Scalable Component Abstractions

Chris Addie

Datacom TSS

July 26, 2017

Table of Contents

Introduction

Dependencies

Scalable Component Abstractions

Table of Contents

Introduction

Dependencies

Scalable Component Abstractions

Abstract and Audience

I'll be discussing a component programming pattern described by Martin Odersky and Matthias Zenger in their highly influential paper *Scalable Component Abstractions*. SCA Provides the mechanisms for describing software components, their dependencies, and those dependencies satisfaction.

Audience:

- This presentation intended for imperative and functional programmers interested in component programming
- Code samples will be in Scala

Background

- Component-based software engineering is the abstract idea of building system from discrete components that each fulfil a specific purpose.
- Components can be services, Actors, Objects, etc.
- Generally Software libraries are not considered components, but specific library modules could be.
- Component programming aims to maximise code re-use by building generic components that can be used by multiple systems.

Table of Contents

Introduction

Dependencies

Scalable Component Abstractions

Dependencies

- Dependencies are the components required by a component for it to operate
- Generally it is desirable to describe and satisfy dependencies separately for the purposes of modularity, testability, and deployment.
- A dependency might be as simple as a structure of immutable configuration values
  - It is desirable to describe such configuration abstractly to avoid being opinionated about how it is provided
  - If the component loaded config directly from a file it could not be strictly functional
  - When writing tests we may not want to have to prepare a config file
  - In production, config may be provided from a URL rather than a local file, etc
- Abstractly describing dependencies is at the very core of component programming because it is essential to keep components discrete and focused

Dependencies Continued

- One of the pitfalls of component programming, is how to describe and satisfy the dependencies of software components
- In the model of *components as classes* dependencies can be specified parameters to the class constructor
- *Parameterisation from the top down* has the undesirable property that for class A to construct class B it must carry the dependencies for B in the constructor for A
- Dependency injection frameworks such as Spring solve this problem by allowing dependencies to be satisfied at runtime via configuration, but this has the effect of converting compile time errors into run time errors

Table of Contents

Introduction

Dependencies

Scalable Component Abstractions

Scalable Component Abstractions

- SCA aims to solve the problem of how to model components, describe dependencies, and then satisfy them entirely at compile time using three Scala language features:
  - Abstract Type Interfaces
  - Self Type Annotation
  - Modular Mixin Composition

Abstract Type Interfaces

- Abstract Type Interfaces are in Scala are known as *Traits*
- Traits are a collection of values behaviors
- Traits always have a no-args constructor
- Traits can be fully abstract, partially abstract, or fully concrete

Trait Example

```
trait HttpSink[T] {
  \\Implementation removed for brevity
  protected val httpSink: Sink[T, Future[akka.Done]]
}
```

Self Type Annotation

- Self type annotation allows for a component to describe what it's dependencies are
- In Scala Traits can describe what additional types they must have in order to be constructed by providing a type hint on *this*

Self Type Example

```
trait HttpSinkConfig {
  protected val method: HttpMethod = HttpMethods.POST
  protected val path: String
}
trait MaterializerProvider {
  protected implicit val mat: Materializer
  protected implicit val execContext: ExecutionContext
}
trait HttpFlowProvider[T] {
  this: MaterializerProvider =>
  protected def httpFlow: Flow[(HttpRequest, T), (Try[HttpResponse], T)
}
trait HttpSink[T] {
  this: HttpFlowProvider[T] with
    MaterializerProvider with
    HttpSinkConfig with
    Marshalling[T] with Logging =>
  protected val httpSink: Sink[T, Future[akka.Done]]
}
```

Modular Mixin Composition

- MMC is the final component of SCA that allows for dependencies to be satisfied
- In Scala a Class or Trait can extend exactly one class, but any number of traits
- Furthermore, not all self type annotations must be satisfied, only those that are not re-stated

Production Injection

```
import dtss.soc.common.security.{ Event, EventMarshalling }
trait ProductionEventSink extends HttpSink[Event] with
  HttpFlowProvider[Event] with
  HttpSinkConfig with
  EventMarshalling {
  this: Logging with ActorSystemProvider =>

  import dtss.soc.common.conf
  private def eventSinkHost = conf.getString("common.eventSinkHost")
  private def eventSinkPort = conf.getInt("common.eventSinkPort")
  protected val path = "/event"
  override protected def httpFlow = akka.http.scaladsl.Http()
    .cachedHostConnectionPool[Event](eventSinkHost, eventSinkPort)
}
```

Test Injection

```
val mockHttp = mockFunction[HttpRequest, Try[HttpResponse]]
def sendRecieve(t: Tuple2[HttpRequest, Event]) =
  mockHttp(t._1) -> t._2

trait TestEventSink extends HttpSink[Event] with
  HttpFlowProvider[Event] with
  HttpSinkConfig with
  EventMarshalling with
  MockLogging with
  ActorSystemProvider {

  val system = _system
  private def eventSinkHost = "127.0.0.1"
  private def eventSinkPort = 80
  protected val path = "/event"
  override protected def httpFlow =
    Flow[(HttpRequest, Event)].map(sendRecieve)
}
```

Questions?