thing that turns growth into breakage. So if you suddenly need to allow a new attribute, and before we had a spec that said you were not allowed to have that attribute, then you have turned growth into breakage. So we do not want to do that. That is the wrong direction.

And then Rich also talked at length about semantic versioning, and how semantic versioning basically has these three levels of major, minor, and patch changes. And change at the patch level is just fixing.

It is really fixation. Change at the minor level is really accretion or relaxation. And change at the major level is breakage.

And so Rich's point there is that semantic versioning itself is broken, in that it allows breakage. You do not have to break things. You can just add instead.

[Time 0:05:10]

```
slide title: Dependencies

+ Dependencies declared at the artifact level
+ Artifacts are stored in Maven repos
```

So dependencies. This is moving into the second part, around artifact focus. Right now we declare dependencies at the artifact level. We declare that we depend on core.async, or compojure, or whatever. And those artifacts are stored in Maven repos.

So there are some good things about this. The Maven repo is an immutable, append-only, distributed store. All of that is awesome. That is the same things we like about persistent data structures, about git, about Datomic. All of those things have the same property of being immutable append-only stores.

So like last year, when Clojars had some down time, it is conceptually, architecturally easy to add a mirror for that stuff, because it is just caching. So you are taking this stuff and I can cache it somewhere else, and I can choose what I do with those caches. Caching is a really easy thing conceptually, compared to building a distributed system that has to deal with updates and deletes, things like that.

[Time 0:06:14]

```
slide title: The good

+ Maven repo is an immutable append-only distributed store
+ Access to vast Java ecosystem of libraries
```

The Maven repo system also gives us access to the vast Java ecosystem of libraries that already exist. That is goodness. We do not want to get rid of those things.

[Time 0:06:24]

```
slide title: The bad

+ Clojure doesn't care at all about artifacts
+ Clojure doesn't need to be "built" to be useful
+ Yet we force all code to be bundled into artifacts
+ We have entangled dependency mgmt and artifact production
```

The bad things. Clojure does not actually care about artifacts. When you start the JVM, you do not talk about artifacts. You are building a class path. Clojure does not actually require stuff to be built at all. You just need Clojure files that are on the class path somewhere.

And so we have sort of added this constraint around how we use Clojure code that is artificial. It has the up side of allowing us to play in the Maven ecosystem and do those kinds of things, but it has the down side of disallowing certain things that the JVM allows, and that Clojure supports, that would be useful.

We have also sort of entangled this problem of dependency management – how we say which code we need to use – and artifact production: our build systems, our build tools, and things like that. Those two things are entwined in something like Leiningen or Boot or Maven. All of those kinds of things.

There are lots of cases where we run into this restriction. I do not know how many people have needed a database driver that was not in a public Maven repo. That is a common thing. So that is an example of: I might have a jar file. I want to use it on my class path, but I cannot get it out of a repo. And in that case, you have to do all sorts of crazy workarounds to make that work. And that is dumb.

Or you might run into the case where you have your project split into multiple code bases, right? Your application is split into multiple projects. And in that case you have to do this cross project dev, and you use checkouts or something like that, to deal with that. But that is silly that we have to do that. There is no reason that we cannot start a REPL, and include all of these different source files that are all over the place, and work with them directly. The JVM does that. Clojure does that. Our artifact focus has prevented us from doing that.

And then the other thing is that the source of truth for most of us now is files in git on Github. And we cannot just use those things directly right now. But it would be useful to do so. And so I think that is something that also – there are Clojure files in Github that we could get to, and we do not really care about the fact that they get packaged into artifacts. They might be useful without that.

[Time 0:08:46]

```
slide title: The ugly

+ Our dependency declarations are lies
+ We rely on Maven to select versions to put in our uberjar or
  classpath
+ Version selections are subject to (broken) semantic version rules
```

And the ugly part of it is that our dependency declarations are lies. They are actually not even true. What we say when we specify a library is: we are requesting this version of a library. But when we go actually build a class path, or we build an uberjar, or whatever, you are actually running a Maven algorithm that is going to expand the full tree, and it might find multiple versions of a library. And it is going to pick a version for you based on a bunch of constraints and rules, some of which are based on semantic versioning, which has these issues that we have talked about.

So it is very possible, and actually likely, for you to end up with a set of versioned artifacts on your class path that are a set of versions that you never specified, and that you have never tested with. That is actually a pretty easy thing to happen. And so that is pretty ugly.

We would really like things that we do right now – things like dependency management in Maven, or now in Leiningen it was added within the last year or so, is really the ability to specify an external source of truth for dependency versions. We should be able to do that. That is just a thing that is useful, and there are lots of cases for that.

[Time 0:09:57]

```
slide title: Getting started

+ As of Clojure 1.9, Clojure itself has multiple artifacts
+ A new user is now immediately confronted with how to build a
  classpath
```

So moving on to the third set of problems: the getting started experience. For a long time you have been able to just download the Clojure jar and run clojure.main directly out of the Clojure jar. And that was maybe not the friendliest thing that you could do, but it was at least relatively easy to do.

As of Clojure 1.9, Clojure itself has multiple artifacts now. Clojure depends on spec. It also depends on the Clojure core specs library. So there are actually at least three jars that you need in your classpath to start Clojure. So a new user is immediately confronted with this problem. It is no longer – we have sort of moved past the point where that was possibly tolerable. This seems like something that we had to have a better solution for in 1.9.

And that is how this has gotten into the top of my queue, basically. Because my primary goal right now, as a top line goal, is getting 1.9 out the door, and this is in the way of that. That is the reason I have been working on it lately.

[Time 0:11:00]

```
slide title: What are we going to do about it?

+ Working on a new contrib library tools.deps.alpha
+ A Clojure installer that uses tools.deps.alpha
+ System-specific installer packages (brew, apt, chocolatey, etc)
```

So what are we going to do about it? So we have been working on a new contrib library tools.deps. Dot alpha initially. I expect that relatively soon that we will move out of alpha, and we will move into a more stable release. I am not sure if that will happen before 1.9 or after 1.9. It is really not particularly important for most of the other stuff.

The Clojure installer, that can provide a way to get the latest version of Clojure and tools.deps onto your system. The Clojure installer does not assume Clojure, so it is not written in Clojure. It is written in Java. We do make that assumption – that you are going to get Java on your machine somehow before hand.

And then system specific installer packages for brew, or apt get, or chocolatey, or whatever is the right thing for your system.

So these are the things that I have been working on, and a substantial portion of this now exists and works, and will be released soon, whenever I have time to do all of that.

[Time 0:12:03]

```
slide title: tools.deps.alpha

slide title: What are the high-level constraints?

+ JVM needs a classpath
+ Have to do transitive dependency traversal
+ Download artifacts from Maven repos
```

So tools.deps.alpha. The high level constraints here are – the sort of ground truth of what we have to deal with is that when we start the JVM, it takes a class path. And so we have to be able to build a class path.

We have to be able to do transitive dependency traversal. That is just a fact of life. Dependencies depend on other things. We have to be able to download artifacts from Maven repos. So those are all things that we know we are going to have to do.

[Time 0:12:28]

```
slide title: High-level process

+--------------+        +-------------+         +-----------+
| Initial deps | =====> | Library map |  =====> | Classpath |
+--------------+        +-------------+         +-----------+

          _resolve-deps_              _make-classpath_
          expand transitive deps
          resolve version differences
          download artifacts

                    ^                         ^
                    |                         |
                    |                         |
              add extra deps            override path
              override deps
          default deps
```

Sort of the high level process that we have settled on after a lot of discussion is: starting from an initial set of dependencies, applying this operation called resolve-deps that can expand the transitive dependencies, resolve version differences, fail on version conflicts if you find those, and download those artifacts to your actual system.

And the result of that is going to be a thing called a library map, which I will talk more about. There are also some extra use cases there on the bottom. I will also talk about those more.

And then a second operation, make-classpath, that really takes a library map and then produces the actual class path out of that, which is the simpler of the two operations.

So what are the steps in resolving deps? We have to start with the initial deps, expand the transitive dependency graph, resolve these version differences, and return this library map.

[Time 0:13:25]

```
slide title: What are the use cases we currently see?

+ Add extra deps - add things to initial deps
+ Override deps - override selected version
+ Default deps - default version in the case where none is specified
```

Three extra use cases that we have drilled into – we looked at a lot of ways that people use Leiningen profiles, and different features out there in the wild today, and these are the three use cases we focused on.

The need to add extra dependencies when we are creating a class path. And that is common when you are doing development time, or you are at development time, and you are trying to add some extra testing libraries. You might want to add criterium to do benchmarking, or whatever. Something like that. So it is not part of our core dependency set. It is not what you need to use the software, but it is an extra library that I want to use right now, for whatever purpose.

We have the problem of overriding dependencies. So say I know that I am going to have this dependency show up somewhere in the expanded dependency graph, and I am going to tell you that I am going to use exactly this version of it. And I do not care what version comes out of all of that stuff. I am going override to just this one.

And then default dependencies. In the case where you have a dependency that shows up in the dependency graph, and it does not have a version, or does not have a version specified, then use this version. I will tell you what version to use by default.

And then I think a fourth case, maybe, that needs a little bit more thought is exclusions. Rich and I have not had a chance to talk about that in depth, so I will table that one.

[Time 0:14:54]

```
slide title: Library map

+ Represents the complete set of all resolved libraries
  + Key = a library (user's intent)
  + Value = a coordinate (version + impl to use) + path on disk
```

So the output of all of this is a library map. The library map has, as a key, the library – the user's intent to use Compojure. As the value, it has a coordinate, which is both a version, and some particular implementation to use, and an actual path on disk of where I can go find it. And so it is implicit in this thing that part of doing resolve-deps is that it has actually got to download a jar, or make a path available somewhere on disk.

[Time 0:15:28]

```
slide title: Making a classpath

+ What do we need?  A classpath.
+ What is the classpath?
+ What are its characteristics?
```

So making a class path. Our goal here is to generate a class path, which is just a path separated string. What is a class path? A class path is an ordered list of class path roots. Those roots can be jar files. They can be directories. There is also some special syntax to include like a glob of jars. But the key thing is that it is either a jar file or a directory.

What are its characteristics? And this is something we went and looked at. And one of the things that you notice immediately when you look at it is:

[Time 0:16:02]

```
slide title: Classpath order

+ Is classpath order important?
+ What use cases are actually important?
```

a class path has an order. It is a sequence of things. And that order is important because that is the order that the JVM will look in those different roots for things.

So the question is: is that order actually important, though? Is it actually relevant for the class path? So if all of your classes exist only in one of your roots, then it is not relevant. The JVM will find it across one of those roots, and it really does not matter.

The cases where it is relevant is when it is in more than one of those class path roots. I will contend that most time this happens, it is actually a bug. How many people have debugged something where you had the same class in multiple jar files? That is awful, right? It is really hard to figure out, and

it most commonly happens when you accidentally AOT compile a downstream dependency class into your jar file, or something like that.

[Time 0:17:01]

So I am going to contend that that is a bug, and we should not do that. And if it does happen, it is a problem, not an intended use case.

The other case that you do see fairly frequently is: including another class file into your class path in a different root, specifically for the purpose of overriding the one that is already there. And so that one is kind of interesting. It is like: I have this class path. I am getting this class file from somewhere, but I actually want to use a different version of that. I am going to override it with a specific version.

So our contention is that that is really a good use case, and we should be sure to support that. But the way that people do that with prepending is really more of a hack than doing what they want to do, which is to say: I want to go to this particular thing in the class path and replace it. So we should support replacement. That is fine. But we do not really have any goal or interest in supporting prepending for the purposes of overriding.

So if you get rid of that, then we can say basically that order is not that important. And so we are going to give up on trying to maintain the ordering in a class path, and instead try to provide other operations that give us that same functionality.

[Time 0:18:20]

```
slide title: clj

+ New script for invoking clojure.main
+ Goal: java <jvm-args> -classpath <cp> clojure.main <args>
+ clojure.main can run a repl, run a -main, run test, eval exprs
+ Main question: how do we construct <cp>?
```

So around tools.deps we have a script. The script is called clj. The real point of clj is effectively to run clojure.main. And you run clojure.main by running Java. So Java has some JVM args. We need to have a class path. That is where we will do more work. We are going to invoke clojure.main, and then take args for clojure.main.

clojure.main can be used to run a REPL. That is the default with no arguments. It can be used to run a Clojure expression, and just give you the result back. And it can also be used to run the "dash main" [-main] function in any namespace. So you can use it as a general runner as well. And it could be used to run tests, even, that way, with a little bit of additional glue code.

So that is all useful. The main question here is: how do we construct the class path?

[Time 0:19:16]

```
slide title: deps.edn

+ clj will use a deps.edn file in the current directory if it exists

{:deps
  {
    org.clojure/clojure {:type :mvn :version "1.8.0"}
    org.clojure/core.async {:type :mvn :version "0.3.442"}
  }
}
```

And the first thing we need is some way to remember which dependencies we care about in the current context, the current directory. And we are doing that by storing that in a file called deps.edn, which is obviously an edn file. And it has a number of different keys in this map. The :deps key is the main one, and that will store a mapping from the library to a coordinate.

You will notice that this is slightly different than the syntax in Leiningen, or Boot, or other places. And there is a reason for that. We are trying to add some additional context here about where this particular library is coming from. And right now everything is going to be coming from Maven.

But we are sort of expanding the ability to build pluggable systems that pull artifacts, or directories, even, from other locations. So imagine there are other kinds of types there as well, like files, and maybe Github. Things like that.

[Time 0:20:18]

```
slide title: Aliases

+ deps.edn can include aliases to cover any combination of
  resolve-deps args:

{:deps ...

  :aliases {
    :bench {
      :exra-deps {criterium {:type :mvn :version "0.4.4"}}
    }
    :1.9 {
      :override-deps {org.clojure/clojure {:type :mvn :version "1.9.0-alpha17"}}
    }
  }
}
```

deps.edn can also include aliases, to cover any combination of these different use cases that I talked about. :override-deps, :extra-deps, things like that. It looks like I have a typo in there on :extra, but ...

So here I have created a couple of different aliases. One called :bench that includes criterium. I also included one for 1.9. You will see a lot of times in Leiningen profiles, people will have profiles for a bunch of different Clojure versions, to do matrix testing type stuff. And that is something that is easy to do here as well.

It could be done even at a sort of user wide level, rather than at a project level.

[Time 0:21:01]

```
slide title: Coordinates

+ Library version coordinates contain a type indicator

+ While only Maven is supported initially, there are lots of
  interesting things to do with expanding deps directly from files or
  even Github

+ This system is open and can be further extended to other artifact
  sources
```

I mentioned coordinates a little bit. The library version coordinates contain this :type indicator. I have already mentioned most of this stuff. That :type flag ends up flowing down through some multimethods to an extensible system.

[Time 0:21:17]

```
slide title: clj aliases

+ resolve-deps aliases can be used with clj -R (concatenated alias
  names):

clj -R:bench:1.9
```

So this Clojure script also takes some extra arguments that allow you to activate aliases that are used to build the class path. And so this -R will give you some additional resolve arg aliases that can pull in additional functionality. And then you can use multiple of these as well, so a command line like this will actually pull in both the benchmarking alias, which included criterium as an extra dependency, and 1.9 as a way to override my Clojure dependency and use a specific version of Clojure.

I think that these things can be sort of mixed and mashed and combined to cover a wide variety of use cases. Most of the things that I am aware of out there.

[Time 0:22:08]

```
slide title: Classpath overrides

+ Either specified using aliases with -C

+ Or hard-coded at command-line with -P

clj -C:dev

clj -Porg.clojure/clojure=/Users/alex/code/clojure/target/classes
```

You can also specify these last minute class path overrides. The overrides that you get in making the class path, to say I want to specify a hard coded directory location, or jar file, to use instead of whatever you found during the transitive dependency walk.

You can either use -C to pass an alias that maps to something that is in the deps.edn file, or you can use -P to build an explicit map actually on the command line directly. And so you can sort of swap something in if you need to debug with a specific version, or something like that.

[Time 0:22:44]

```
slide title: Classpath caching

+ Both the intermediate libs map and the final classpath are cached

+ If the classpath given the flags exists and is newer than deps.edn,
  then cache file is used instead

+ In this way, can often avoid invoking JVM with tools.deps.alpha
```

Another benefit we get here is that, one reason we built all of this stuff into aliases is to have names for single or combination of these things that activate the building of different class paths. And so both the intermediate libs map and the final class path are cached in the local directory, in a dot directory.

If the class path, given the set of flags that you specified, exists, and it is newer than deps.edn, so nobody has changed what those things mean, then that cache cp file can be used instead. And you can avoid invoking another JVM with tools.deps to build the class path. You already have the class path. It is sitting in a file. It is just a string in a file, and you can just spit it directly into the command line that is going to start Java.

[Time 0:23:33]

```
slide title: What clj does NOT do

+ Compile
+ Package
+ Deploy
+ etc
```

What clj does not do is AOT compile things. It does not package things into jars. It does not deploy things to Clojars. It does not do any of those things.

And that is pretty intentional. We want to disentangle these sort of build and artifact deployment kinds of use cases from the actual dependency management use case. So the initial audience here is really focused on new users that are just getting started, or any existing Clojure user who wants to do sort of more ad hoc REPL based development. Things like that. It will work really well for that.

I think there are vast opportunities to integrate these things with Leiningen, Boot, whatever, in a couple of different directions, either coming from those tools, or going to those tools. All of those things are just a little bit of glue code, I think, to do those migrations. I think it will require a little bit more work to figure out what the best way is to do those kinds of things, and where they make sense.

[Time 0:24:40]

```
slide title: Installer

+ Needed some way to get initial Clojure and tools.deps.alpha onto a
  machine

+ clojure-install is a small Java app that:

  + uses Maven Resolver to determine the newest stable version of
    Clojure and tools.deps.alpha

  + writes a system level deps.edn used when none exists in local
    directory

  + writes the classpath for using tools.deps.alpha itself
```

We also need some way to get all of this stuff onto the actual system, and so we have a new Clojure installer. This is a Java app that basically uses Maven Resolver, which is the thing that used to be Aether. That stuff has been pulled out of Eclipse and moved into an Apache project and is now called

Maven Resolver. But that is basically using the standard Maven APIs to find the newest stable version of Clojure and tools.deps.alpha. And so it is going to go out there and say: I am going to find Clojure 1.8. I am going to use tools.deps 0.0.3, or whatever version that is.

And it is then going to record that into a system level, really user level, deps.edn file. So you can remember that. And that will actually be used by the clj thing as well. So if you try to run the clj script, and you do not have a local deps.edn file, it will use this sort of user-wide deps.edn file. So you can just start a Clojure REPL anywhere, by just doing clj.

It also writes the class path for using tools.deps itself. So inside that clj script, if we determine that we need to build a class path, you have to then invoke tools.deps, which means you need to run a JVM, which means you need a class path, and you need this class path. And so that is a bit of work to do that.

But that is a one time thing to do that, and after that point you basically have tools.deps on your system. You actually have enough functionality at that point to effectively do that work again any time you need to. So this is sort of a bootstrapping operation.

[Time 0:26:33]

```
slide title: brew et al

+ brew formula for OS X (will be published to main brew tap):

+ Would welcome help making installers for other systems too
```

I have written a brew formula for OS X. I am on OS X, so that is what I care about most immediately, and what I can easily test. And so that is where I have started. Obviously there are a lot of other things as well, and I would welcome help in making those things, and we would love to start making Clojure packages on a regular basis.

We do not actually need to update this every time that you run Clojure, because the installer has the ability to find the newest stable version of Clojure. When 1.9 is released, the existing brew formula would actually find the newest version of 1.9 and use it, when you ran it. So the only time you really need to install a new version of the installer is when the installer itself changes, which is basically like fifty lines of Java code, or whatever.

The other thing it installs are the scripts, so the clj script itself. And so that would be the other kind of a change that would require updates.

But I envision that this is the sort of thing that, after the initial period of development, could probably go lengthy periods of time without being modified at all, across multiple Clojure releases. That is, at least, the goal.

[Time 0:27:54]

```
slide title: Future

+ More work on different provider types
+ Additional command-line assistance
+ More installers
+ Integration with build tools
```

So there is more work to do on the different provider types. I have done some tinkering. We have started to do some serious work around sort of using local directories, and depending on local directories

that have their own deps.edn file, and allowing that stuff to all work outside of any artifacts. And that is sort of "in work".

And then we have done some talking about utilizing git and Github, and things like that. I have not actually done any work on that yet.

It is an open protocol, so it is possible to add other things as well. If people have ideas, I would love to hear those.

There are a few different niceties on the command line that I would like to have that I have not added yet. Some more assistant around the format of the deps.edn file, and ways to dump the class path out of the cache, and things like that.

More installers I already mentioned, and integration with build tools I have already mentioned. So those are all things for the future.

[Time 0:29:01]

```
slide title: Thanks!

+ Coming soon:
  + https://github.com/clojure/tools.deps.alpha
  + https://github.com/clojure/clojure-install
  + https://github.com/clojure/brew-install
+ Contact:
  + alex.miller@cognitect.com
  + @puredanger
```

And I think that is it. That is all I have. And these things do not exist yet, but they will as soon as I get a chance to make them. And I think I have another ten minutes, so I am happy to take questions. But I am not going to run up the mike with you, because I can only do so much.

[audience applause]

[Time 0:29:26]