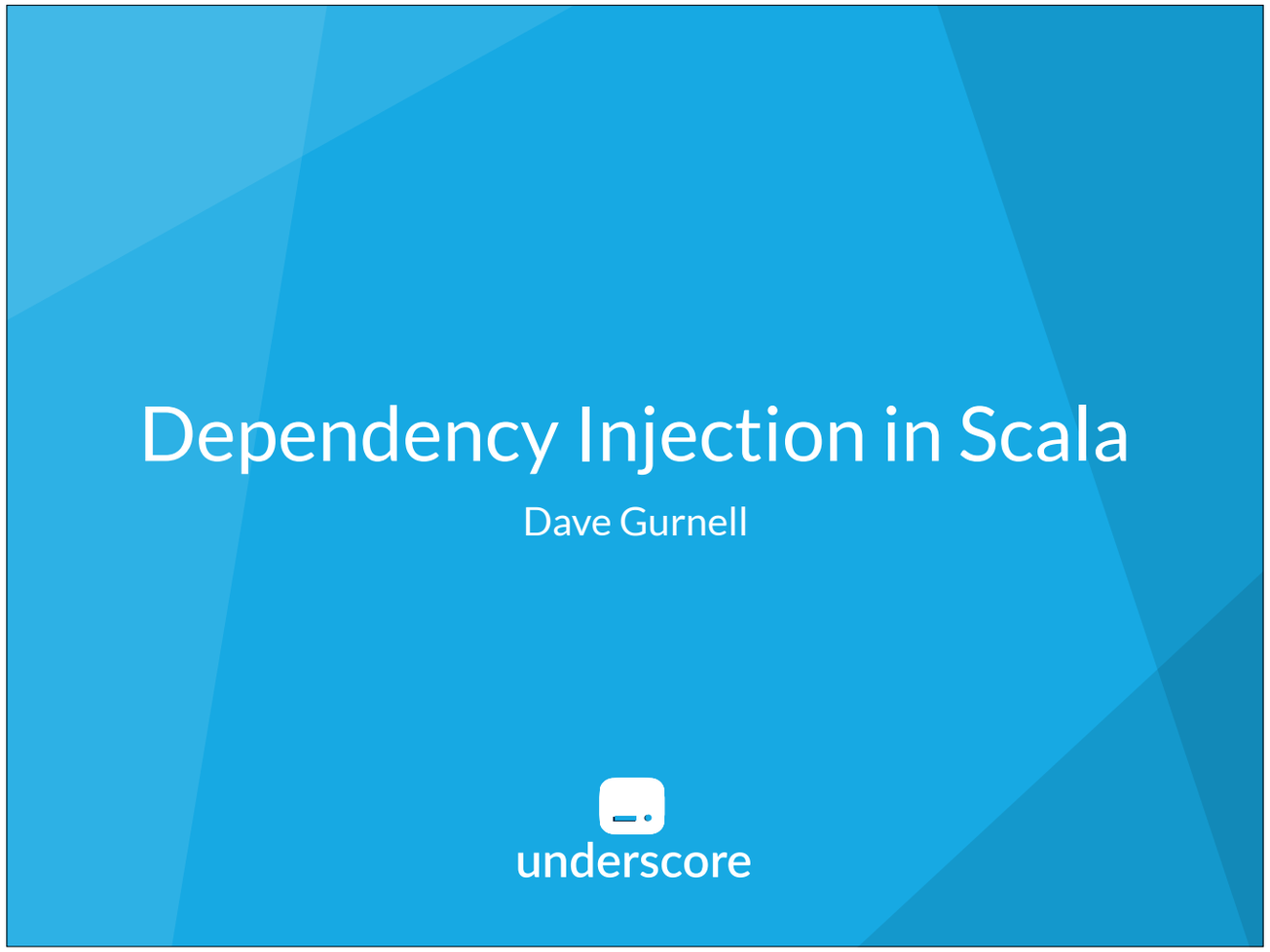
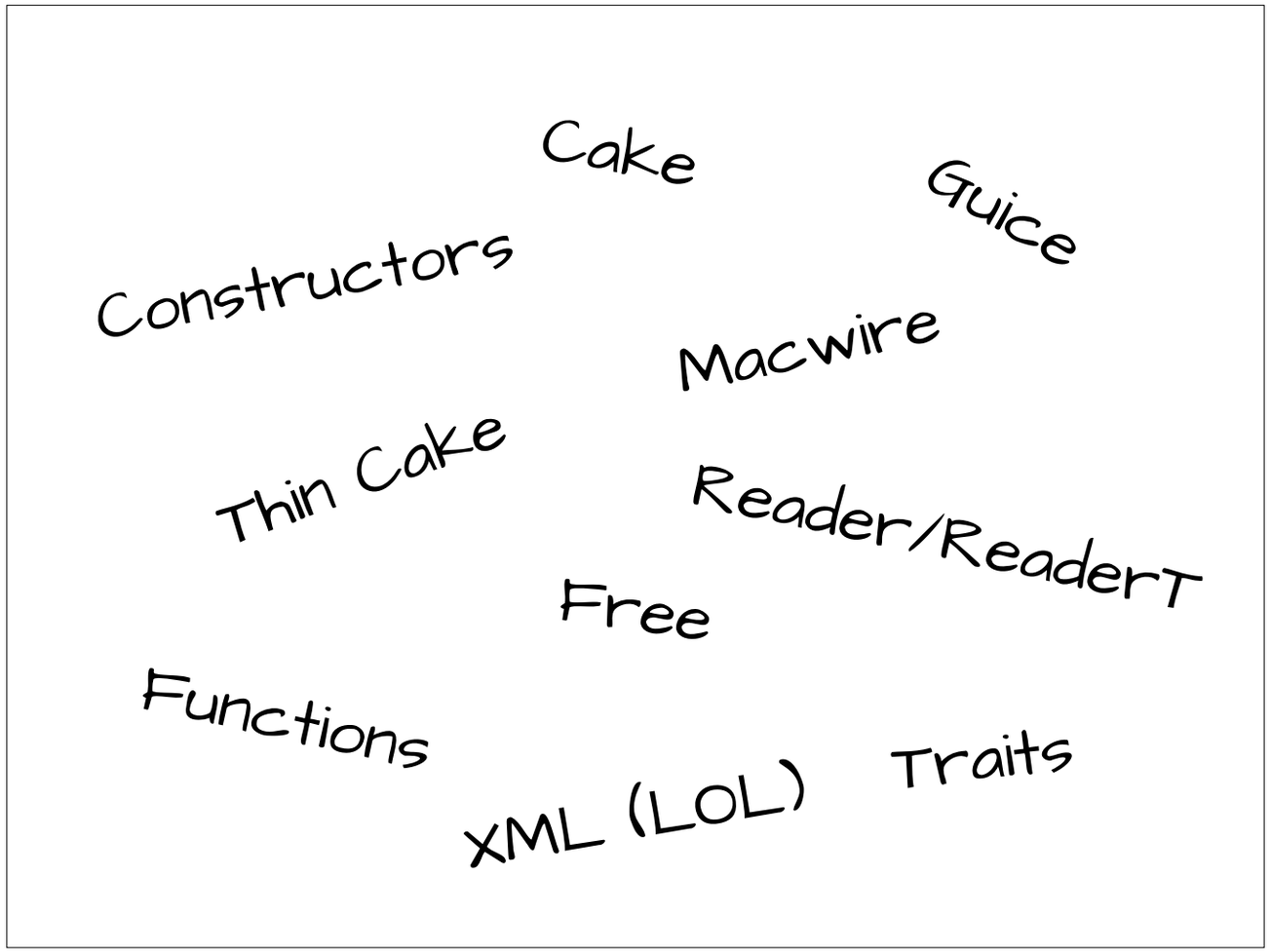


This talk is about dependency injection in Scala.

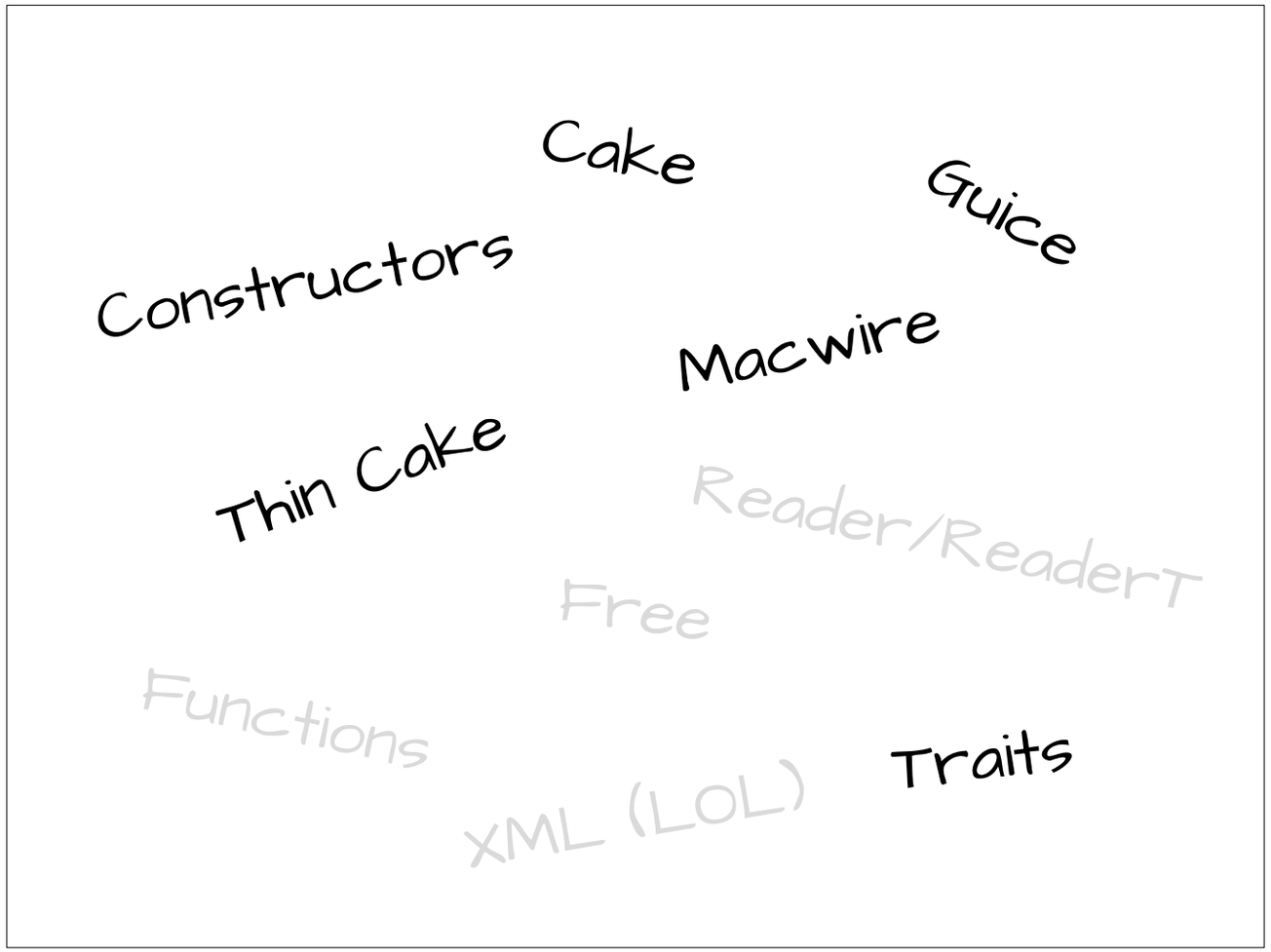
AKA

99 Ways To DI

I originally had a cool joke in here about Megadeth.
Watch the video from Scala Central for more :)



There are a lot of ways to do DI in Scala.



We only have time to cover some of them in this talk, so I'll concentrate on these.

What is DI?

We’ll start by defining our goals for DI.

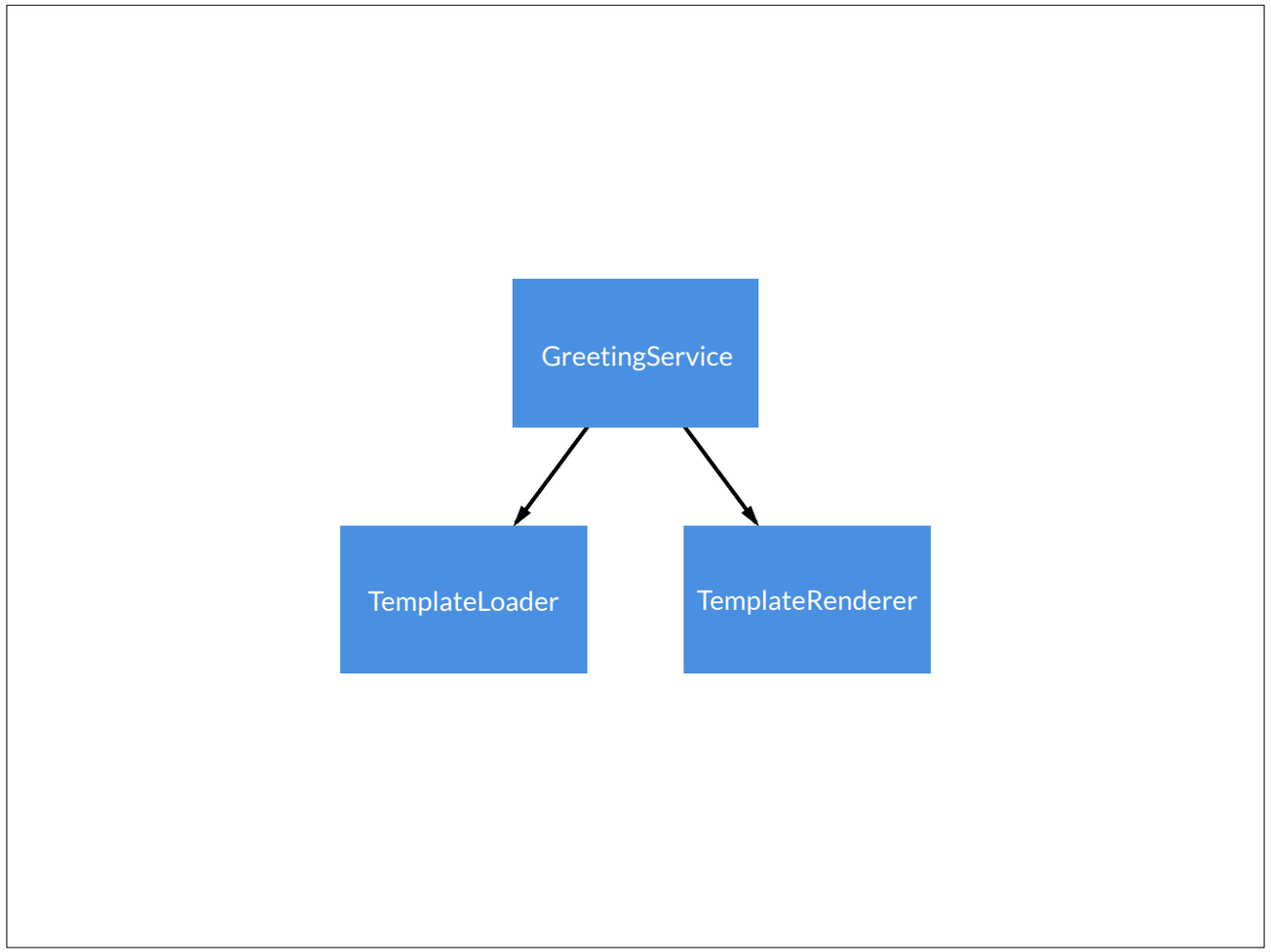
#1
Build app from components

#2
Swap parts out for testing

#3
Do it all at compile time!

Give me an example!

Now we'll define a really simple example that we can reimplement using different techniques.



It’s a greeting card service.

“GreetingService” has a “greet()” method to generate a greeting.

It has two dependencies: one for loading a greeting card templates,
and one for rendering a template with parameters such as name and welcome message.

```scala
object TemplateLoader {
  def apply(id: TemplateId): Template =
    ???
}

object TemplateRenderer {
  def apply(id: Template, params: Params): String =
    ???
}

object GreetingService {
  val loader = TemplateLoader
  val renderer = TemplateRenderer

  def greet(id: TemplateId, params: Params): String =
    renderer(loader(id), params)
}
```

Here’s an example implementation with hard-coded dependencies.

```
"greeting service" should {
  "render a greeting" in {
    // Test body:
    val params   = Params(Map("name" -> "Dave"))
    val actual   = GreetingService.greet(Greeting, params)
    val expected = "Test greeting for Dave"

    // Postconditions:
    actual should be(expected)
  }
}
```

Here’s a test for GreetingService.

It sets up some parameters, calls greet(), and checks the output.

The trouble is it relies on TemplateLoader and TemplateRenderer.
TemplateLoader is pulling templates from wherever we store them in production
(let’s say Amazon S3).

This means our unit tests depend on Amazon S3!
They’ll be slow and potentially brittle.

```
trait TemplateLoader {
  def apply(id: TemplateId): Template
}

class S3TemplateLoader extends TemplateLoader {
  def apply(id: TemplateId): Template =
    ???
}

class FakeTemplateLoader(template: Template)
    extends TemplateLoader {
  def apply(id: TemplateId): Template =
    template
}
```

Let’s create different implementations of TemplateLoader,
one for production and one for test.

```
object GreetingService(
  val loader: TemplateLoader = ???
  val renderer: TemplateRenderer = ???

  def greet(id: TemplateId, params: Params): String =
    renderer(loader(id), params)
}
```

We need to replace our hard-coded dependencies in GreetingService with dependencies that can be swapped out in different scenarios.

This is the job of DI.

Constructor-based DI

Let’s look at our first DI approach.

Super simple. We’ll inject dependencies with constructor parameters.

```scala
class GreetingService(
  loader: TemplateLoader,
  renderer: TemplateRenderer
) {
  def greet(id: TemplateId, params: Params): String =
    renderer(loader(id), params)
}
```

We turn GreetingService into a class
and inject TemplateLoader and TemplateRenderer.

```
object App {
  val loader = new S3TemplateLoader()
  val renderer = new MustacheTemplateRenderer()
  val greetings = new GreetingService(loader, renderer)
}
```

The “cost” of switching from objects to classes
is that we have to create an instance of GreetingService before we use it.

In order to create a GreetingService,
we need to create a TemplateLoader and a TemplateRenderer.

We end up with a bunch of initialisation code,
which I’ve called “App” here
but you may have seen called “Wiring” or “Components”.

```
"greeting service" should {
  "render a greeting" in {
    // Preconditions:
    val template = Template("Test greeting for {{name}}")
    val loader = new FakeTemplateLoader(template)
    val renderer = new MustacheTemplateRenderer()
    val greetings = new GreetingService(loader, renderer)

    // Test body:
    val params = Params(Map("name" -> "Dave"))
    val actual = greetings.greet(Greeting, params)
    val expected = "Test greeting for Dave"

    // Postconditions:
    actual should be(expected)
  }
}
```

Our unit tests also need wiring code.
The wiring becomes part of the preconditions of our tests.

```
trait Fixtures {
  val template = Template("Test greeting for {{name}}")
  val loader = new FakeTemplateLoader(template)
  val renderer = new MustacheTemplateRenderer()
  val greetings = new GreetingService(loader, renderer)
}

"greeting service" should {
  "render a greeting" in new Fixtures {
    // Test body:
    val params = Params(Map("name" -> "Dave"))
    val actual = greetings.greet(Greeting, params)
    val expected = "Test greeting for Dave"

    // Postconditions:
    actual should be(expected)
  }
}
```

In ScalaTest and Specs2 tests, we typically use “Fixture” classes to factor out the wiring code so we don’t repeat it in each test.

```
object App {
  val loader = new S3TemplateLoader()
  val renderer = new MustacheTemplateRenderer()
  val greetings = new GreetingService(loader, renderer)
}

trait Fixtures {
  val template = Template("Test greeting for {{name}}")
  val loader = new FakeTemplateLoader(template)
  val renderer = new MustacheTemplateRenderer()
  val greetings = new GreetingService(loader, renderer)
}
```

And there we have it.

Our code is now configurable
at the cost of wiring for our production app
and wiring for out unit tests.

Gotcha

There’s kind of a gotcha, though.

```
trait Components {
  val config: Config = Config.read(configuration) match {
    case Left(err)  => configReadError(err)
    case Right(cfg) => cfg
  }

  // Low-level config:
  val urls: Urls = Urls(config)
  val timeouts: Timeouts = new Timeouts
  val metrics: Metrics = Metrics(config)

  // Email modules:
  val emailer: Emailer = config.notification.email match {
    case config: TestEmailerConfig => new TestEmailer(config)
    case config: MailgunEmailerConfig => new MailgunEmailer(config, wsClient)
    case config: SendGridEmailerConfig => new SendGridEmailer(config)
  }

  // Database modules:
  val dbConfig: PostgresDatabaseConfig = new PostgresDatabaseConfig(config)
  val mapDatabase: MapDatabase = new SlickMapDatabase(dbConfig)
  val notificationDatabase: NotificationDatabase = new SlickNotificationDatabase(dbConfig)
  val surveyDatabase: SurveyDatabase = new SlickSurveyDatabase(dbConfig)
  val uploadDatabase: UploadDatabase = new SlickUploadDatabase(dbConfig)
  val userDatabase: UserDatabase = new SlickUserDatabase(dbConfig)

  // Redis cache modules:
  val redisClient: RedisClient = new RedisClient(config)
  val forgotCache: ForgotCache = new RedisForgotCache(redisClient)
  val sessionCache: SessionCache = new RedisSessionCache(redisClient)
  val signupCodeCache: SignupCodeCache = new RedisSignupCodeCache(redisClient)
  val unsavedSurveyCache: UnsavedSurveyCache = new RedisUnsavedSurveyCache(redisClient)

  // S3 access:
  val s3 = S3(config)

  // Service modules:
  val mapService: MapService = new MapService(mapDatabase)
  val mapboxService: MapboxService = MapboxService(config)
  val newsletterService: NewsletterService = new NewsletterService(config, wsClient, timeouts)
  val notificationService: NotificationService = new NotificationService(notificationDatabase, emailer, urls)
  val organizationService: OrganizationService = new OrganizationService
  val uploadService: UploadService = UploadService(uploadDatabase, s3, config, urls)
  val reportService: ReportService = new ReportService(uploadService, mapboxService, urls)
  val signupCodeService: SignupCodeService = new SignupCodeService(signupCodeCache, timeouts)
  val surveyService: SurveyService = new SurveyService(surveyDatabase, mapService, notificationService, unsavedSurveyCache)
  val userService: UserService = new UserService(userDatabase, forgotCache, organizationService, notificationService, newsletterService, timeouts)
  val authService: AuthService = new AuthService(sessionCache, forgotCache, userService, signupCodeService, organizationService, notificationService, newsletterService, timeouts)

  // Controller modules:
  val uploadJson: UploadJson = new UploadJson(uploadService)
  val authController: AuthController = new AuthController(authService, timeouts, urls)
  val mapController: MapController = new MapController(mapService, authService, timeouts, urls)
  val newsletterController: NewsletterController = new NewsletterController(newsletterService, authService, timeouts, urls)
  val notificationController: NotificationController = new NotificationController(notificationService, authService, timeouts, urls)
  val organizationController: OrganizationController = new OrganizationController(organizationService, authService, timeouts, urls)
  val reportController: ReportController = new ReportController(surveyService, authService, reportService, timeouts, urls, wsClient)
  val signupCodeController: SignupCodeController = new SignupCodeController(signupCodeService, authService, timeouts, urls, actorSystem)
  val surveyController: SurveyController = new SurveyController(surveyService, authService, timeouts, urls)
  val userController: UserController = new UserController(userService, authService, timeouts, urls)
  val uploadController: UploadController = new UploadController(uploadService, surveyService, authService, timeouts, uploadJson, urls)
  val preFlightController: PreFlightController = new PreFlightController
```

In a real application the wiring can get fairly verbose.

```scala
  // Redis cache modules:
  val redisClient: RedisClient = new RedisClient(config)
  val forgotCache: ForgotCache = new RedisForgotCache(redisClient)
  val sessionCache: SessionCache = new RedisSessionCache(redisClient)
  val signupCodeCache: SignupCodeCache = new RedisSignupCodeCache(redisClient)
  val unsavedSurveyCache: UnsavedSurveyCache = new RedisUnsavedSurveyCache(redisClient)

  // S3 access:
  val s3 = S3(config)

  // Service modules:
  val mapService: MapService = new MapService(mapDatabase)
  val mapboxService: MapboxService = MapboxService(config)
  val newsletterService: NewsletterService = new NewsletterService(config, wsClient, timeouts)
  val notificationService: NotificationService = new NotificationService(notificationDatabase, emailer, urls)
  val organizationService: OrganizationService = new OrganizationService
  val uploadService: UploadService = UploadService(uploadDatabase, s3, config, urls)
  val reportService: ReportService = new ReportService(uploadService, mapboxService, urls)
  val signupCodeService: SignupCodeService = new SignupCodeService(signupCodeCache, timeouts)
  val surveyService: SurveyService = new SurveyService(surveyDatabase, mapService, notificationService, unsavedSurveyCache)
  val userService: UserService = new UserService(userDatabase, forgotCache, organizationService, notificationService, newsletterService, timeouts)
  val authService: AuthService = new AuthService(sessionCache, forgotCache, userService, signupCodeService, organizationService, notificationService, newsletterService, timeouts)

  // Controller modules:
  val uploadJson: UploadJson = new UploadJson(uploadService)
  val authController: AuthController = new AuthController(authService, timeouts, urls)
  val mapController: MapController = new MapController(mapService, authService, timeouts, urls)
  val newsletterController: NewsletterController = new NewsletterController(newsletterService, authService, timeouts, urls)
  val notificationController: NotificationController = new NotificationController(notificationService, authService, timeouts, urls)
  val organizationController: OrganizationController = new OrganizationController(organizationService, authService, timeouts, urls)
  val reportController: ReportController = new ReportController(surveyService, authService, reportService, timeouts, urls, wsClient)
  val signupCodeController: SignupCodeController = new SignupCodeController(signupCodeService, authService, timeouts, urls, actorSystem)
  val surveyController: SurveyController = new SurveyController(surveyService, authService, timeouts, urls)
  val userController: UserController = new UserController(userService, authService, timeouts, urls)
  val uploadController: UploadController = new UploadController(uploadService, surveyService, authService, timeouts, uploadJson, urls)
  val preFlightController: PreFlightController = new PreFlightController
  val versionController: VersionController = new VersionController

  // HTTP modules:
  val httpFilters: Seq[EssentialFilter] = new Filters(metrics).filters
  val httpErrorHandler: ErrorHandler = new ErrorHandler(environment, configuration, sourceMapper, Option(router), metrics)
  val httpRequestHandler: RequestHandler = new RequestHandler(router, httpErrorHandler, httpConfiguration, metrics, httpFilters : _*)

  // Router:
  val router: Router = new Routes(
    httpErrorHandler,
    versionController,
    surveyController,
    reportController,
    mapController,
    uploadController,
    userController,
    notificationController,
    authController,
    signupCodeController,
    organizationController,
    newsletterController,
    preFlightController,
    prefix = ""
  )
}
```

These slides show code taken from a real app.
The whole app is 20k lines of code and the wiring is about 100.
The wiring is 0.5% of the codebase!

Personally I'm happy with this.
The code is simple and maintainable, even if it is verbose.

You may not agree with me, and you won't be alone.
There are lots of DI tools out there that are effectively designed
to reduce the verbosity of this wiring.

Let's see some.

Google Guice

The first candidate is Google Guice.

This automates a bunch of wiring
and lets us specify dependencies using annotations.

```
import com.google.inject._

@ImplementedBy(classOf[S3TemplateLoader])
trait TemplateLoader { ... }

@ImplementedBy(classOf[MustacheTemplateRenderer])
trait TemplateRenderer { ... }

class GreetingService @Inject() (
  loader: TemplateLoader,
  renderer: TemplateRenderer) { ... }

object App {
  val injector = Guice.createInjector()
  val greetings = injector
        .getInstance(classOf[GreetingService])
}
```

Here we can see how it works:

- in the App object we're using an "Injector" to create a GreetingService;
- the injector inspects the constructors of the component
  and walks the dependency graph to instantiate it;
- we use annotations like "@Inject" and "@ImplementedBy"
  (and three or four others) to help guide the process.

There are other concepts ("wiring modules")
for dealing with more complex scenarios.

Gotcha

There are a couple of gotchas with Google Guice.
It’s designed for Java, so it doesn’t know about
companion objects, apply methods, implicit parameters, and so on.

But there’s one big gotcha…

It does everything at runtime!

…it traverses the DI graph at runtime using reflection.
This means our application can fail at runtime.

Runtime failure isn’t a big deal if the app fails on startup,
but couple this with lazy component loading
and we can get errors hours or even days after deployment.

Macwire

Yes! In the form of Macwire, by Adam Warski of Software Mill.

```
object App {
  import com.softwaremill.macwire._

  val loader = wire[S3TemplateLoader]
  val renderer = wire[MustacheTemplateRenderer]
  val greetings = wire[GreetingService]
}
```

At its core, Macwire provides one simple thing:
a “wire” macro that expands to a constructor call.

“Wire” takes a type as a parameter,
inspects the corresponding class definition,
and searches for constructors
for which there are suitable parameters in scope.

It’s a bit like implicit resolution in this regard,
although it’s a completely separate process by design.

```scala
object App {
  import com.softwaremill.macwire._

  val loader = new S3TemplateLoader()
  val renderer = new MustacheTemplateRenderer()
  val greetings = new GreetingService(loader, renderer)
}
```

The wire macros expand to regular constructor calls
(and a bunch of other idiomatic ways of creating components).
Ultimately, it’s just a shorthand for constructor DI.

```
val config = Config.read(configuration) match {
  case Left(err)  => configReadError(err)
  case Right(cfg) => cfg
}

// Low-level config:
val urls: Urls = wire[Urls]
val timeouts: Timeouts = wire[Timeouts]
val metrics: Metrics = wire[Metrics]

// Email modules:
val emailer: Emailer = config.notification.email match {
  case config: TestEmailerConfig => wire[TestEmailer]
  case config: MailgunEmailerConfig => wire[MailgunEmailer]
  case config: SendGridEmailerConfig => wire[SendGridEmailer]
}

// Database modules:
val dbConfig = wire[PostgresDatabaseConfig]
val mapDatabase = wire[SlickMapDatabase]
val notificationDatabase = wire[SlickNotificationDatabase]
val surveyDatabase = wire[SlickSurveyDatabase]
val uploadDatabase = wire[SlickUploadDatabase]
val userDatabase = wire[SlickUserDatabase]

// Redis cache modules:
val redisClient = wire[RedisClient]
val forgotCache = wire[RedisForgotCache]
val sessionCache = wire[RedisSessionCache]
val signupCodeCache = wire[RedisSignupCodeCache]
val unsavedSurveyCache = wire[RedisUnsavedSurveyCache]

// S3 access:
val s3 = S3(config)

// Service modules:
val mapService = wire[MapService]
val mapboxService = wire[MapboxService]
val newsletterService = wire[NewsletterService]
val notificationService = wire[NotificationService]
val organizationService = wire[OrganizationService]
val uploadService = wire[UploadService]
val reportService = wire[ReportService]
val signupCodeService = wire[SignupCodeService]
val surveyService = wire[SurveyService]
val userService = wire[UserService]
val authService = wire[AuthService]

// Controller modules:
val uploadJson = wire[UploadJson]
val authController = wire[AuthController]
val mapController = wire[MapController]
val newsletterController = wire[NewsletterController]
val notificationController = wire[NotificationController]
val organizationController = wire[OrganizationController]
val reportController = wire[ReportController]
val signupCodeController = wire[SignupCodeController]
val surveyController = wire[SurveyController]
val userController = wire[UserController]
val uploadController = wire[UploadController]
val preFlightController = PreFlightController
```

Here’s the configuration for that real-world app again.

```
  // Email modules:
  val emailer: Emailer = config.notification.email match {
    case config: TestEmailerConfig => wire[TestEmailer]
    case config: MailgunEmailerConfig => wire[MailgunEmailer]
    case config: SendGridEmailerConfig => wire[SendGridEmailer]
  }

  // Database modules:
  val dbConfig = wire[PostgresDatabaseConfig]
  val mapDatabase = wire[SlickMapDatabase]
  val notificationDatabase = wire[SlickNotificationDatabase]
  val surveyDatabase = wire[SlickSurveyDatabase]
  val uploadDatabase = wire[SlickUploadDatabase]
  val userDatabase = wire[SlickUserDatabase]

  // Redis cache modules:
  val redisClient = wire[RedisClient]
  val forgotCache = wire[RedisForgotCache]
  val sessionCache = wire[RedisSessionCache]
  val signupCodeCache = wire[RedisSignupCodeCache]
  val unsavedSurveyCache = wire[RedisUnsavedSurveyCache]

  // S3 access:
  val s3 = S3(config)

  // Service modules:
  val mapService = wire[MapService]
  val mapboxService = wire[MapboxService]
  val newsletterService = wire[NewsletterService]
  val notificationService = wire[NotificationService]
  val organizationService = wire[OrganizationService]
  val uploadService = wire[UploadService]
  val reportService = wire[ReportService]
  val signupCodeService = wire[SignupCodeService]
  val surveyService = wire[SurveyService]
  val userService = wire[UserService]
  val authService = wire[AuthService]

  // Controller modules:
  val uploadJson = wire[UploadJson]
  val authController = wire[AuthController]
  val mapController = wire[MapController]
  val newsletterController = wire[NewsletterController]
  val notificationController = wire[NotificationController]
  val organizationController = wire[OrganizationController]
  val reportController = wire[ReportController]
  val signupCodeController = wire[SignupCodeController]
  val surveyController = wire[SurveyController]
  val userController = wire[UserController]
  val uploadController = wire[UploadController]
  val preFlightController = PreFlightController
  val versionController = VersionController

  // HTTP modules:
  val httpFilters = wire[Filters].filters
  val httpErrorHandler = wire[ErrorHandler]
  val httpRequestHandler = wire[RequestHandler]

  // Router:
  val router = wire[Routes]
```

There are still 100 lines of code,
but there are no screen-wide constructor calls.

The wiring module is effectively
a list of the components in the app.

(Note: The real version of this code uses “lazy vals” not “vals”,
for reasons we’ll see in a moment.)

Gotchas

We've seen three DI approaches and they're all fairly similar.

Here are some gotchas that you should be aware of,
regardless of which approach you use.

Gotcha #1

First one.

```
object App {
  val foo = new Foo(bar, baz)
  val bar = new Bar()
  val baz = new Baz()
}

App.foo
```

Here’s a simple wiring file.

What happens when we call “App.foo”?

```
object App {
  val foo = new Foo(bar, baz)
  val bar = new Bar()
  val baz = new Baz()
}

App.foo
// OK!
```

Nothing… yet.

When we construct “App”,
the lines in the object body are run in sequence.

The “val foo” line is run first,
creating a “Foo” and passing references to “bar” and “baz”.

But “bar” and “baz” are null at this point.
They only become non-null later.

```
object App {
  val foo = new Foo(bar, baz)
  val bar = new Bar()
  val baz = new Baz()
}

App.foo.methodThatUsesBarOrBaz
// NullPointerException
//    at ...
//    at ...
```

As soon as we call a method on “foo”
that makes use of its internal references to “bar” and “baz”,
we will get a NullPointerException.

```
object App {
  lazy val foo = new Foo(bar, baz)
  lazy val bar = new Bar()
  lazy val baz = new Baz()
}

App.foo.methodThatUsesBarOrBaz
// OK!
```

We can get around this by
declaring all concrete components using lazy vals
(always declare abstract components using defs).

This is always recommended,
whether you're using plain constructor DI or Macwire.

It causes the RHS of assignments to be evaluated as needed,
rather than as "App" is constructed.
We create "bar" and "baz" in response to creating "foo",
eliminating the nulls.

Gotcha #2

Gotcha #2…

```
class Foo(b: Bar)
class Bar(f: Foo)

object App {
  lazy val foo: Foo = new Foo(bar)
  lazy val bar: Bar = new Bar(foo)
}

App.foo
```

We have two mutually dependent modules: Foo and Bar.
They're declared as lazy vals as recommended!

What happens when I call App.foo?

```
class Foo(b: Bar)
class Bar(f: Foo)

object App {
  lazy val foo: Foo = new Foo(bar)
  lazy val bar: Bar = new Bar(foo)
}

App.foo
// StackOverflowException
//    at ...
//    at ...
```

Stack overflow exception.

When we access foo we have to instantiate it, which instantiates bar, which instantiates foo, and so on.

```scala
class Foo(b: => Bar)
class Bar(f: => Foo)

object App {
  lazy val foo: Foo = new Foo(bar)
  lazy val bar: Bar = new Bar(foo)
}

App.foo
// OK!
```

The remedy for this is to make the constructor parameters lazy (“by name” parameters).

This allows us to instantiate “foo” without forcing the instantiation of “bar”.
Later, if we access “bar” or “b”, “bar” will be instantiated
and the reference substituted into “f”.

Use by-name parameters if you want to have mutually dependent modules.
If your modules aren’t mutually dependent, you don’t need by-name.

Gotcha #3

Final gotcha.

```scala
class Foo(b: => Bar) { val x = b }
class Bar(f: => Foo) { val y = f }

object App {
  lazy val foo: Foo = new Foo(bar)
  lazy val bar: Bar = new Bar(foo)
}

App.foo
```

Now suppose I have mutually dependent modules
with lazy vals and by-name parameters.

BUT I’m referring to the by-name parameters
in the constructors of each module.

What happens now?

```scala
class Foo(b: => Bar) { val x = b }
class Bar(f: => Foo) { val y = f }

object App {
  lazy val foo: Foo = new Foo(bar)
  lazy val bar: Bar = new Bar(foo)
}

App.foo
// StackOverflowException
//    at ...
//    at ...
```

Another stack overflow.

Referencing the dependencies within the constructors
requires us to evaluate the by name reference.

This effectively cancels out the laziness,
causing the same infinite recursion.

```scala
class Foo(b: => Bar) { def x = b }
class Bar(f: => Foo) { def y = f }

object App {
  lazy val foo: Foo = new Foo(bar)
  lazy val bar: Bar = new Bar(foo)
}

App.foo
```

If we change the “vals” to “defs”,
we move from referencing the dependencies at construction time
to referencing them *after* construction.

This is fine because by this point “foo” and “bar” have both been created.

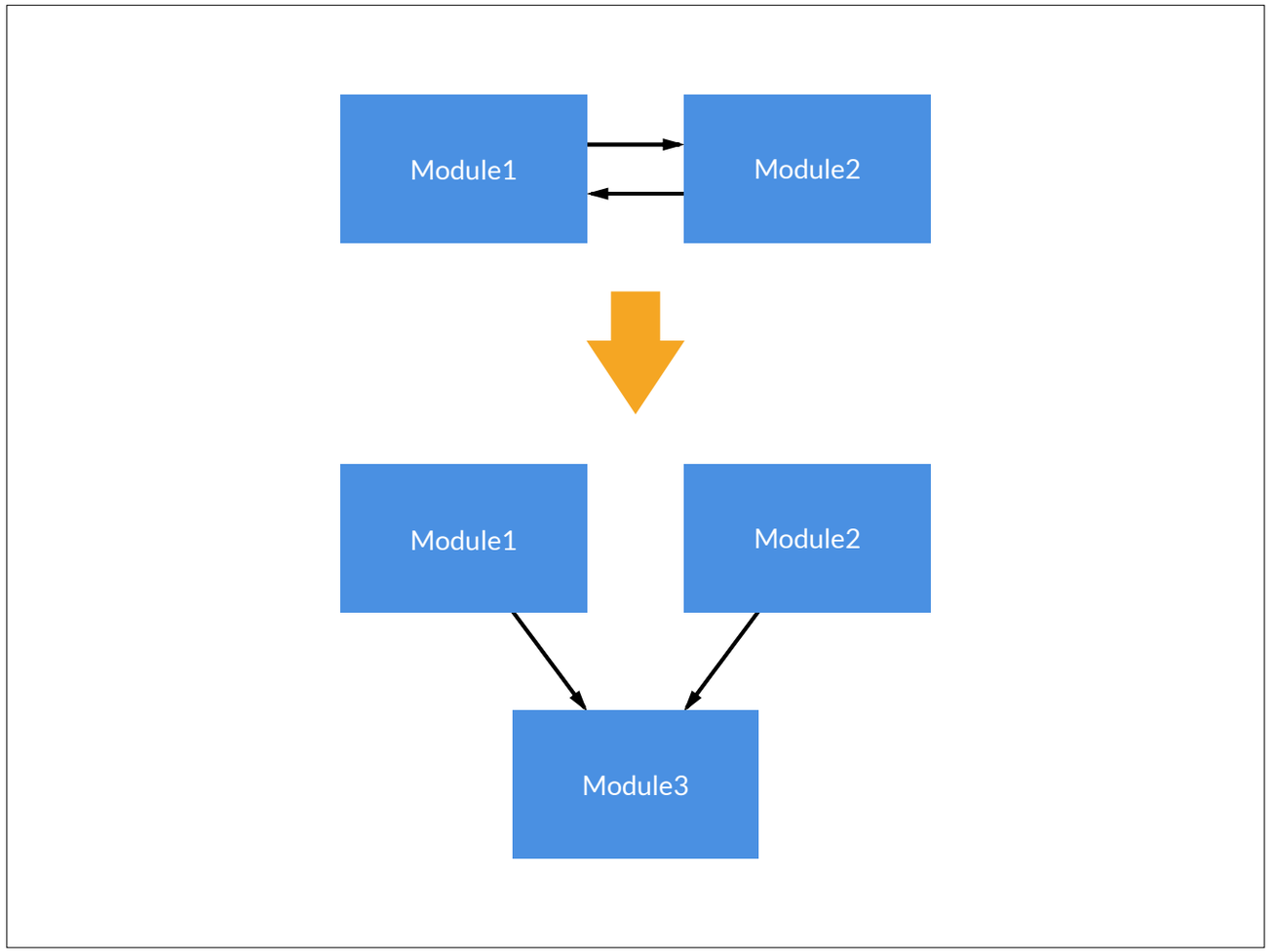
```scala
class Foo(b: => Bar) { lazy val x = b }
class Bar(f: => Foo) { lazy val y = f }

object App {
  lazy val foo: Foo = new Foo(bar)
  lazy val bar: Bar = new Bar(foo)
}

App.foo
```

Same thing with “lazy vals”.

These are not initialised as the components are constructed,
so they don’t cause infinite recursion.

If these gotchas have you worried, you’re not alone.

Dave Mustaine is also concerned by this brittleness.



That said, remember that two our of three of theses gotchas
are only applicable when we have mutually dependent modules.

Take solace in the fact that you can always refactor code
to remove mutual dependencies.
Simply take the mutually dependent parts
and move them into a shared dependency.

Or allow mutual dependencies in your code and JUST… BE… CAREFUL!

Trait-based DI

Some people prefer organising their applications using traits instead of classes.

The common folklore is that traits are “better” at handling things like cyclic dependencies, so they should be better for doing DI.

Let’s find out if that’s true…

```scala
trait TemplateLoader {
  def load(id: TemplateId): Template
}

trait TemplateRenderer {
  def render(template: Template, params: Params): String
}

trait GreetingService {
  self: TemplateLoader with TemplateRenderer =>

  def greet(id: TemplateId, params: Params): String =
    render(load(id), params)
}
```

Here’s the refactor.

GreetingService is now a trait with a self-type.
The self-type ensures that every concrete instance of GreetingService
is also a TemplateLoader and a TemplateRenderer.

This allows us to rely on the load() and render() methods
in the body of GreetingService.

```
object App extends GreetingService
  with S3TemplateLoader
  with MustacheTemplateRenderer

App.greet(...)
```

We do DI by mixing components together to create an application object.
The self-types ensure we always mix in the correct dependencies.

All seems simple.
However, there’s one significant downside to this approach.

Naming things!

One of the three great problems of computer science. Naming things.

```scala
trait TemplateLoader {
  def load(id: TemplateId): Template
}

trait TemplateRenderer {
  def render(template: Template, params: Params): String
}

trait GreetingService {
  self: TemplateLoader with TemplateRenderer =>

  def greet(id: TemplateId, params: Params): String =
    render(load(id), params)
}
```

Here's the code again.

Did you notice I've renamed the methods
from TemplateLoader and TemplateRenderer?

They're called "load" and "render"
but they used to be called something else…

```scala
trait TemplateLoader {
  def apply(id: TemplateId): Template
}

trait TemplateRenderer {
  def apply(template: Template, params: Params): String
}

trait GreetingService {
  self: TemplateLoader with TemplateRenderer =>

  def greet(id: TemplateId, params: Params): String =
    apply(apply(id), params)
}
```

They used to be called apply().

I renamed them because
the “apply(apply())” line in GreetingService is confusing!

We’re not able to rename methods when we inherit them in Scala.
This can cause confusing name clashes or even compiler errors.

This direct inheritance pattern,
while great for small sets of traits,
isn’t suitable as a general DI mechanism.

“Thin” Cake Pattern

Enter something called the “thin cake pattern”,
a phrase coined by Macwire’s own Adam Warski.

In this pattern we switch back to using constructor-based DI,
but specify dependencies by mixing together “module” traits
that contain references to components and their dependencies.

```scala
trait TemplateLoaderModule {
  def loader: TemplateLoader
}

trait S3TemplateLoaderModule
    extends TemplateLoaderModule {
  lazy val loader: TemplateLoader =
    new S3TemplateLoader()
}
```

For example, here is a module for TemplateLoader.

The variable “loader” refers to the same “TemplateLoader” trait
we used in the constructor-based DI approach.

TemplateLoader has an “apply()” method again.
The name is no longer going to be a problem.

We also create a module for S3TemplateLoader.
This module extends the TemplateLoaderModule
and specifies a concrete definition of “loader”.

```
trait GreetingServiceModule {
  self: TemplateLoaderModule
      with TemplateRendererModule =>

  lazy val greeter: GreetingService =
    new GreetingService(loader, renderer)
}
```

Now we create a module to wrap “GreetingService”.
Again, this is the class version of GreetingService from our constructor-based DI.

We include a self-type that refers
to the module wrappers for TemplateLoader and TemplateRenderer.
This guarantees that any concrete instance of GreetingService
will include variables called “loader” and “renderer”.

We can call “loader.apply()” and “renderer.apply()”
without worrying that the methods have the same name.
The dependency names disambiguate for us.

```
object App extends GreetingServiceModule
  with S3TemplateLoaderModule
  with MustacheTemplateRendererModule
```

Like the direct trait approach, we do DI by mixing the traits together.
In this case, though, we're mixing the modules instead of the components within.

The term "cake pattern" alludes to the fact that
our application object resembles a cake assembled from a number of module "layers".

Hang on a minute…

There’s an important point to be made here.

```
object App extends GreetingServiceModule
  with S3TemplateLoaderModule
  with MustacheTemplateRendererModule
```

Look at this cake, made from its three layers.

Each layer introduces a number of statements to the constructor:
- S3TemplateLoaderModule creates an S3TemplateLoader
  and assigns it to a variable called “loader”;
- MustacheTemplateRendererModule creates an MustacheTemplateRenderer
  and assigns it to a variable called “renderer”;
- GreetingServiceModule creates a GreetingService
  and assigns it to a variable called “greeting”.

Imagine what the constructor for “App” would look like
if we wrote it all out line by line…

```scala
object App {
  val loader = new S3TemplateLoader()
  val renderer = new MustacheTemplateRenderer()
  val greetings = new GreetingService(loader, renderer)
}
```

It would look exactly like our original wiring module
from our constructor-based DI approach.

All thin cake is doing is compartmentalising our wiring
into a set of smaller modules.

In my mind, it's also making the wiring harder to visualise
because we have to look in multiple places in the codebase
and think about self-types and mixing ordering.

I personally prefer the big-single-wiring-module approach we saw earlier.
But YMMV, of course.

“Full” Cake Pattern

That was the thin cake pattern.

For the sake of completeness let’s look at the “full cake” pattern and why these days it’s commonly considered an anti-pattern.

```
trait TemplateLoaderModule {
  trait TemplateLoader {
    def load(id: TemplateId): Template
  }

  def loader: TemplateLoader
}
```

The difference between full cake and thin cake
is that in full cake we move the definitions of components
into the bodies of their respective modules.

```
trait S3TemplateLoaderModule extends TemplateLoaderModule {
  class S3TemplateLoader extends TemplateLoader {
    ...
  }

  lazy val loader = new S3TemplateLoader()
}
```

This makes modules completely self-contained,
which seems like a good idea.

Also, because we're no longer defining components at the "top level"
(we're defining them inside traits instead of inside packages),
we can have components that are vals, implicits, and type aliases,
in addition to components that are traits, classes, and objects.

```
trait GreetingServiceModule {
  self: TemplateLoaderModule
      with TemplateRendererModule =>

  trait GreetingService {
    def greet(id: TemplateId, params: Params): String =
      renderer(loader(id), params)
  }

  lazy val greeter: GreetingService =
    new GreetingService(loader, renderer)
}
```

We still specify dependencies using self-types.

```
object App extends GreetingServiceModule
  with S3TemplateLoaderModule
  with MustacheTemplateRendererModule
```

And we still wire things together by mixing traits.

Gotcha

There’s a huge gotcha with this, though,
which is why the full cake pattern has generally fallen out of favour.

```
object App extends GreetingServiceModule
  with S3TemplateLoaderModule
  with MustacheTemplateRendererModule

val loader = App.loader
```

When we “bake a cake”,
every component in that cake is defined as part of the application object.

To illustrate this, consider this question…
What is the type of “loader” in the code on this slide?

Hint: it’s not just “TemplateLoader”.

```scala
object App extends GreetingServiceModule
  with S3TemplateLoaderModule
  with MustacheTemplateRendererModule

val loader = App.loader
// loader: App.TemplateLoader = ...
```

It’s “App.TemplateLoader”.

The type of the template loader is parameterised by the type of “App”.

This means we can’t mix components from different application cakes.
They will always have incompatible types.

If we want to use a component from a module in our application,
we either have to instantiate the whole application object,
or (more likely) we have to write a new component contained in a new module.

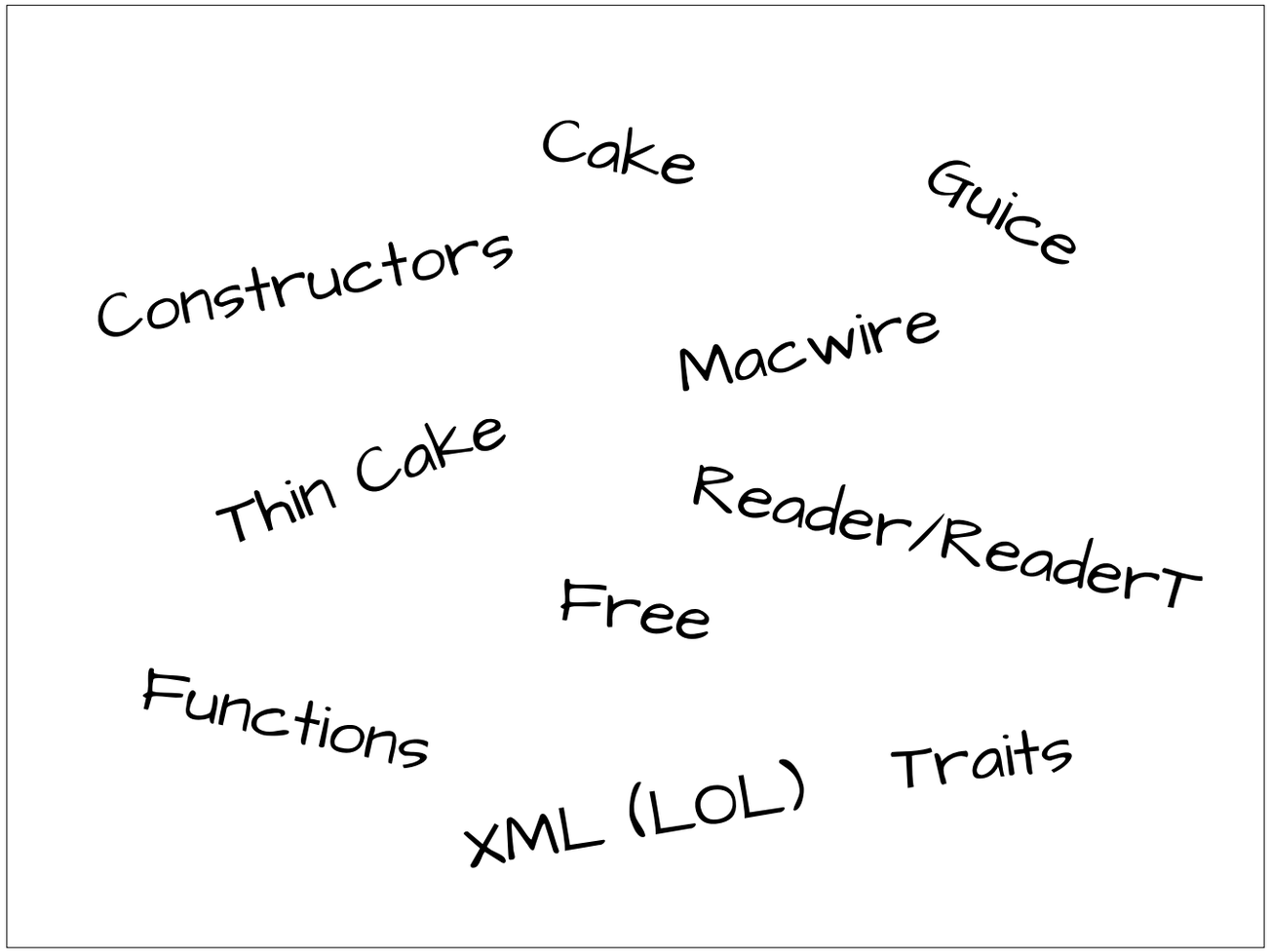
Bakery of DOOM

Once we start using the cake pattern,
we end up trapped in the cake pattern.

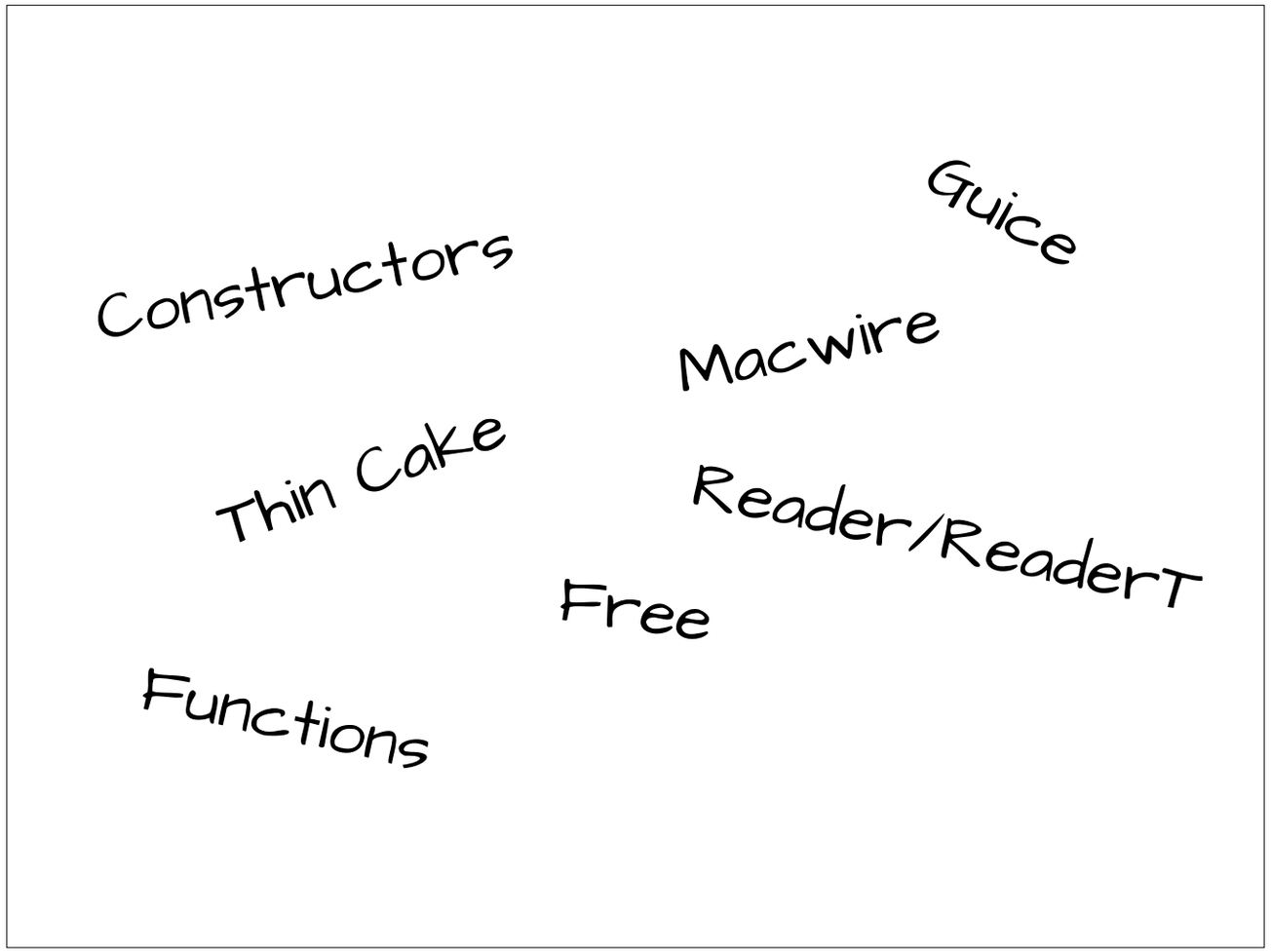
This inspired the infamous term “Bakery of Doom”.

In Summary…

In summary.

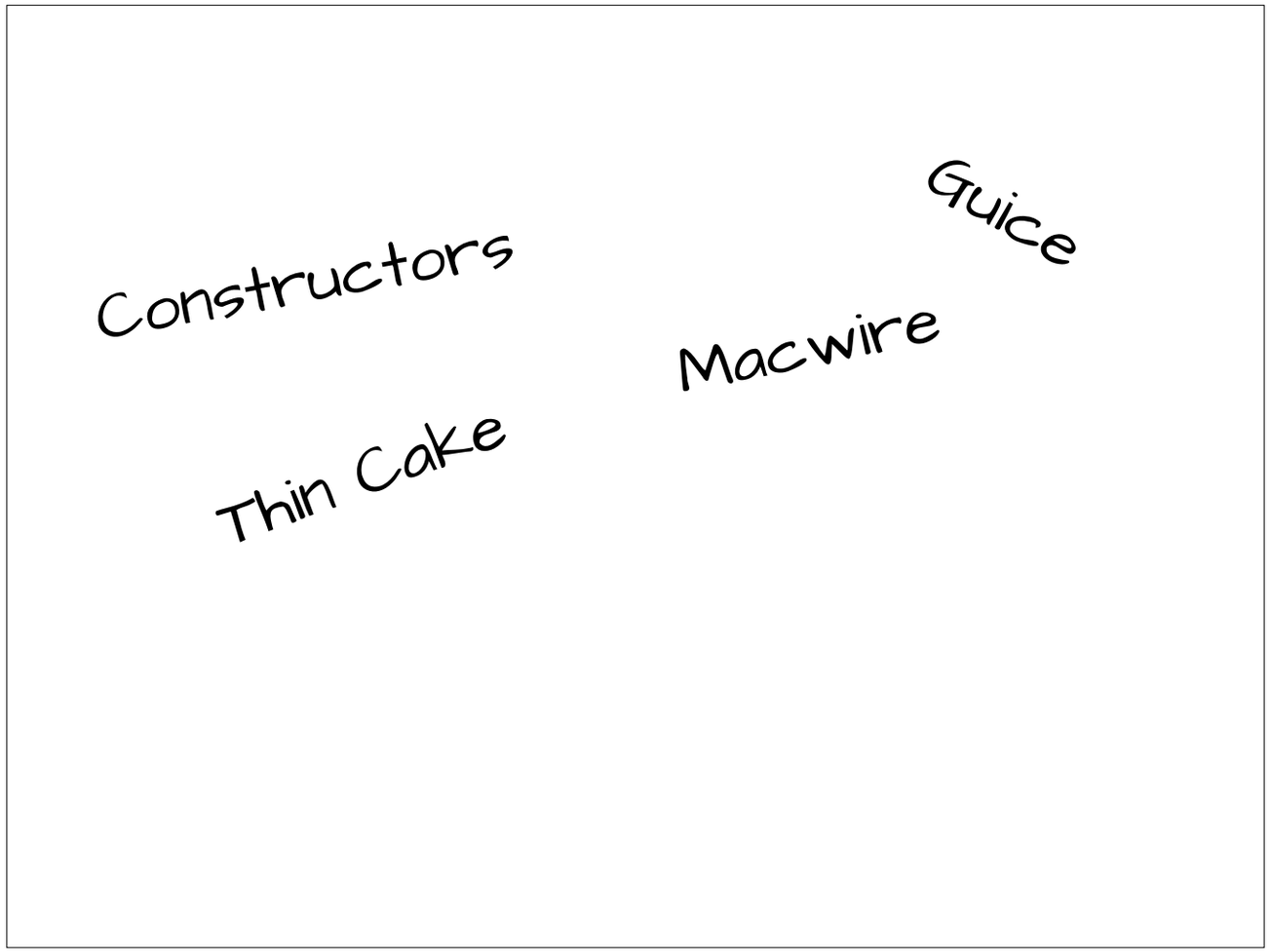


We started by looking at the myriad of approaches to DI in Scala.



We eliminated some approaches as being dead ends:

- direct trait-based DI doesn’t scale due to naming conflicts;
- the cake pattern is a nightmare;
- XML-based DI is (and was) a joke.



Then there were some things we simply didn't have time to cover:
functions, readers, and free.

I hope to cover these in a future talk.
I'll come back to them in a moment.

Constructors

Guice

Macwire

Thin Cake

The key take home points are:

- these are the most common forms of DI in Scala;
- they're all basically different versions of constructor-based DI;
- they're all (with the possible exception of Guice) subject to gotchas to do with initialization order and cyclic dependencies.

So basically, don't worry about it.
Pick a DI approach and run with it.
You can easily swap it out later!

We didn’t cover…

The stuff we didn’t cover…

Functions
Reader/ReaderT

Free

Plain function DI involves assembling an application from individual functions.
It naturally gives rise to the Reader monad,
which allows us to compose functions without repeatedly specifying their dependencies over and over.
The Reader monad then naturally gives rise to the ReaderT monad transformer, AKA the “Kleisli”.
This stuff is provided by Cats and Scalaz.

We also didn’t talk about the Free monad,
which is gaining popularity among functional Scala folks.

Free allows you to specify sequences of operations as simple data structures
that have to be fed to an “interpreter” to do anything practical.
The interpreter carries all our dependencies,
and we can swap interpreters out for testing.

I’ve put some references regarding Reader and Free on the next slides.

References

DI approaches we covered here…

Krzysztof Pado - Dependency injection in Play Framework using Scala
http://www.schibsted.pl/blog/dependency-injection-play-framework-scala

Macwire User Guide (project README)
https://github.com/adamw/macwire

Google Guice User Guide
https://github.com/google/guice/wiki/Motivation

Adam Warski - Using Scala traits as modules, or the "Thin Cake" Pattern
http://www.warski.org/blog/2014/02/using-scala-traits-as-modules-or-the-thin-cake-pattern

Mark Harrison - Cake Pattern in Depth
http://www.cakesolutions.net/teamblogs/2011/12/19/cake-pattern-in-depth

Andrew Rollins - The Cake Pattern in Scala - Self Type Annotations vs. Inheritance
http://www.andrewrollins.com/2014/08/07/scala-cake-pattern-self-type-annotations-vs-inheritance

Krzysztof Pado's blog post on DI in Play is a must-read.
He contrasts all of the approaches we covered here,
plus the Reader monad,
using a Play web application as a running example.

Functional approaches…

Mark Seemann - Dependency injection is passing an argument
http://blog.ploeh.dk/2017/01/27/dependency-injection-is-passing-an-argument

Mark Seemann - Partial application is dependency injection
http://blog.ploeh.dk/2017/01/30/partial-application-is-dependency-injection

Mark Seemann - Dependency rejection
http://blog.ploeh.dk/2017/02/02/dependency-rejection

The Reader monad…

Jason Arhart - Scrap Your Cake Pattern Boilerplate: DI Using the Reader Monad
http://blog.originate.com/blog/2013/10/21/reader-monad-for-dependency-injection/

The Free monad…

Pere Villega - On Free Monads
http://perevillega.com/understanding-free-monads

Pere Villega - Free Monads using FreeK
http://perevillega.com/freek-and-free-monads

I highly recommend Perl Villega's blog posts on Free.
They are really clear and concise.

Mark Seemann's trilogy of posts on function DI
is worth a read from an academic perspective.