

# End-to-End And On The 'Level

Dave Gurnell, Underscore  
[@davegurnell](#)

I'm here to talk about writing applications using Typelevel libraries... end-to-end, using exclusively Typelevel libraries, insofar as that's possible.



This talk is about software stacks.

The idea and title came from a friend of mine, Kingsley. We were discussing possible talk proposals and he pointed out that it would be worth investigating this buzzword we'd been hearing a lot...

# “Typelevel Stack”

The phrase “Typelevel Stack” keeps coming up online. If people are using the phrase, they’re interested in the idea. And that idea seems worth investigating.

But is the “Typelevel Stack” a real thing, or are people just using the phrase because it sounds like something else...

# Typesafe Stack

The phrase “Typesafe Stack” has been around a long time, and “Typelevel” and “Typesafe” sound similar... so maybe this is just the natural evolution of language in play.

# What is the Typelevel Stack?

So the primary starting point for this talk is the question “Is there a Typelevel stack?”

And if so, what does it look like? What’s it like to work with? And how can we improve it.

# What is a software stack?

Thinking about this question led to other questions, like “What is a stack anyway?”

# Are software stacks important?

And “Are stacks a thing we should care about?”

I’m going to investigate all three of these questions here, because I think there are some interesting things here that we should be thinking about as a community.



# What is a software stack?

Let's start with this notion of a software stack.



“[...] a **set of software subsystems or components**  
needed to create a **complete platform**  
such that **no additional software is needed** [...]”

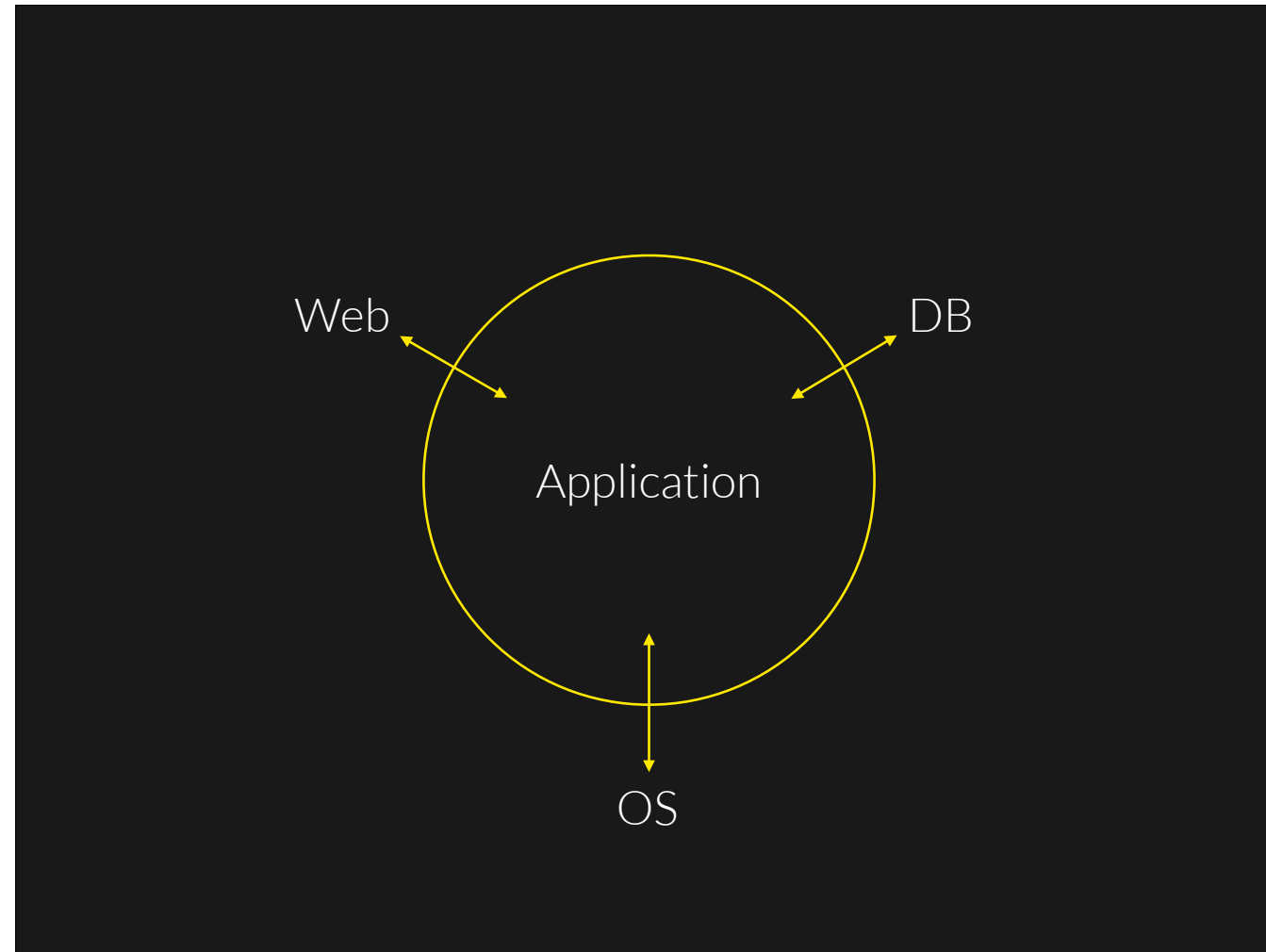
[https://en.wikipedia.org/wiki/Solution\\_stack](https://en.wikipedia.org/wiki/Solution_stack)

The standard place to look for any definition is Wikipedia, which defines a “solution stack” or “software stack” as:

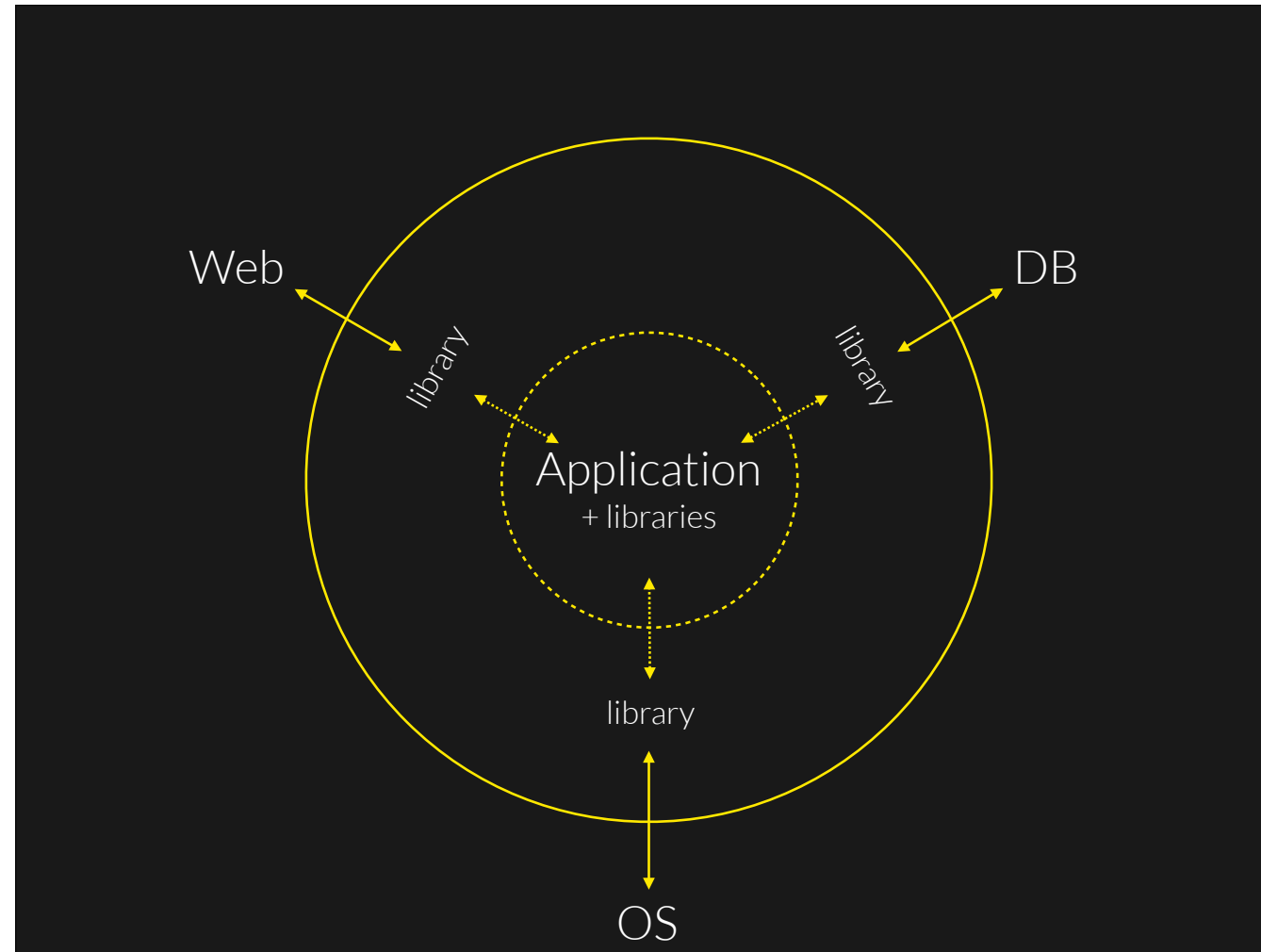
- a set of components;
- that create a complete platform;
- so that we don’t need anything else to write an application.

There’s an emphasis here on completeness and bespoke-ness—an architect chooses a stack to support his/her application, and that stack may evolve with the application over time.

But there’s also an undertone of compatibility—popular stacks arise because the components work well together.



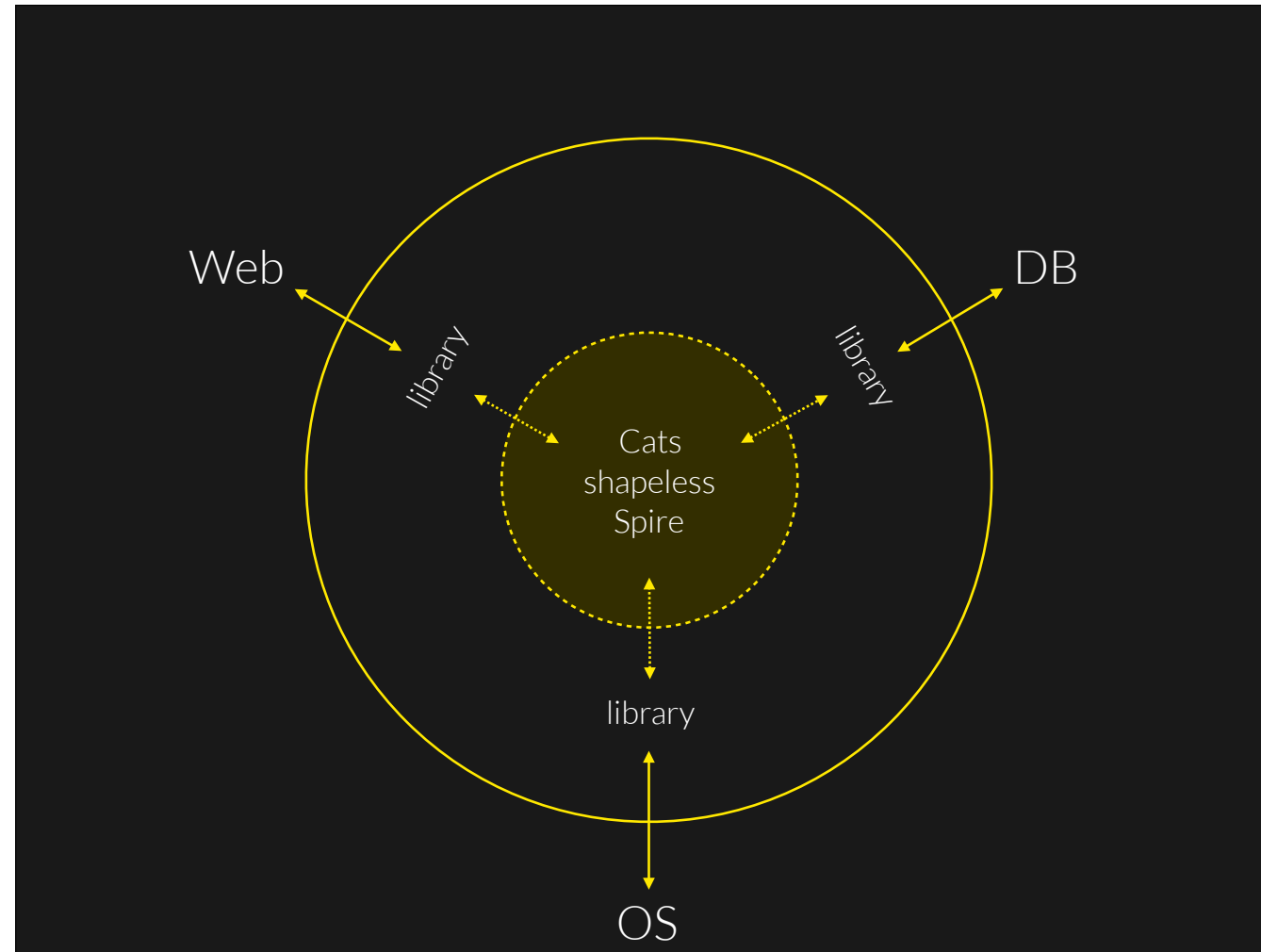
Here's a pictures of a simple stack as an architect would see it. We have our application talking to a set of external services.



From a developer's point of view, it looks more like this. Libraries fit into various points of the diagram.

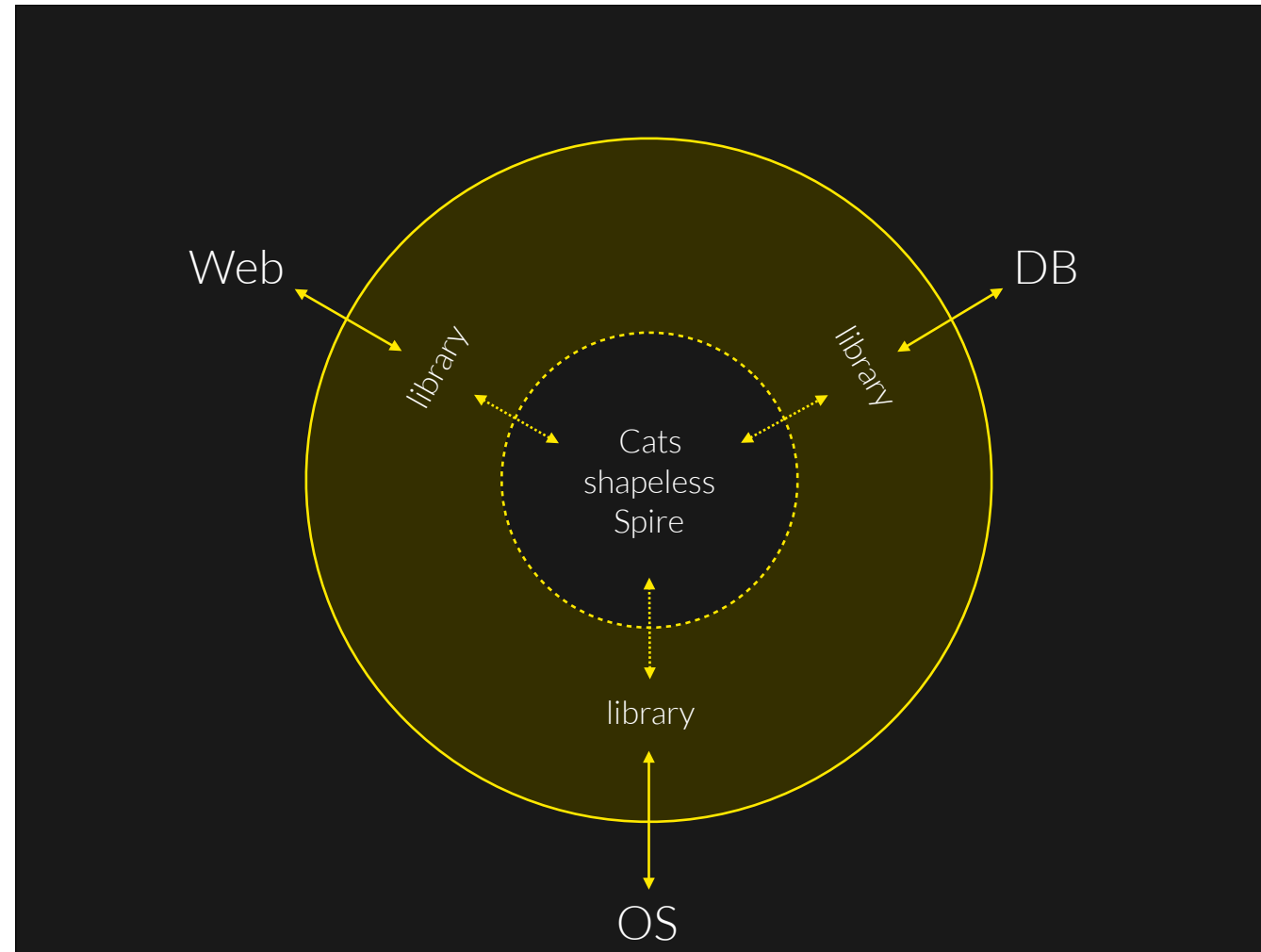
Our application talks to external services via libraries.

We also use libraries as the substrate of our application.



People wanting to buy into the Typelevel Stack are most likely interested in using Cats, shapeless, and Spire as the substrate of their application.

But the substrate isn't the whole stack.



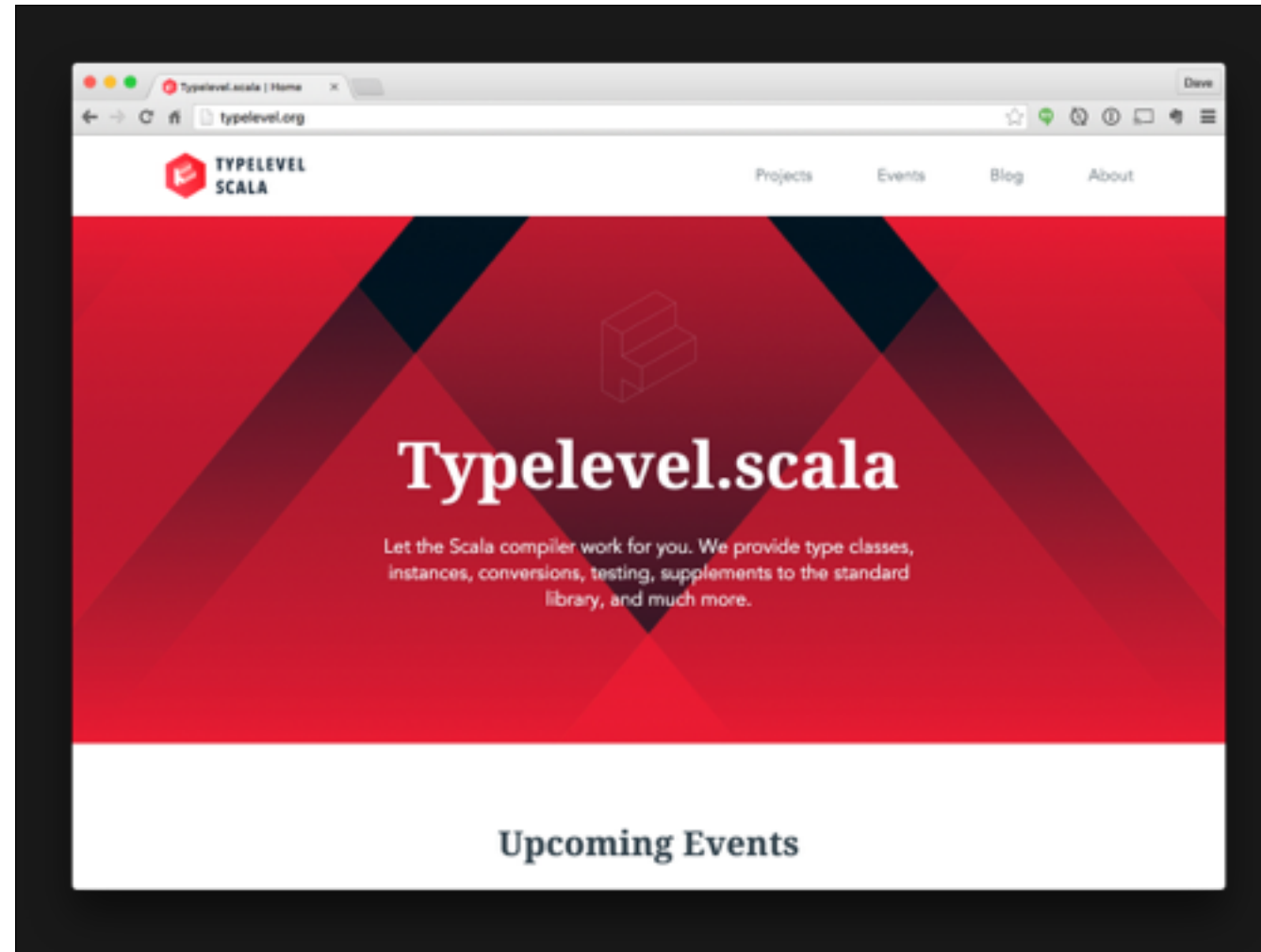
Service-level libraries—ones that talk to external services—need to communicate with the substrate and with each other.

The simpler that communication and the fewer the teething issues, the better.



# What is the Typelevel Stack?

With that definition of a stack in mind, let's turn our attention to Typelevel.

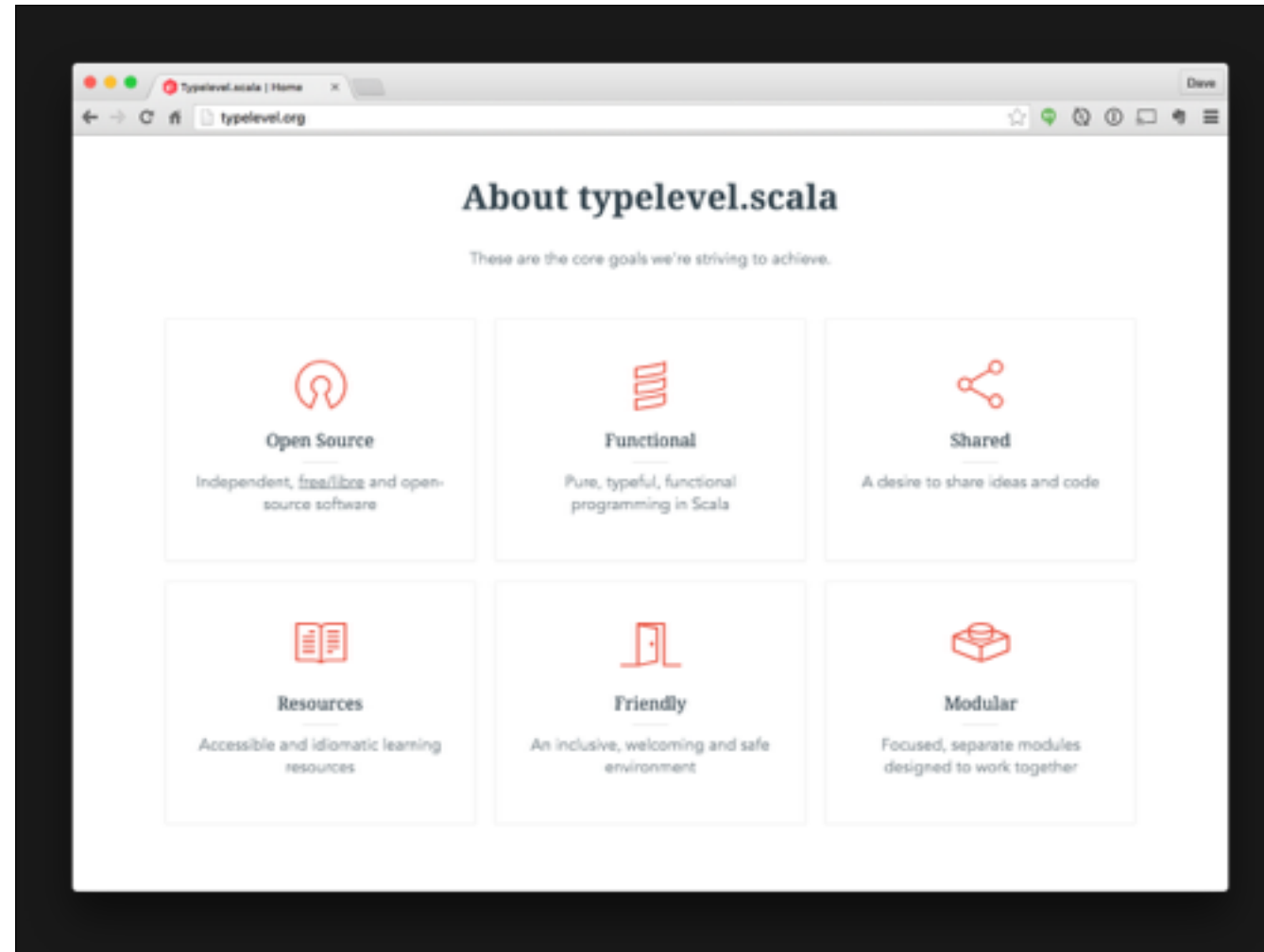


The first thing to mention is that Typelevel aren't actually making any claims about interoperability.

The strap line here says:

"Let the Scala compiler work for you. We provide type classes, instances, conversions, testing, supplements to the standard library, and more"

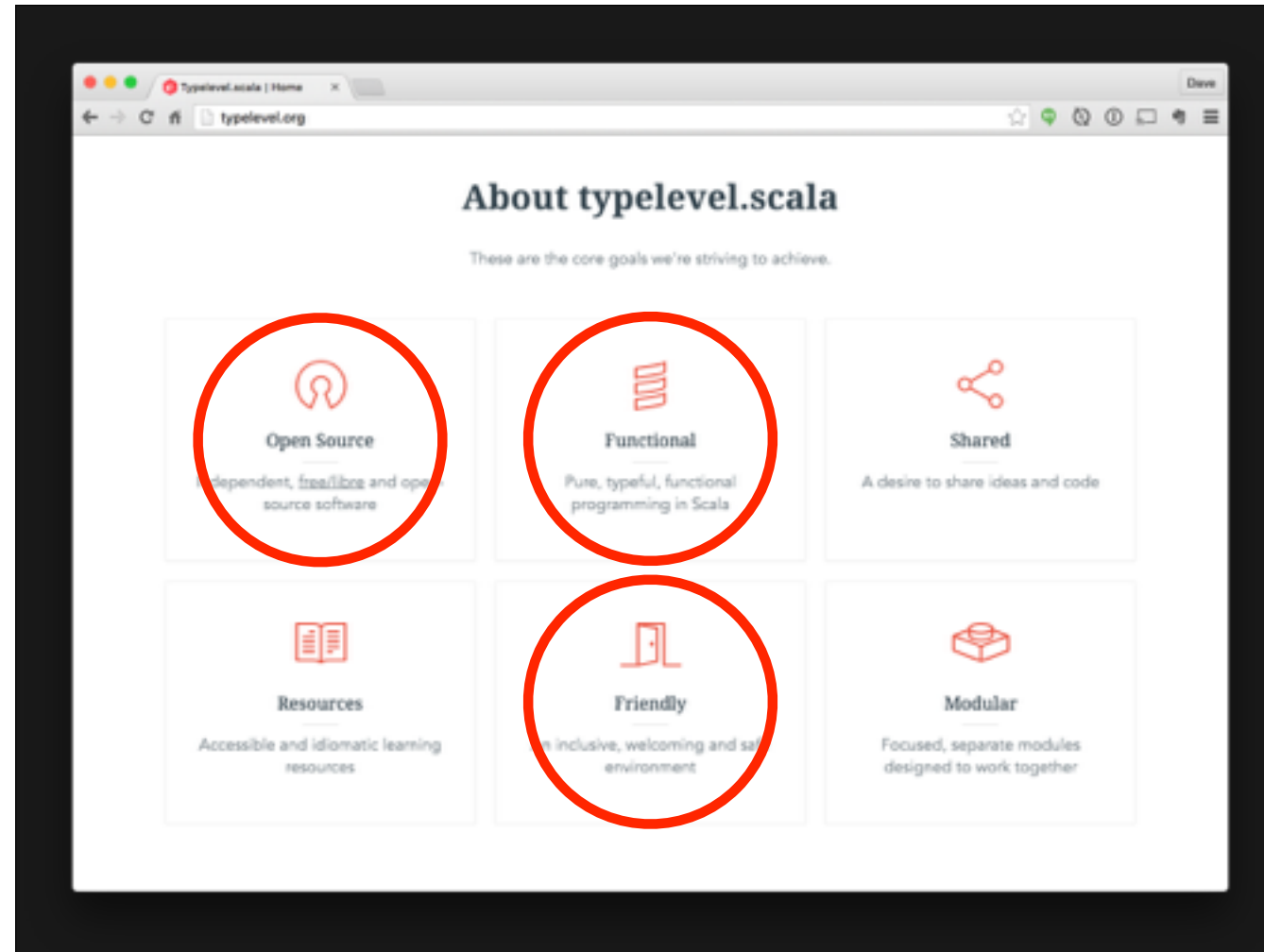
So it says "we build a lot of stuff", but it's very cagey about promising anything more than that.



If we scroll to the bottom of the page, we see the core goals that Typelevel is aiming to achieve.

These goals are reflected by Typelevel libraries as well.

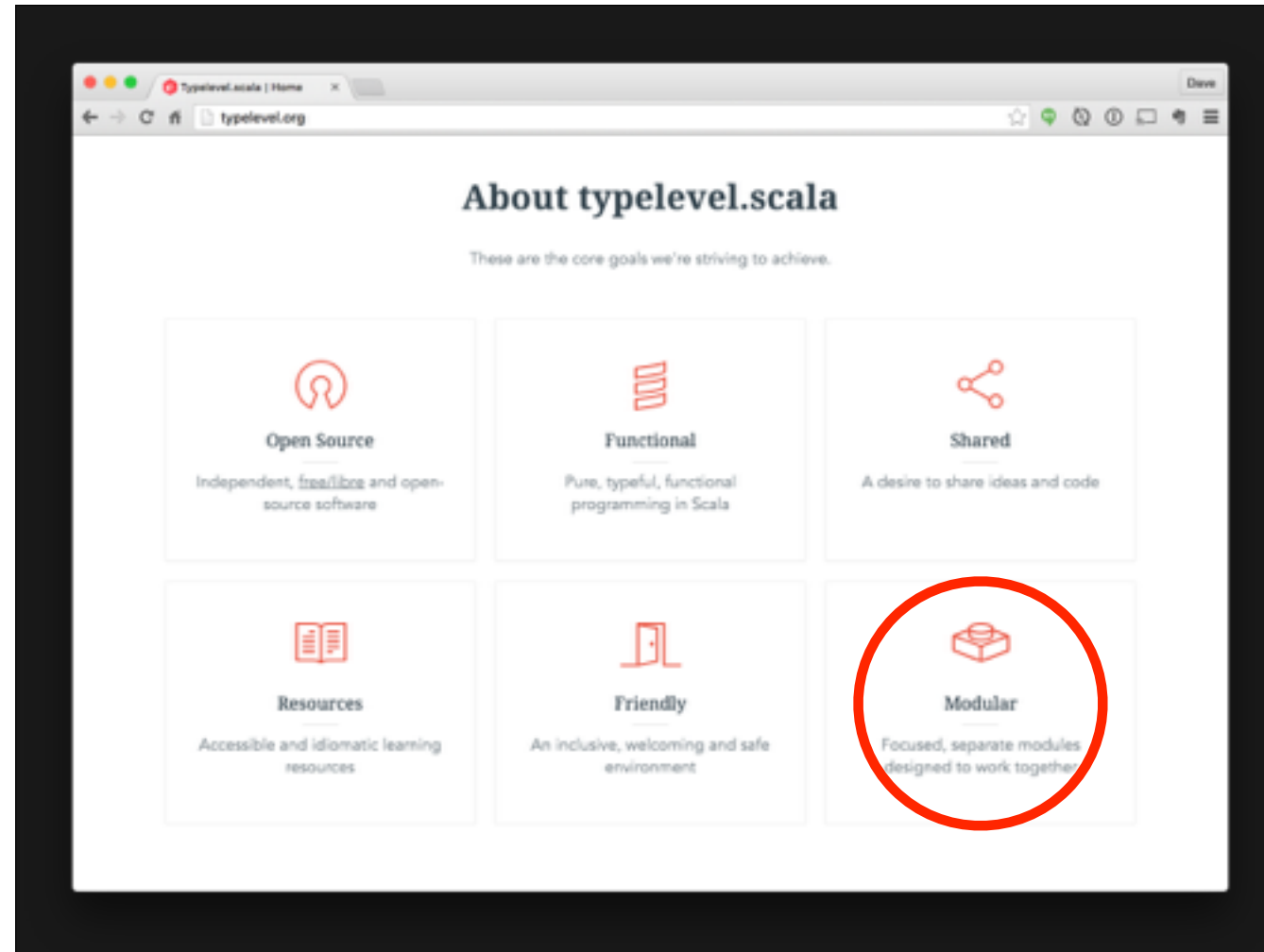




In my interpretation, these are the main goals on the list:

- free/libre open source software;
- pure, typeful functional programming;
- an inclusive, welcoming, and safe environment.

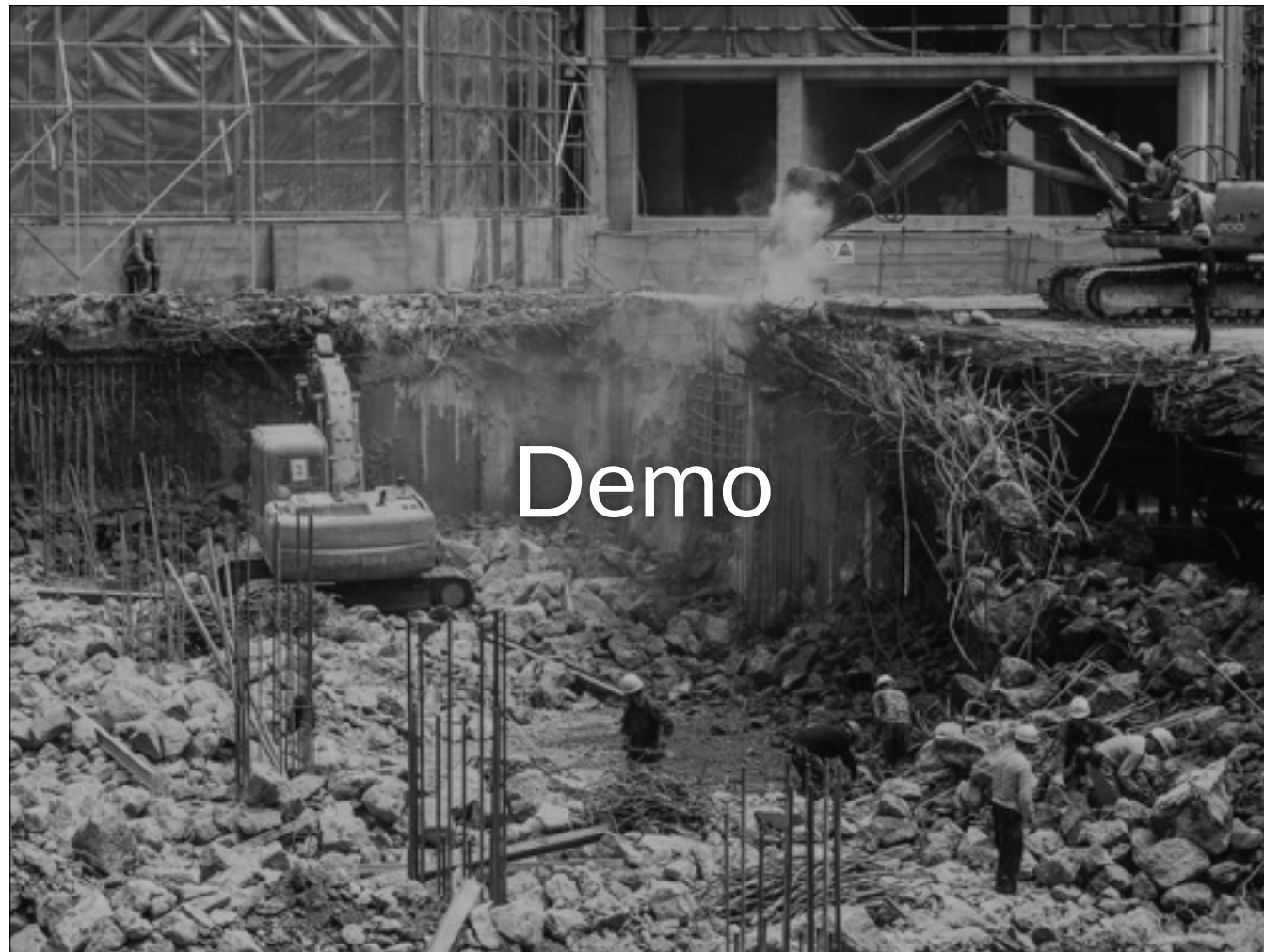
So if you want your library to be part of Typelevel, you broadly have to align with these goals.



However, almost a footnote here on the bottom right, there's a goal to:

"Create focused, separate modules that work together."

So there's at least a goal for interoperability.



So what stack are we going for?

To choose a stack, we have to choose an app to build.

# Typelevel TodoMVC

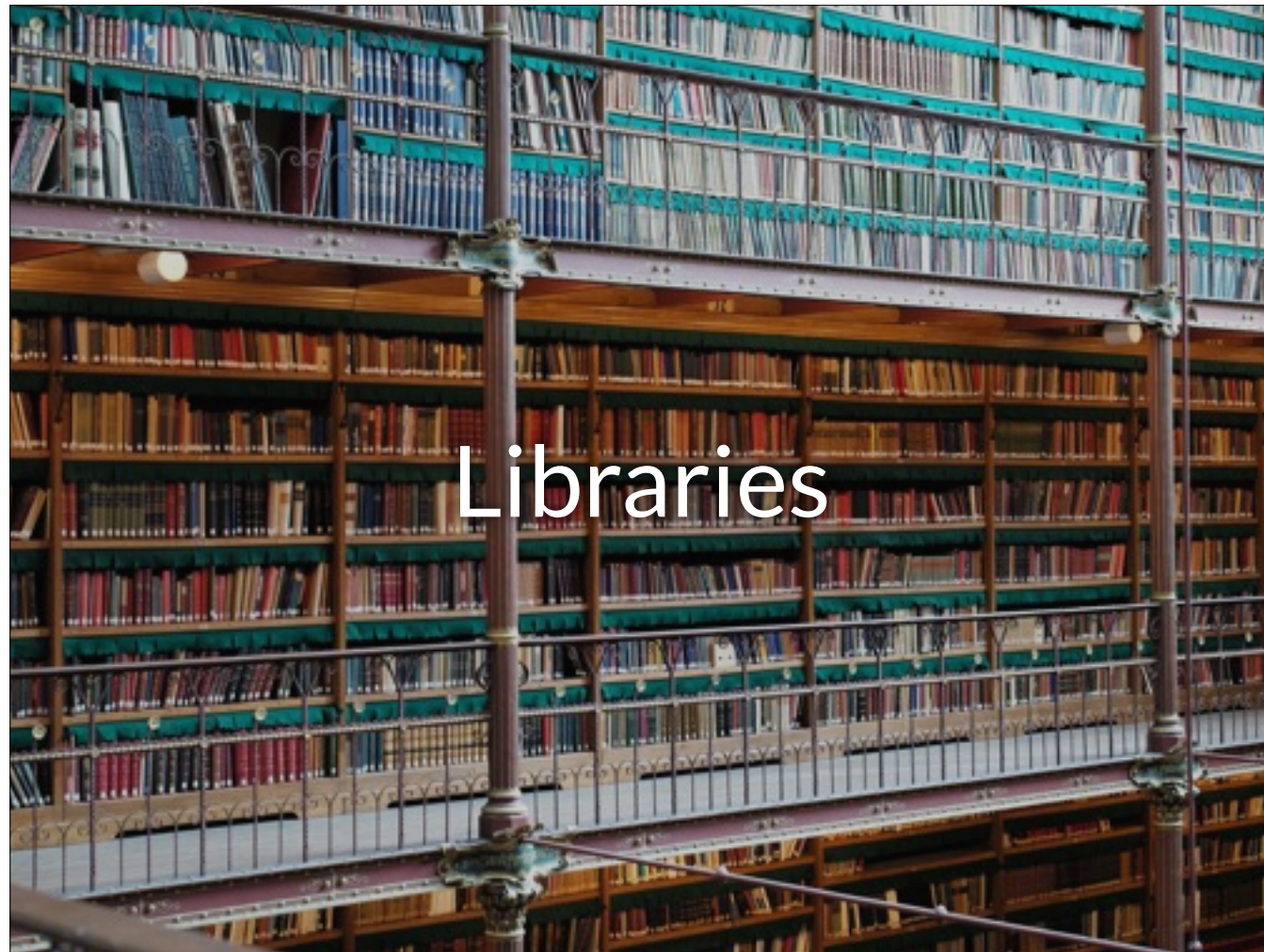
<https://github.com/davegurnell/typelevel-todomvc>



For this talk I implemented a simple todo list app.

I took an existing implementation of TodoMVC, built on top of Diode and ScalaJS React, and added a server component that persists the todo list between page-views.

The ScalaJS app is mostly existing code (except for the ApiClient class), but the server is all built using Typelevel libraries.



So what libraries would we need for this app?

Cats	shapeless	Spire
Algebra	Catalysts	Circe
Discipline	Doobie	Enzyme
Finch	Monocle	Refined
Scala Compiler	ScalaCheck	Scodec
Specs2	Structures	Tut
Export Hook	Imp	Kind Projector
Local Implicits	Machinist	Macro Compat
Simulacrum		

Here's a list of all the libraries on the Typelevel web site...

Cats	shapeless	Spire
Algebra	Catalysts	Circe
Discipline	Doobie	Enzyme
Finch	Monocle	Refined
Scala Compiler	ScalaCheck	Scodec
Specs2	Structures	Tut
Export Hook	Imp	Kind Projector
Local Implicits	Machinist	Macro Compat
Simulacrum		

...there's the core three at the top...

Cats	shapeless	Spire
Algebra	Catalysts	Circe
Discipline	Doobie	Enzyme
Finch	Monocle	Refined
Scala Compiler	ScalaCheck	Scodec
Specs2	Structures	Tut
Export Hook	Imp	Kind Projector
Local Implicits	Machinist	Macro Compat
Simulacrum		

...and a lot of low-level libraries like compiler plugins and compatibility libraries at the bottom.

Some of these libraries are tools to help solve compatibility issues. These are a really interesting area of development... I'll come back to them later.



Cats	shapeless	Spire
Algebra	Catalysts	Circe
Discipline	Doobie	Enzyme
Finch	Monocle	Refined
Scala Compiler	ScalaCheck	Scodec
Specs2	Structures	Tut
Export Hook	Imp	Kind Projector
Local Implicits	Machinist	Macro Compat
Simulacrum		

Finally, in the middle, are the meat. The service level libraries that we'll use to build out our stack.

Cats	shapeless	Spire
Algebra	Catalysts	<b>Circe</b>
Discipline	<b>Doobie</b>	Enzyme
<b>Finch</b>	Monocle	Refined
Scala Compiler	ScalaCheck	Scodec
<b>Specs2</b>	Structures	Tut
Export Hook	Imp	Kind Projector
Local Implicits	Machinist	Macro Compat
Simulacrum		

There's a nice, compact, complete web stack here...

**Finch** - HTTP interface, **Finagle**, **Cats**, **shapeless**, **Circe**

**Circe** - JSON library, **Cats**, **shapeless**

**Doobie** - JDBC interface, **Scalaz**, **shapeless**

**Specs2** - testing framework, **Scalaz**

- Finch for HTTP
- Circe for JSON
- Doobie for database access
- and Specs2 for testing

**Finch** - HTTP interface, **Finagle**, **Cats**, shapeless, Circe

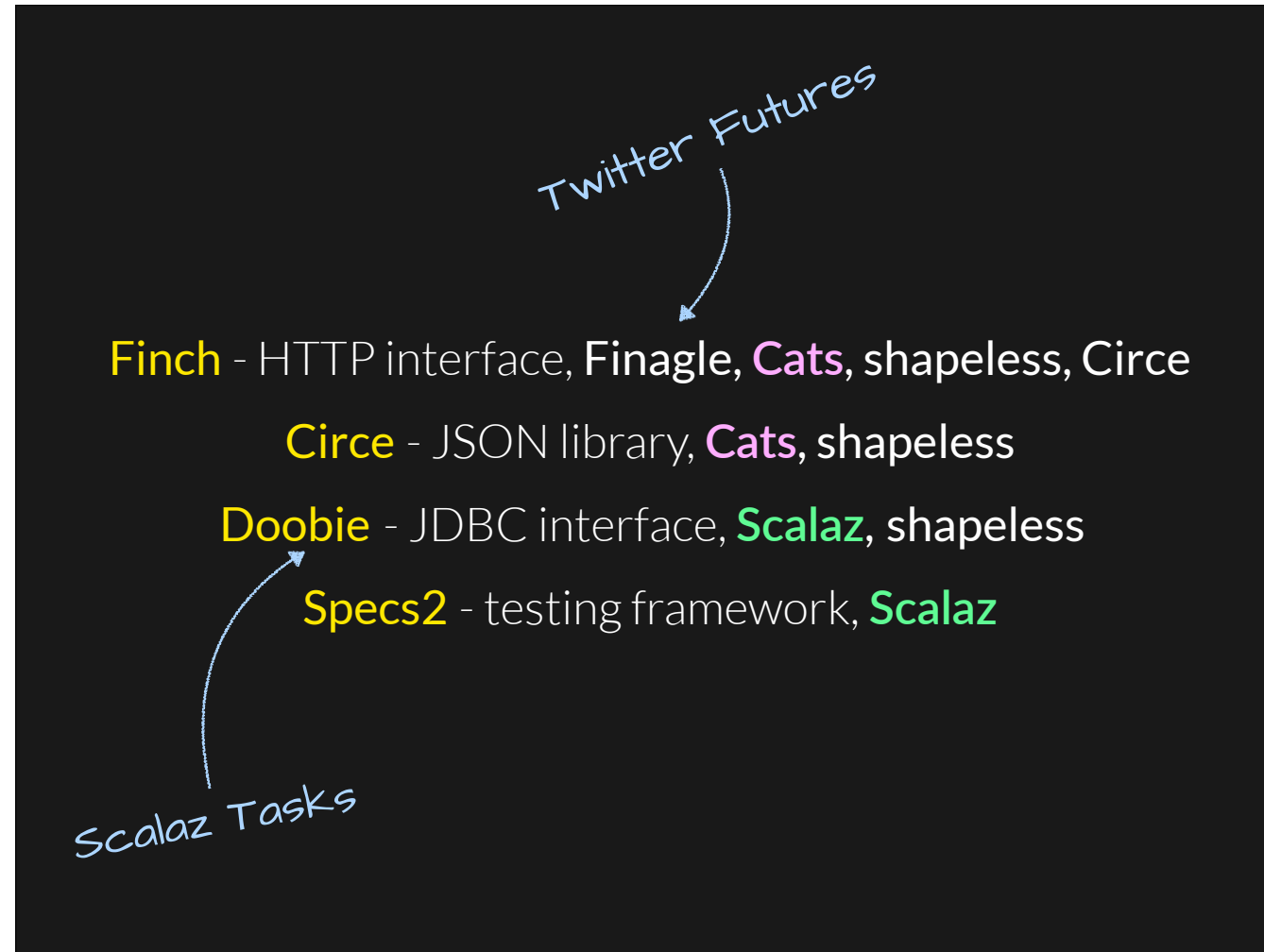
**Circe** - JSON library, **Cats**, shapeless

**Doobie** - JDBC interface, **Scalaz**, shapeless

**Specs2** - testing framework, **Scalaz**

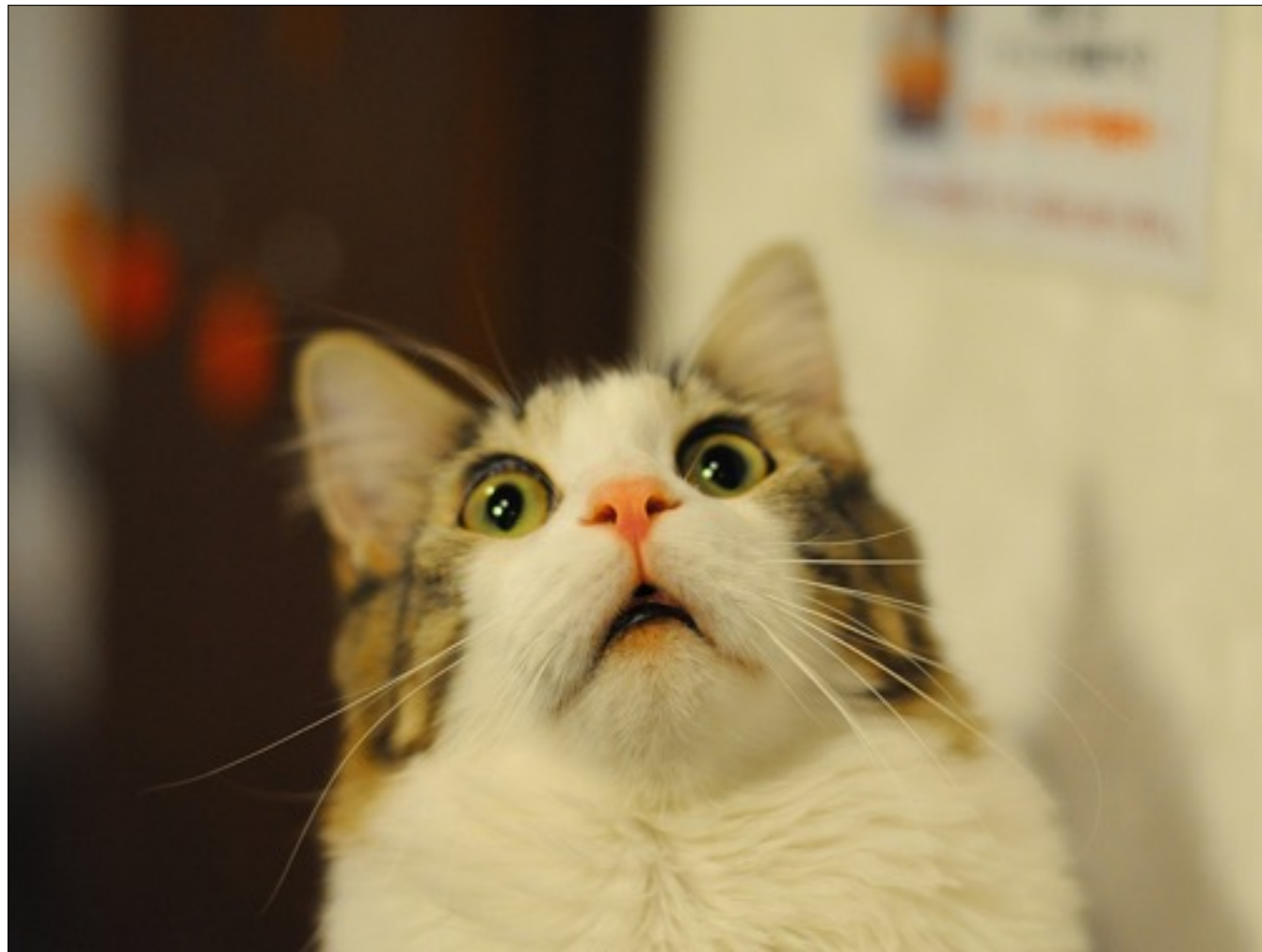
But there are some potential causes for concern. If we look at the dependencies of these libraries, we see that they have very different provenance.

First, Finch and Circe are build on Cats and Doobie and Specs are build on Scalaz.



Second, Finch is built on top of Finagle and Twitter Futures, and Doobie is built on top of scalaz.concurrent.

These are all fine choices in their own right, but if we're starting out on a project, we might rightly be worried about problems getting these things to work together.



Panic!

If we delve into the details, it's not actually that bad.

Let's look at the libraries to get a feel for the situation.



## Your REST API as a Monad

Vladimir Kostyukov

[https://skillsmatter.com/skillscasts/  
6876-finch-your-rest-api-as-a-monad](https://skillsmatter.com/skillscasts/6876-finch-your-rest-api-as-a-monad)

This is a great talk if you're looking to get into Finch.

In it, Vladimir describes Finch's model in detail and live-codes a simple API.



```
import io.finch._

val readTodo = get("todos" / long) { (id: Long) =>
  val todo: Future[Todo] = findTodo(id)
  todo.map(Ok(_))
}

Http.server.serve(":8080", readTodo.toService)
```

Here's the code for a "read todo" endpoint to show the basic layout.

```
import io.finch._

val readTodo = get("todos" / long) { (id: Long) =>
  val todo: Future[Todo] = findTodo(id)
  todo.map(Ok(_))
}

Http.server.serve(":8080", readTodo.toService)
```

Finch provides a bunch of predefined combinators that produce “Endpoints”.

An Endpoint[A] is a function that accepts a request and returns a value of type A.

Here you can see a simple DSL defining an endpoint that:

- checks that the request has a GET method and a URL of the form “/todos/<number>”
- extracts that number and returns it as a Long.

The endpoint here forms the input to our application logic...

```
import io.finch._

val readTodo = get("todos" / long) { (id: Long) =>
  val todo: Future[Todo] = findTodo(id)
  todo.map(Ok(_))
}

Http.server.serve(":8080", readTodo.toService)
```

...which we bolt on by mapping a function over the endpoint.

(Note: The method used here is called “apply” for aesthetic reasons, but it has “map” semantics.)

```
import io.finch._

val readTodo = get("todos" / long) { (id: Long) =>
  val todo: Future[Todo] = findTodo(id)
  todo.map(Ok(_))
}

Http.server.serve(":8080", readTodo.toService)
```

The finish our application logic by returning an "Ok" of a "Todo". So it's a wrapper object representing a response, but the content is a semantic type belonging to our application domain.

```
import io.finch._

val readTodo = get("todos" / long) { (id: Long) =>
  val todo: Future[Todo] = findTodo(id)
  todo.map(Ok(_))
}

Http.server.serve(":8080", readTodo.toService)
```

The final step in our codebase is to use this “toService” method to turn our endpoint into a Finagle service.

Finch uses a bunch of implicits to work out how to turn the Todo in our Ok response to something we can actually send to the client.

In the TodoMVC app we’re using JSON. I’ve missed off the imports and elided some small details about how we generate the JSON here. I’ll talk about that next...

# Finch + Circe



Circe is a fork of Argonaut that concentrates heavily on shapeless to generate Decoders and Encoders with no boilerplate.

```
import io.finch._
import io.finch.circe._
import io.circe.generic.auto._

val saveTodo: Endpoint[Todo] =
  post("todos" :: body.as[Todo]) { (posted: Todo) =>
    val saved: Future[Todo] = save(posted)
    todo.map(Ok(_))
  }
```

The best libraries are the ones you don't have to write code for!

There almost zero lines of Circe code in this API.

```
import io.finch._
import io.finch.circe._
import io.circe.generic.auto._

val saveTodo: Endpoint[Todo] =
  post("todos" :: body.as[Todo]) { (posted: Todo) =>
    val saved: Future[Todo] = save(posted)
    todo.map(Ok(_))
  }
```

The highlighted section here is an endpoint that extracts the request body as a Todo.

Finch does this by summoning a type class instance for a request body parser.



```
import io.finch._  
import io.finch.circe._  
import io.circe.generic.auto._  
  
val saveTodo: Endpoint[Todo] =  
  post("todos" :: body.as[Todo]) { (posted: Todo) =>  
    val saved: Future[Todo] = save(posted)  
    todo.map(Ok(_))  
  }
```

Type types of request we can handle depend on the body parsers we have in scope.

The highlighted package is a Finch/Circe module, which gives us a body parser provided we have a JSON decoder in scope.

```
import io.finch._
import io.finch.circe._
import io.circe.generic.auto._

val saveTodo: Endpoint[Todo] =
  post("todos" :: body.as[Todo]) { (posted: Todo) =>
    val saved: Future[Todo] = save(posted)
    todo.map(Ok(_))
  }
```

And this highlighted package gives us Circe's shapeless-powered automatic decoder generation.

So we don't need to declare the JSON decoder. Circe generates it for us. And Finch generates the body parser. And we're good.

```
import io.finch._
import io.finch.circe._
import io.circe.generic.auto._

val saveTodo: Endpoint[Todo] =
  post("todos" :: body.as[Todo]) { (posted: Todo) =>
    val saved: Future[Todo] = save(posted)
    todo.map(ok(_))
  }
```

And it's the same imports that allow us to serialize a Todo to JSON at the end of our application logic. This is the detail I elided earlier.

And before we move on, note that the async primitive we're using here is a Twitter Future. This will be important in a bit.

# Circe + ScalaJS



One thing that's really awesome about Circe is that it works in ScalaJS as well as Scala.

```
// imports ...

def syncTodos(todos: Seq[Todo]): Future[List[Todo]] =
  Ajax.put(s"${apiRoot}/todo", todos.toJson.noSpaces)
    .map(decodeTodoList)

def decodeTodoList(xhr: XMLHttpRequest): List[Todo] =
  decode[List[Todo]](xhr.responseText)
    .getOrElse(sys.error("Could not decode JSON"))
```

Here's an example method from our API client, that syncs a todo list to the server.

```
// imports ...

def syncTodos(todos: Seq[Todo]): Future[List[Todo]] =
  Ajax.put(s"${apiRoot}/todo", todos.toJson.noSpaces)
    .map(decodeTodoList)

def decodeTodoList(xhr: XMLHttpRequest): List[Todo] =
  decode[List[Todo]](xhr.responseText)
    .getOrElse(sys.error("Could not decode JSON"))
```

The important parts here are the only code we need to:

- serialise a List[Todo] to JSON to pass it to the server;
- deserialise the List[Todo] that we get back.



That's all for Finch and Circe. On to Doobie.

## Programs as Values: JDBC Programming with Doobie

Rob Norris

<https://www.youtube.com/watch?v=M5MF6M7FHPo>

If you want to learn more about Doobie, watch this excellent talk by Rob Norris in which he:

- shows you how to use Doobie;
- shows you how Doobie is made;
- throws teaching you about the Free monad into the bargain.

Doobie also has an excellent manual in the Book of Doobie.



```
import doobie.imports._

def readTodo(id: Long): ConnectionIO[Option[Todo]] =
  sql"""select * from todos where id = $id"""
    .query[Todo]
    .option

val result: Task[Option[Todo]] =
  readTodo(123).transact(xa)
```

Here's some Doobie code.

```
import doobie.imports._

def readTodo(id: Long): ConnectionIO[Option[Todo]] =
  sql"""select * from todos where id = $id"""
    .query[Todo]
    .option

val result: Task[Option[Todo]] =
  readTodo(123).transact(xa)
```

This interpolated string builds a prepared statement and provides the ID parameter in an injection-safe manner.

```
import doobie.imports._

def readTodo(id: Long): ConnectionIO[Option[Todo]] =
  sql"""select * from todos where id = $id"""
    .query[Todo]
    .option

val result: Task[Option[Todo]] =
  readTodo(123).transact(xa)
```

We call “.query” to specify the result type for this query.

Doobie is using shapeless magic here to inspect the fields on Todo, inspect the columns returned by the query, and align them to extract Todos.

```
import doobie.imports._

def readTodo(id: Long): ConnectionIO[Option[Todo]] =
  sql"""select * from todos where id = $id"""
    .query[Todo]
    .option

val result: Task[Option[Todo]] =
  readTodo(123).transact(xa)
```

Then we use this “.option” combinator to turn our query into a ConnectionIO object.

ConnectionIO is like an IO monad for database queries. It represents one or more queries with Scala code to interpret the results. So we can use “flatMap”, and “|@|” to combine ConnectionIOs in sequence and parallel to represent entire query plans.

```
import doobie.imports._

def readTodo(id: Long): ConnectionIO[Option[Todo]] =
  sql"""select * from todos where id = $id"""
    .query[Todo]
    .option

val result: Task[Option[Todo]] =
  readTodo(123).transact(xa)
```

Then we call “transact” and pass it a transactor object, which is effectively the same as a database.

This turns the ConnectionIO query plan into a Scalaz Task, which we can run. When we run it, Doobie takes care of allocating connections, establishing transactions, and parsing the result set.

```
import doobie.imports._

def readTodo(id: Long): ConnectionIO[Option[Todo]] =
  sql"""select * from todos where id = $id"""
    .query[Todo]
    .option

val result: Task[Option[Todo]] =
  readTodo(123).transact(xa)
```

And from the point of view of this talk, the key point here is that Doobie gives us back a Scalaz Task.



# Tasks vs Futures

The one incompatibility we've found so far is the different async primitives being used by Doobie and Finch.

## Scalaz Task: the Missing Documentation

Tim Perrett

<http://timperrett.com/2014/07/20/scalaz-task-the-missing-documentation/>

Twitter Futures are very similar to Scala Futures, but Scalaz Tasks are very different beasts.

If, like me, you're new to Scalaz Tasks, here's a great blog post by Tim Perrett that dives into the details.



```
import com.twitter.util.{Future, Promise}
import scalaz.\
import scalaz.concurrent.Task

implicit class TaskOps[A](task: Task[A]) {
  def runAsTwitterFuture: Future[A] = {
    val promise = Promise[A]
    task.runAsyncInterruptibly { dis: Throwable \/ A =>
      dis.fold(
        exn => promise.setException(exn),
        ans => promise.setValue(ans)
      )
    }
    promise
  }
}
```

I managed to hack together the code above from some examples in the Finch documentation and some examples on Stack Overflow.

```
import com.twitter.util.{Future, Promise}
import scalaz.\
import scalaz.concurrent.Task

implicit class TaskOps[A](task: Task[A]) {
  def runAsTwitterFuture: Future[A] = {
    val promise = Promise[A]
    task.runAsyncInterruptibly { dis: Throwable \/ A =>
      dis.fold(
        exn => promise.setException(exn),
        ans => promise.setValue(ans)
      )
    }
    promise
  }
}
```

It provides this method, “runAsTwitterFuture”, that runs a Task and captures the result in a Twitter Future.

```
import io.finch._

val readTodo = get("todos" / long) { (id: Long) =>
  val todo = findTodo(id).runAsTwitterFuture
  todo.map(Ok(_))
}
```

Here's an example of the usage.

# Is this a problem?

The big question is: is this a problem?

In many ways, this is fine. It's not complex code, particularly at the use site. However, there are two issues:

- Beginning developers might have trouble defining `runAsTwitterFuture`;
- It's not clear what the runtime behaviour is going to be. What happens when we tie two async execution libraries together like this? Who knows?

Should this be  
documented?


Another question is: should this be documented somewhere? And somewhere other than SO?

# Where should this be documented?

And if so, where?

Is it Doobie's job to document interop with Twitter Futures? Is it Finch's job to document interop with Scalaz Tasks? Probably not.

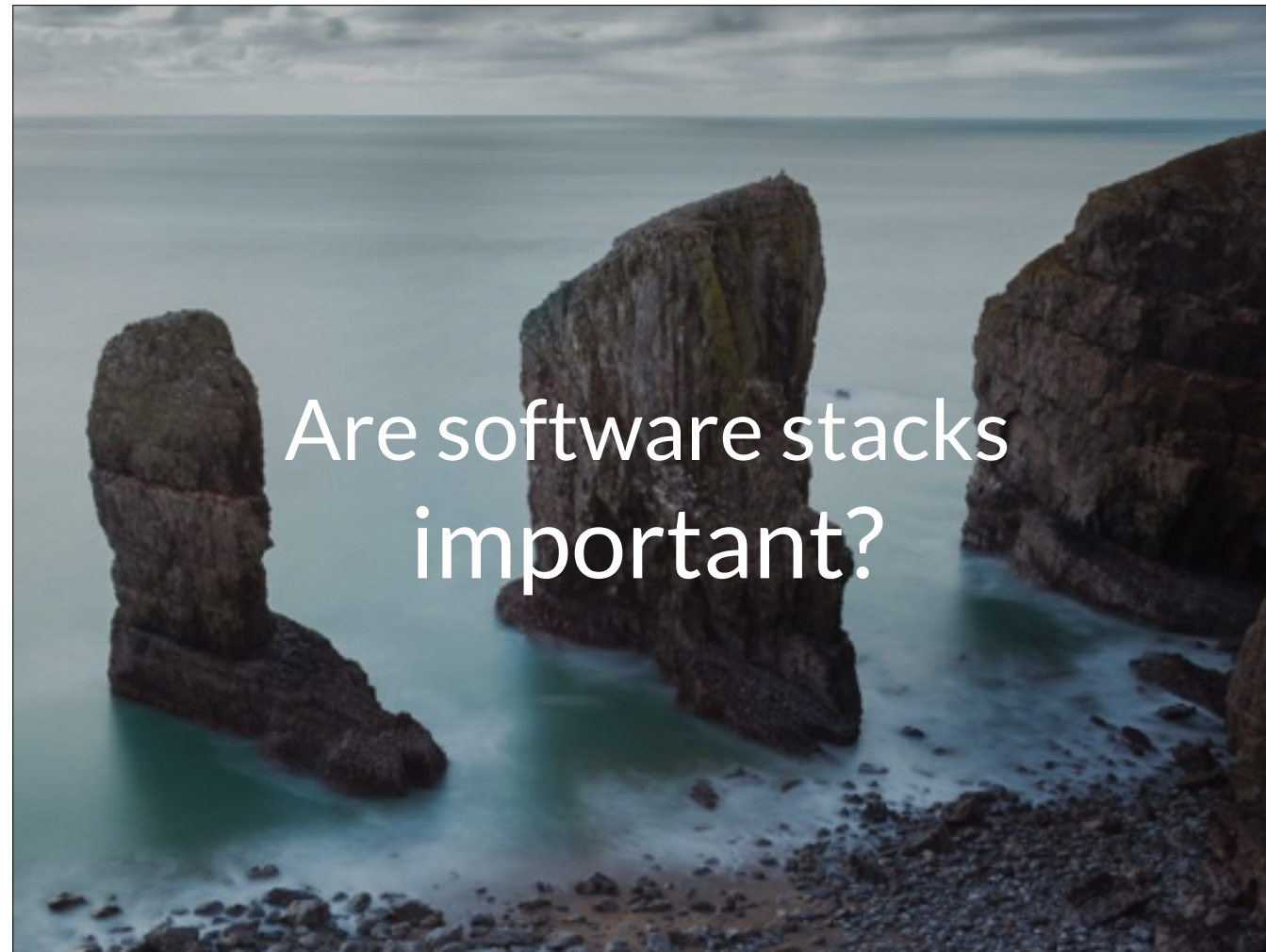
Is there a call for a higher level of documentation here, that documents the libraries working together in one stack?



Should it be  
a library?

Should this, maybe, part of some async-combat library? With code to do the conversions and documentation on best practices?

It's an open question.



I want to finish by talking about the importance of stacks.

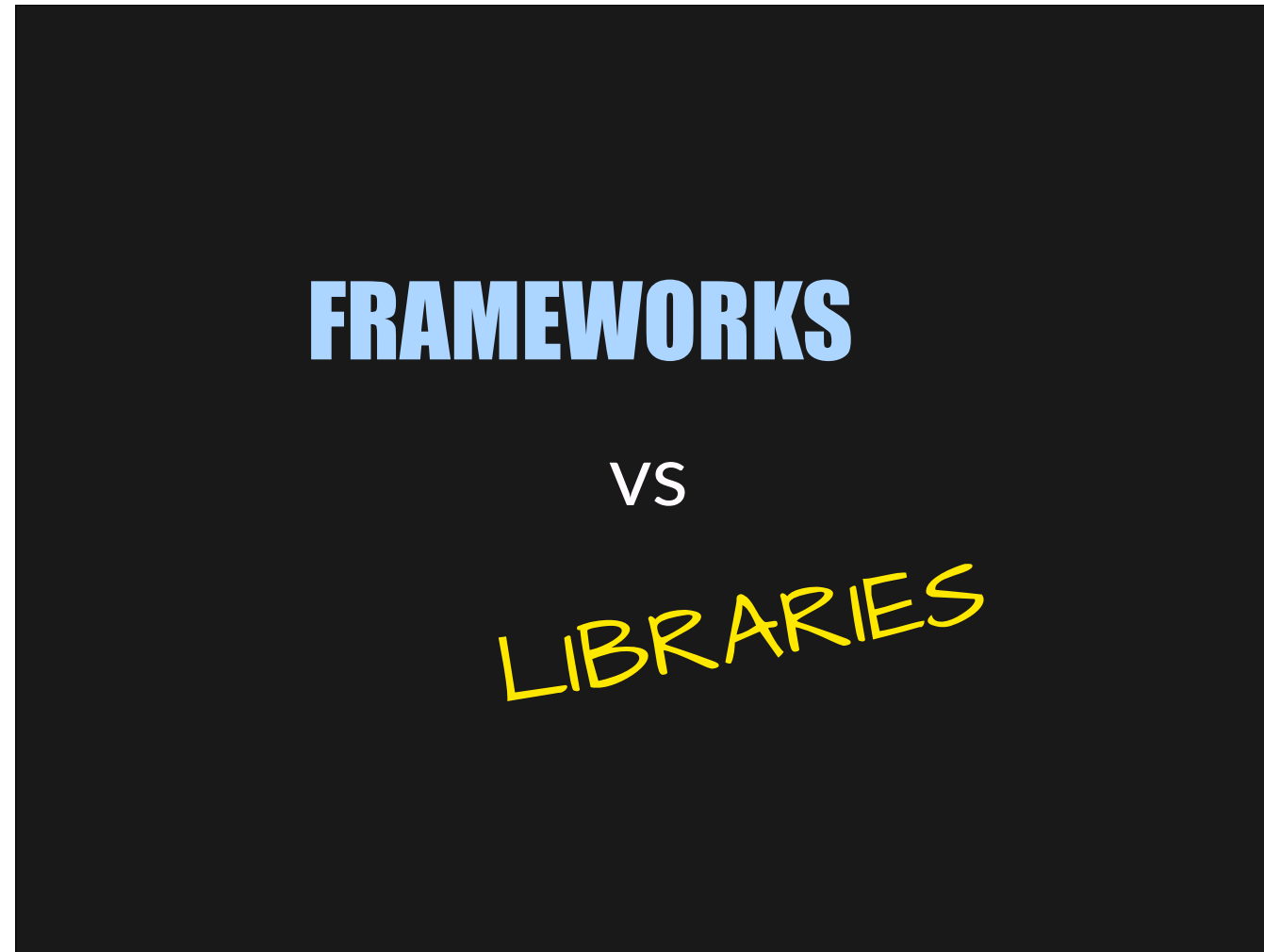
This falls into a couple of categories.



# Importance to developers

First, importance to developers.

Are stacks important to developers? Of course! We want to pick up libraries and have them work seamlessly together!



We've all heard the old frameworks-versus-libraries battle.

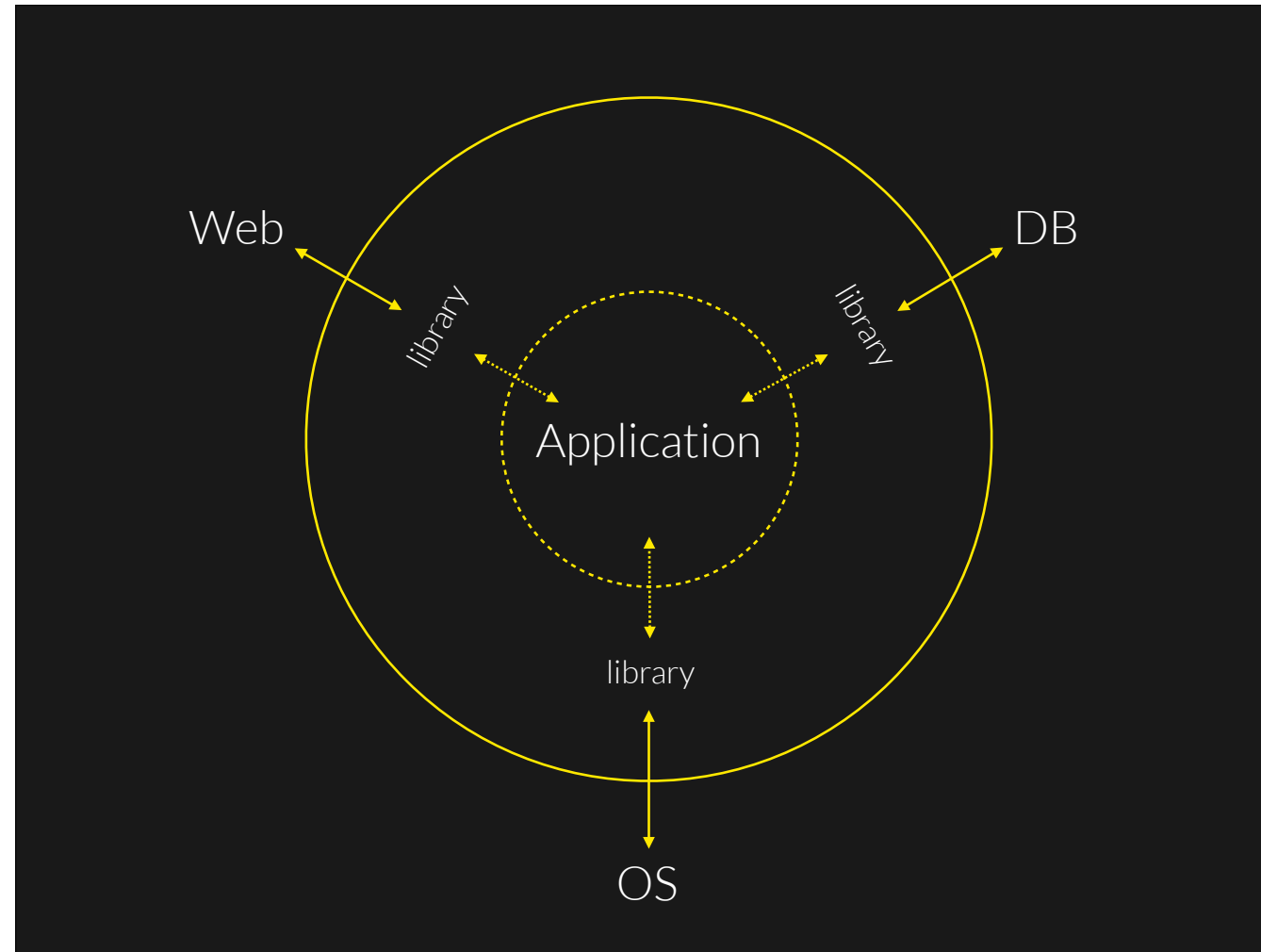
Frameworks give us the promise of lots of sub-libraries that all work well together. But when we add monolithic libraries to our stacks, they come with problems:

- it's more work to upgrade to new versions;
- it's more painful when development on the framework slows;
- it's higher risk (more eggs in one basket).

Libraries give us more choice, but they introduce more complexity and more opportunity for conflicts.

# Importance to library maintainers

Second, let's look at the importance of stacks to library maintainers.



Our libraries are living in ecosystems, and stacks are going to form around them. We can react to this and promote documentation compatibility layers for the most common cases.

One of the advantages of pure functional programming is that we can minimise the visible surface area of our libraries, pushing areas of uncertainty to the edges.

Think of the IO monad, for example, or Doobie's ConnectionIO monad.

```
connectionIO.transact(xa)
```

In Doobie, there's this fantastic pivot point when we call `transact`.

Everything we do up until calling `transact` is 100% Doobie. Everything afterwards is 100% Scalaz. Similarly with Finch, there are only a couple of points of interaction between our application code and the library.

Library developers can exploit this. If there are limited points of interaction, we can anticipate teething problems with documentation, best practices, and compatibility libraries to bridge gaps.

# Importance to Typelevel

Finally, let's consider how stacks are important to Typelevel.

And when I say Typelevel, I'm not talking about Miles or Lars or Erik any particular core people. I'm talking about all of us, as community members.

Cats	shapeless	Spire
Algebra	Catalysts	Circe
Discipline	Doobie	Enzyme
Finch	Monocle	Refined
Scala Compiler	ScalaCheck	Scodec
Specs2	Structures	Tut
Export Hook	Imp	Kind Projector
Local Implicits	Machinist	Macro Compat
Simulacrum		

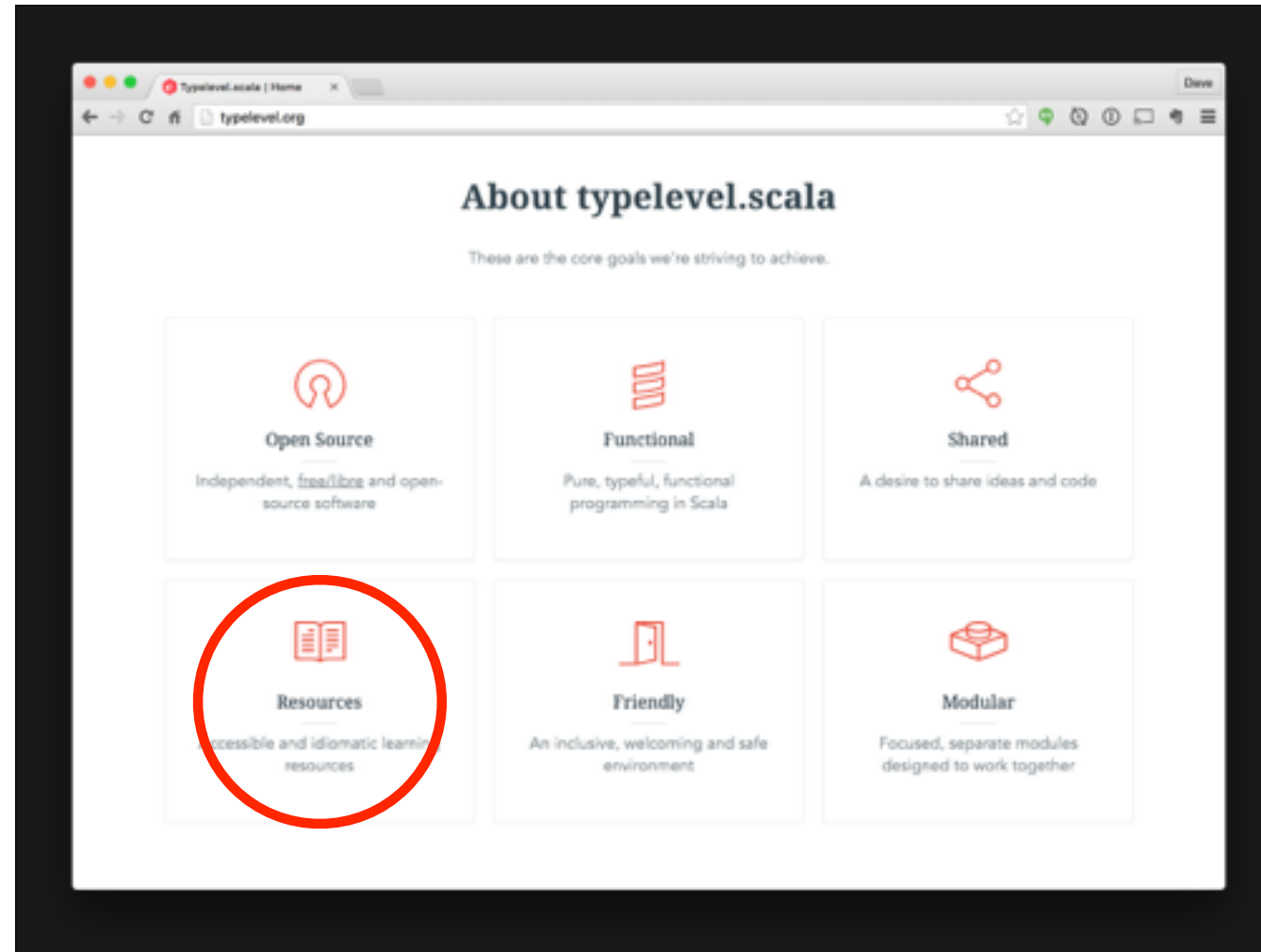
We've arrived at this fantastic point now where we have lots of great libraries in the Typelevel pantheon.

In the absence of frameworks that dictate how every piece fits together, we need to find ways to make these libraries fit together.

Cats	shapeless	Spire
Algebra	Catalysts	Circe
Discipline	Doobie	Enzyme
Finch	Monocle	Refined
Scala Compiler	ScalaCheck	Scodec
Specs2	Structures	Tut
Export Hook	Imp	Kind Projector
Local Implicits	Machinist	Macro Compat
Simulacrum		

One option is to introduce more compatibility libraries. Help bridge the gap between libraries that are commonly used together, taking the burden off the library maintainers themselves.





Another key area is in documentation

One of the key goals I skipped over was to “create accessible and idiomatic learning resources” for Typelevel libraries.

We’ve made great strides in this direction. The Cats documentation, for example, is head over heels better than the documentation for Scalaz before it.

But we need new types of documentation to help with higher goals. For example...

How do I use a  
**monoid?**

I can go to the Cats documentation and find out how to use a monoid.

# What does a monoid do?

I can even find out what a monoid is (it's a "semigroup with a zero") and what it does.

# How do I write a web app?

But these aren't the questions that new developers are asking.

We need to move away from the conceptual and the mechanical. We need to answer concrete goal-focused questions like "How do I write this kind of app?"

We need to make suggestions about library choices, and help developers use these libraries together.

# What is the Typelevel Stack?

So we started by asking “What is the Typelevel Stack?”

I hope I’ve convinced you here of two things...

# What are the Typelevel Stacks?

First, there's no one Typelevel Stack. There are many. We've got lots of libraries under our belt, and the combinations are as diverse as the types of application we can conceive of writing.

So now we need to focus on how we can help developers use our libraries together. Through blog posts, documentation, compatibility libraries, Gitter chat, and all of the tools at our disposal.

# Thanks to...

Ferdinand Svehla

Kingsley Davies

Rob Norris

Travis Brown

Vladimir Kostyukov

I'd like to thank these people who were enormously helpful in the writing of this talk.

# Thanks for Listening

## Any questions?

Dave Gurnell, Underscore  
[@davegurnell](https://twitter.com/davegurnell)

And I'd like to thank you for listening (or reading these slides).



Stacks on Stacks by Darwin Bell, CC-BY-NC-2.0  
<https://flic.kr/p/55h26m>

Stacked Stones by Dave Warley, CC-BY-2.0  
<https://flic.kr/p/pWxUZu>

The Stacks by Roman Boed, CC-BY-2.0  
<https://flic.kr/p/mmgwkx>

Scared Cat Is Really Scared by dat', CC-BY-ND-2.0  
<https://flic.kr/p/7WmJPX>

Multnomah Whiskey Library by Roger, CC-BY-NC-ND-2.0  
<https://flic.kr/p/j3fACu>

Gold Finch and House Finch by John Flannery, CC-BY-SA-2.0  
<https://flic.kr/p/xMDTgY>

Circe by Smithsonian Art Museum, CC-BY-NC-ND-2.0  
<https://flic.kr/p/6odp4B>

Doobie Brothers by David Gans, CC-BY-2.0  
<https://flic.kr/p/4VD2a>

No "Future" for Future Shop by Jamie McCaffrey, CC-BY-NC-2.0  
<https://flic.kr/p/qU2oiV>

Elegug Stacks by Alex South, CC-BY-2.0  
<https://flic.kr/p/pVZUQM>

All images sourced from Flickr and licensed Creative Commons. Thanks to their authors.