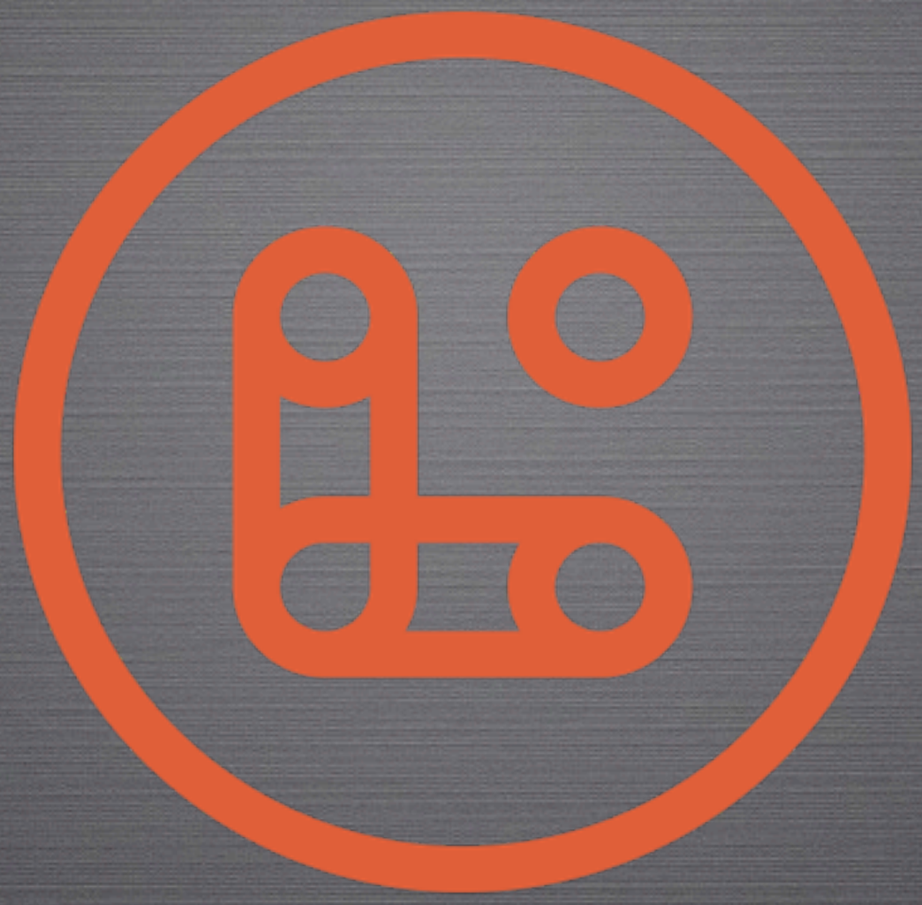
Objectify

Dependency Injection for
Scala Web Applications



Learndot

Arthur Gonigberg
arthur.gonigberg.com

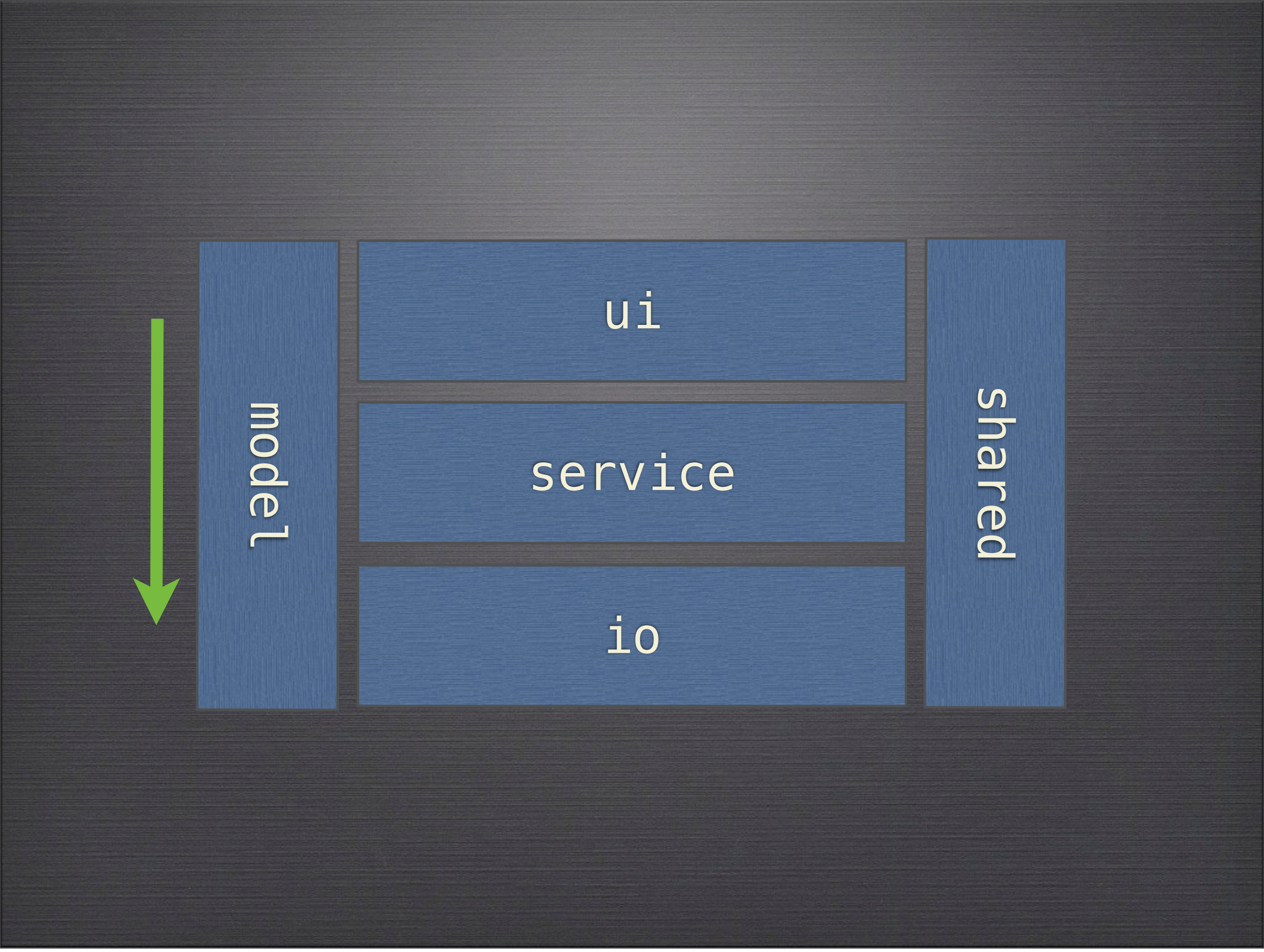
What is Objectify?

- Lightweight framework that integrates with Scalatra and helps you structure your web application
- Inspired by James Gollick's similarly named Ruby framework -- moving Ruby apps closer to Java

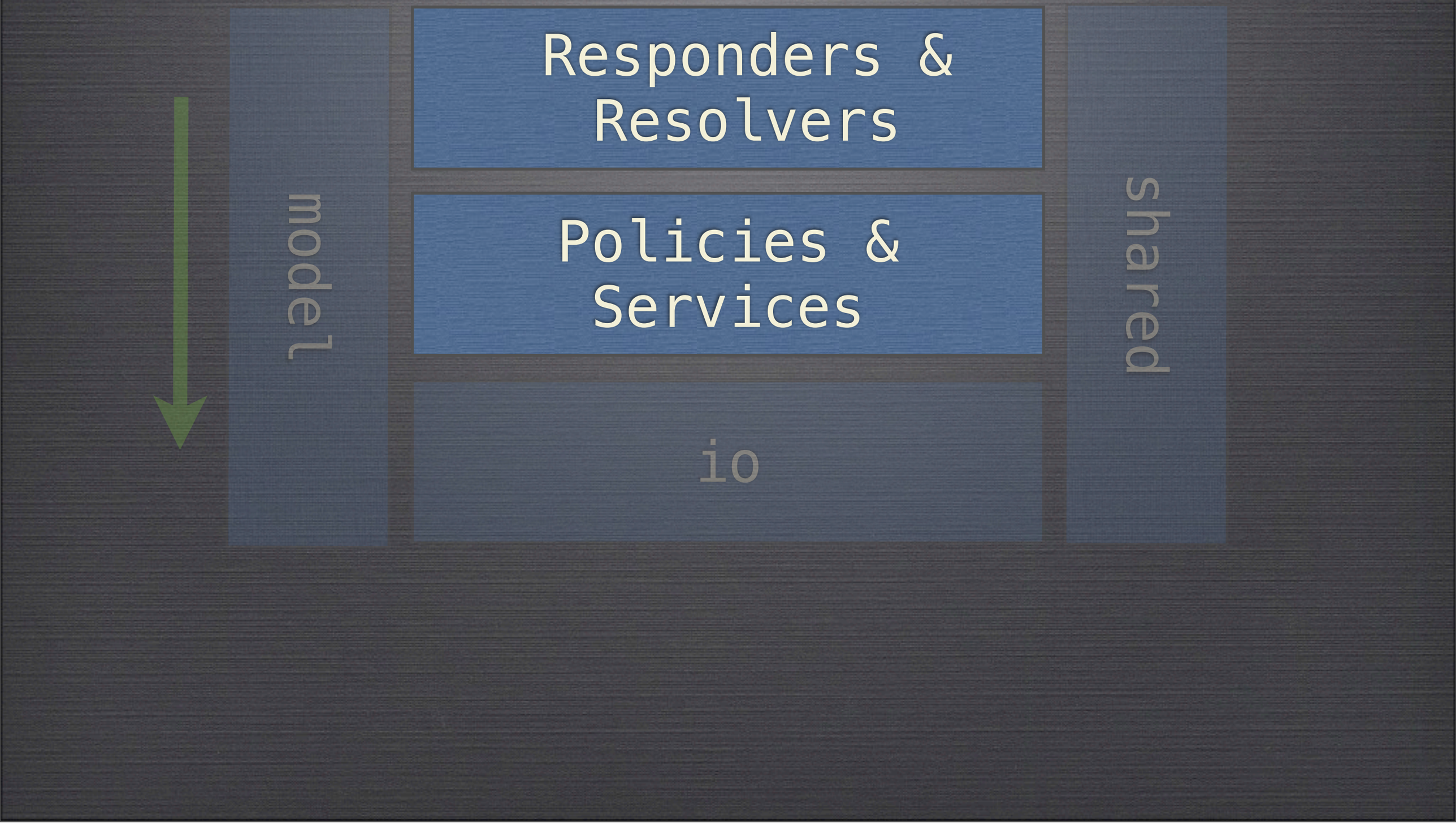
Why Objectify?

- With Scalatra services were defined all over the place with posts/gets/puts etc all in various files -- api definition all over the place
- Shared functionality and codification was done via inheritance

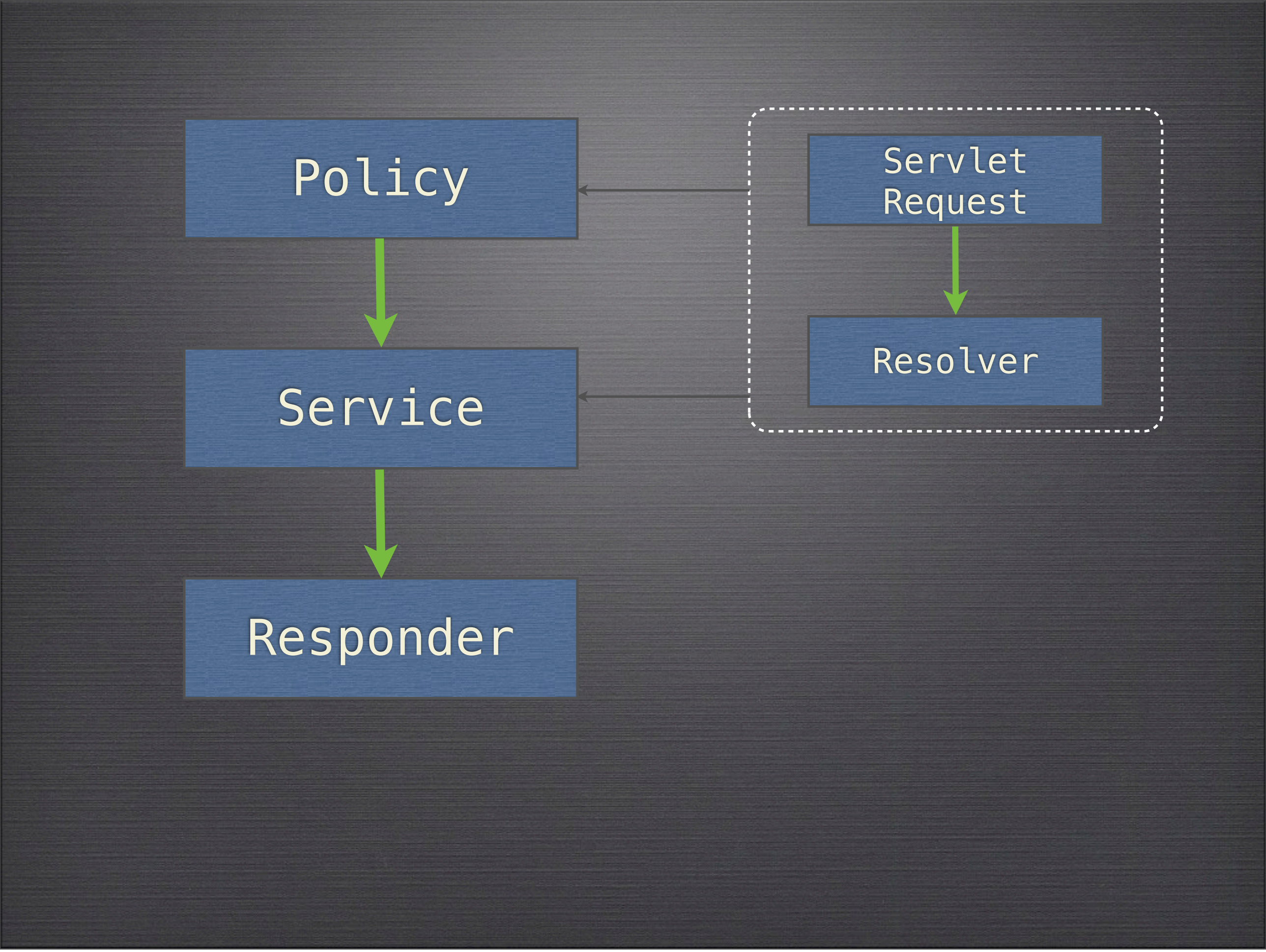
- Codify how services are structured, and how they should function
- Central definition of all paths, policies, responders -- all in one spot
- Single responsibility for classes/separation of concerns
- Injection without singletons



- Any application can be broken down into these parts
- Each section is isolated, with clearly defined interfaces and therefore can be tested sanely
- Green arrow indicates dependencies going downwards -- ui only depends on svc etc..



- Objectify addresses these two segments and seeks to codify and separate their roles
- There are essentially only 4 types of classes to make here



- Inside Scalatra/Servlet request handling scope -- i.e. that thread
- Parse request using resolvers for any particular injectables
- Inject values by type/name into Policies and Services
- If the policy fails, it goes to a policy responder
- If all the policies are pass, execute service
- Pass the service result to the responder and format the result

Why Scala is Awesome!

- Scala is really well suited to this
- Higher order functions, implicits, closures etc. are amazing for building a DSL
- Type safety allows you to maintain sanity when building and testing
- Ability to rely on well-tested and widely-used frameworks from Java land

So how do we use it?

- Some code examples ahead

Basic Definition

```
import org.objectify.adapters.ObjectifyScalatraAdapter
import org.scalatra.ScalatraFilter

object ExampleObjectify extends ObjectifyScalatraAdapter with ScalatraFilter {

    /* Resources */
    actions resource ("users")
}
```

- First, object declaration with two superclasses - the objectify adapter for scalatra and the scalatra filter class for inserting into your app
- Next, the default resource definition - this is a conventional definition and the framework will expect a service and responder class for each action type in the framework
- This can be overridden as you'll see later

Conventional Classes

```
class UserIndexService extends Service[User]{
    def apply() = new User("arthur")
}

class UserIndexResponder extends ServiceResponder[String, User]{
    def apply(user: User) = user.name
}
```

- This is just one of the actions the resource expects
- By default it expects a bunch (still under consideration) but definitely Index, Create, Show, Update, Delete
- The relationships are typed and verified by the framework

Global policies, local policies

Policy Definitions

```
object ExampleObjectify extends ObjectifyScalatraAdapter with ScalatraFilter {

    /* Global policies */
    defaults globalPolicy (~:[AuthenticationPolicy] -> ~:[BadAuthenticationResponder])

    /* Resources */
    actions resource ("users") policies (
        ~:[CourseInstructorPolicy] -> ~:[BadAuthorizationResponder] only ("create"),
        ~:[CourseStudentPolicy] -> ~:[BadAuthorizationResponder] only ("index"),
    )
}
```

- Here there is a global policy that applies to all resources
- Also there is an example of adding policies to specific actions (on top of the global policy)

What about Resolvers?

- The services and policies are pretty useless without access to the request in some capacity
- Reusable, single-responsibility classes that can be chained

Injecting Resolvers

```
class UsersIndexService(@Named("CurrentUserResolver") userOpt: Option[User],
                        @Named("CourseIdResolver") courseIdOpt: Option[List[Int]])
    extends Service[List[User]] {

    def apply(): List[User] = {
        val users = if (courseIdOpt.isDefined && courseIdOpt.get.size == 1) {
            UserDao.usersByCourse(courseIdOpt.get.head).toList
        }
        else if (userOpt.isDefined && userOpt.get.isAdmin) {
            UserDao.all.toList
        }
        else {
            Nil
        }

        UserService.computeJsonSerializableFields(users)
    }
}
```

- Here is the actual UsersIndexService from our application
- userOpt and courseIdOpt are injected at construction time

User Resolver

```
class CurrentUserResolver(sessionSvc: SessionService)
    extends Resolver[Option[User], ObjectifyRequestAdapter] {

    def apply(req: ObjectifyRequestAdapter) = {...}
}
```

- the sessionSvc is injected by type rather than name (unlike previous slides)
- the Request Adapter is just a wrapper for a Scalatra request
- and the apply method takes the request and the sessionSvc (i.e. the cookie service) and figures out the user

Simple Testing

```
@RunWith(classOf[JUnitRunner])
class UsersIndexServiceTest extends UserServiceTestScaffolding {
    "Calling the service" should {
        "generate the empty list for non-admin user w/o courseId" in {
            val service = new UsersIndexService(Some(user), None)
            service() should equal(Nil)
        }
        "generate the correct result for that course" in {
            val service = new UsersIndexService(Some(user), Some(List(course.id)))
            service() should equal(Nil)
        }
        "generate the correct result" in {
            val service = new UsersIndexService(Some(userAdmin), None)
            service() should equal(UserDao.all.toList)
        }
    }
}
```

- testing becomes very easy
- no need to worry about dependencies or huge acceptance tests
- creating mock injectables and pass them in to unit test the various components

Advanced Examples

```
actions resource ("uploadFile", pluralize = false) onlyRoute ("create" -> "files/upload",
    "show" -> "files/upload/:id")

actions resource ("uploadFilePublic", pluralize = false) onlyRoute ("create" -> "files/upload/public",
    "show" -> "files/upload/public/:id") ignoreGlobalPoliciesOnly ("show")

actions resource ("filePreview", pluralize = false) onlyRoute ("show" -> "files/upload/:id/preview")

actions resource ("login", pluralize = false) onlyRoute ("create" -> "login", "show" -> "login",
    "destroy" -> "login") ignoreGlobalPolicies ()
```

```
actions resource ("resetPassword", pluralize = false) onlyRoute ("show" -> "resetPassword/:token",
    "create" -> "resetPassword", "update" -> "resetPassword") ignoreGlobalPolicies ()
```

- scalatra-like route definition, with variables -- specify only the actions you want
- upload public file, ignore global policy for specific action -- login/reset pw, ignore all global policies
- can choose to pluralize or not to pluralize the route name - e.g. LoginCreateService vs LoginsCreateService

Benefits of Objectify

- how does it help us?

Simplicity

- it’s a very, very small framework -- less than 1000 lines of code
- there’s no magic, it doesn’t do anything complicated

Conciseness

- using the Scala DSL makes configuration very concise and easy but also allows for complexity and customization at almost every level
- despite having more classes, they will all be very short and manageable
- small focused classes, means small focused tests

Simpler, shorter tests

- biggest advantage so far has been the dramatic speed increase in running the tests
- the unit tests are self explanatory and concise

Better Maintainability

- this is probably the biggest benefit
- since all the services, responders and policies are mostly decoupled you can be sure that making a change in one service will not affect any other services -- most of the logic you need to tweak will be in these services
- conventional structure promotes consistency so there is no guess work in figuring out the behaviour of an API call

Going Forward

- first goal is open sourcing the framework
- this requires much better documentation and some code comments/cleanup
- no stability issues but also hasn't been tested under a huge load -- need more production usage
- support other web frameworks

Questions?

maybe demo if people are interested