Advanced Dagger2

Justin Inácio
ANDROID DEVELOPER

https://moducode.com

Our Current Component

```
@Component(modules = [SchedulerModule::class, DatabaseModule::class,
EpisodeServiceModule::class])
interface AppComponent {

      fun dbRepo(): DbRepo

      fun schedulers(): SchedulersBase

      fun episodeService(): EpisodeService
}
```

The Problem

```
class EpisodeListPresenter(private val episodeService: EpisodeService,
                           private val schedulers: SchedulersBase,
                           private val dbRepo: DbRepo)
```

The Problem

```
class EpisodeListPresenter(private val episodeService: EpisodeService,
                           private val schedulers: SchedulersBase,
                           private val dbRepo: DbRepo)


fun EpisodeListFragment.buildPresenter(): EpisodeListContract.Actions {
    val component = DaggerAppComponent
            .builder()
            .contextModule(ContextModule(activity?.applicationContext!!))
            .build()

    return EpisodeListPresenter(component.episodeService(),
                                component.schedulers(),
                                component.dbRepo())
}
```

How do we Improve this?

How do we Improve this?

By using @Inject

Constructor Injection

```
class EpisodeListPresenter(private val episodeService: EpisodeService,
                           private val schedulers: SchedulersBase,
                           private val dbRepo: DbRepo)
```

Constructor Injection

```
class EpisodeListPresenter @Inject constructor(
                           private val episodeService: EpisodeService,
                           private val schedulers: SchedulersBase,
                           private val dbRepo: DbRepo)
```

Constructor Injection

```
class EpisodeDetailPresenter @Inject constructor(
                             private val dbRepo: DbRepo,
                             private val schedulers: SchedulersBase)
```

Our Current Component

```
@Component(modules = [SchedulerModule::class, DatabaseModule::class,
EpisodeServiceModule::class])
interface AppComponent {

      fun dbRepo(): DbRepo

      fun schedulers(): SchedulersBase

      fun episodeService(): EpisodeService
}
```

Our New Component

```
@Component(modules = [SchedulerModule::class, DatabaseModule::class,
EpisodeServiceModule::class])
interface AppComponent {

      fun buildEpisodeDetailPresenter(): EpisodeDetailPresenter

      fun buildEpisodeListPresenter(): EpisodeListPresenter
}
```

The Problem

```
fun EpisodeListFragment.buildPresenter(): EpisodeListContract.Actions {
    val component = DaggerAppComponent
            .builder()
            .contextModule(ContextModule(activity?.applicationContext!!))
            .build()

    return EpisodeListPresenter(component.episodeService(),
                                component.schedulers(),
                                component.dbRepo())
}
```

The Problem... Solved!

```
fun EpisodeListFragment.buildPresenter(): EpisodeListContract.Actions =
        DaggerAppComponent
                .builder()
                .contextModule(ContextModule(activity?.applicationContext!!))
                .build()
                .buildEpisodeListPresenter()
}
```

Field Injection

```
class EpisodeListPresenter {

      private val episodeService: EpisodeService = ...
      private val schedulers: SchedulersBase = ...
      private val dbRepo: DbRepo = ...

}
```

Field Injection

```
class EpisodeListPresenter {

      @Inject lateinit var episodeService: EpisodeService
      @Inject lateinit var schedulers: SchedulersBase
      @Inject lateinit var dbRepo: DbRepo
}
```

Writing An Inject Function

```
@Component(modules = [SchedulerModule::class, DatabaseModule::class,
EpisodeServiceModule::class])
interface AppComponent {

      fun injectPresenter(presenter: EpisodeListPresenter)

}
```

Field Injection

```
class EpisodeListPresenter {

      @Inject lateinit var episodeService: EpisodeService
      @Inject lateinit var schedulers: SchedulersBase
      @Inject lateinit var dbRepo: DbRepo

      init{

      }
}
```

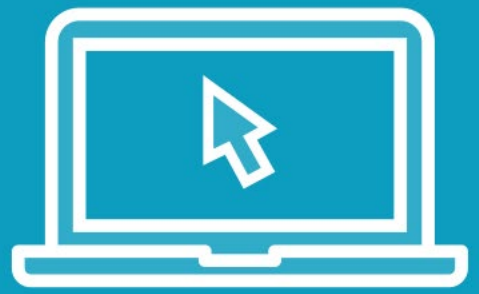
Field Injection

```
class EpisodeListPresenter(context: Context) {

    @Inject lateinit var episodeService: EpisodeService
    @Inject lateinit var schedulers: SchedulersBase
    @Inject lateinit var dbRepo: DbRepo

    init{
        DaggerAppComponent
          .builder()
          .contextModule(ContextModule(context))
          .build()
          .injectPresenter(this)
    }
}
```

Demo



Use of constructor injection using @Inject

Summary

Inject Annotation

- Easier with Dagger
- Constructor Injection
- Field Injection

Component Explanation

- Basic overview

What is a Scope?

Scopes allow us to preserve instances of our dependencies to a component. Once a scoped dependency is initialized, Dagger will reuse that same instance throughout the component.

Scopes are just
name(space)s!

@Singleton is also just a scope!

What A Scope Looks Like

```
@Scope
@Documented
@Retention(RUNTIME)
public @interface Singleton {}
```

What A Scope Looks Like

```
@Scope
@Retention(AnnotationRetention.RUNTIME)
annotation class Singleton
```

What A Scope Looks Like

```
@Scope
@Retention(AnnotationRetention.RUNTIME)
annotation class Pizza
```

How do we use a scope?

Applying A Scope

```
@Module
class EngineModule {

      @Provides
      fun provideEngine(): Engine = Engine()
}
```

Applying A Scope

```
@Module
class EngineModule {

      @Pizza
      @Provides
      fun provideEngine(): Engine = Engine()
}
```

Applying A Scope

```
@Component(modules = [EngineModule::class])
interface AppComponent {

      fun engine(): Engine

}
```

Applying A Scope

```
@Pizza
@Component(modules = [EngineModule::class])
interface AppComponent {

      fun engine(): Engine

}
```

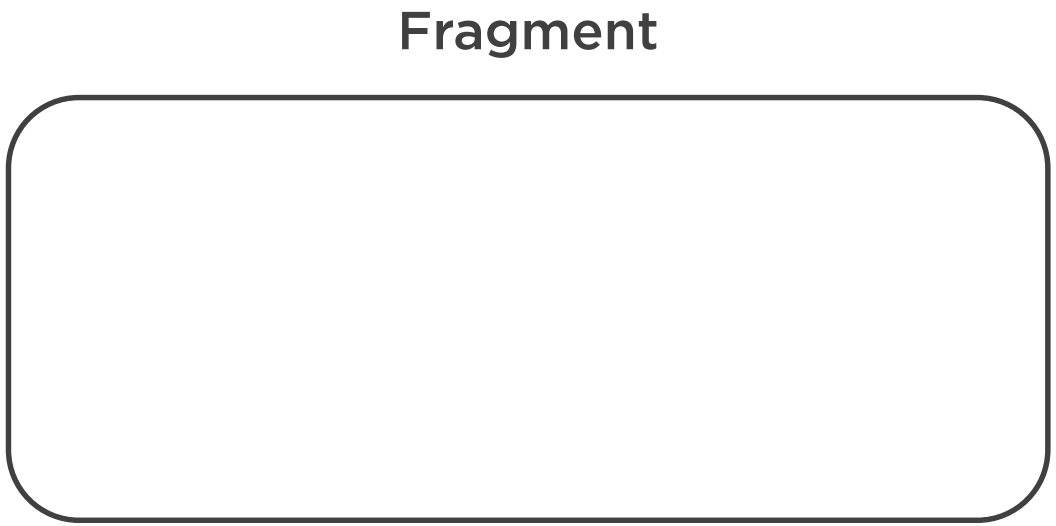
Scopes only work within the same instance of a component!

Summary

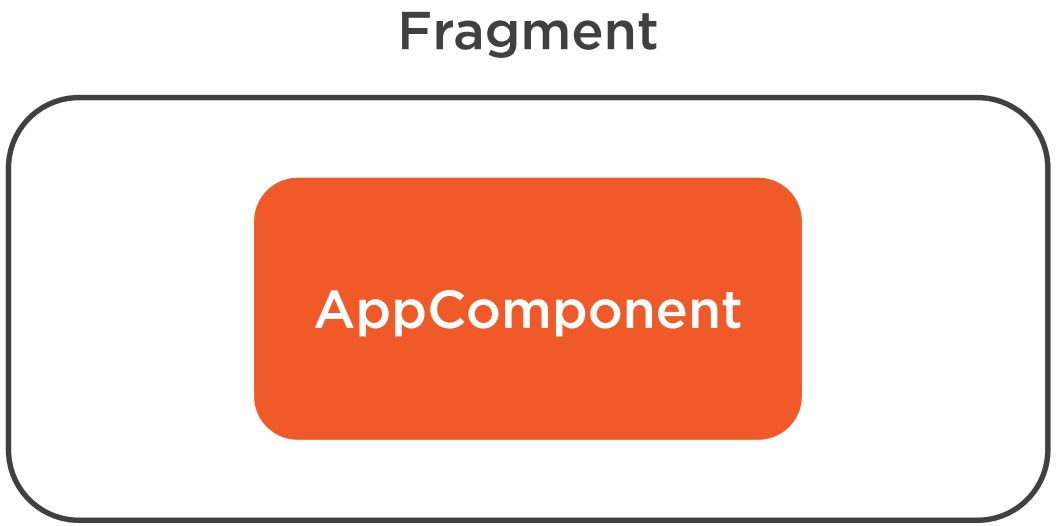
Scopes

- What they are
- How to create one
- How to apply one

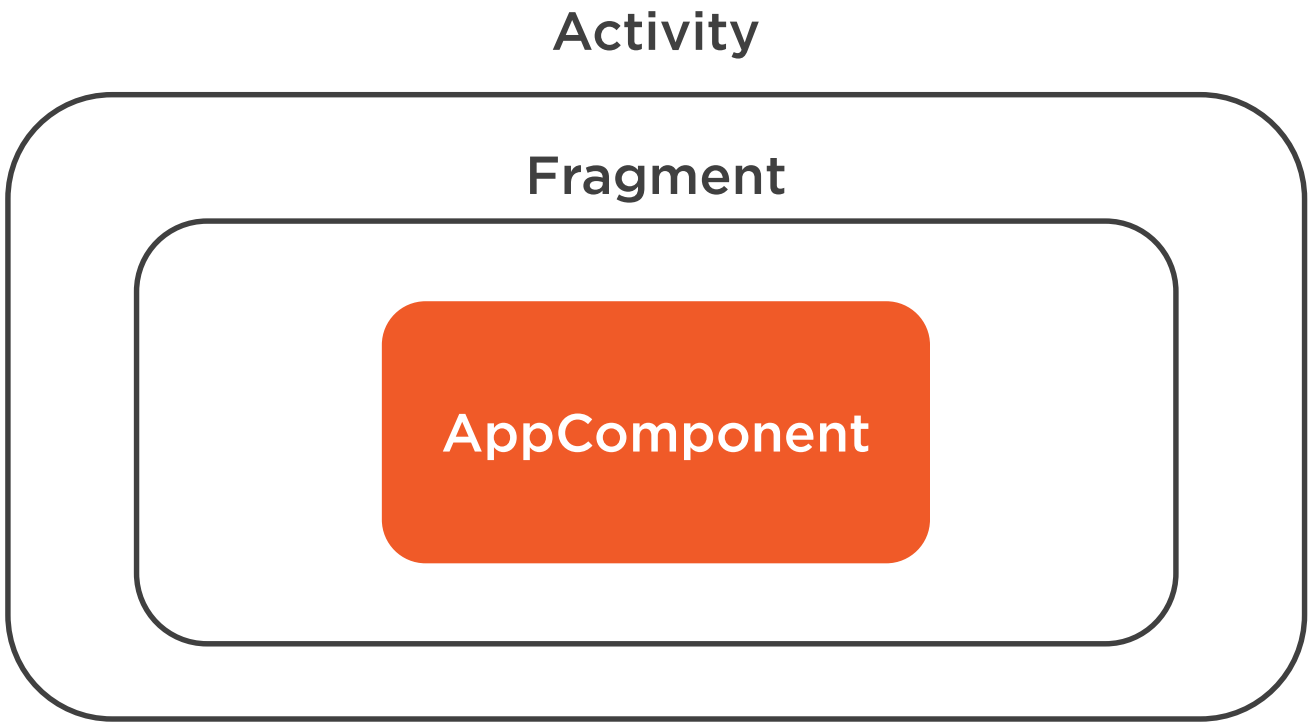
Layering Our Component



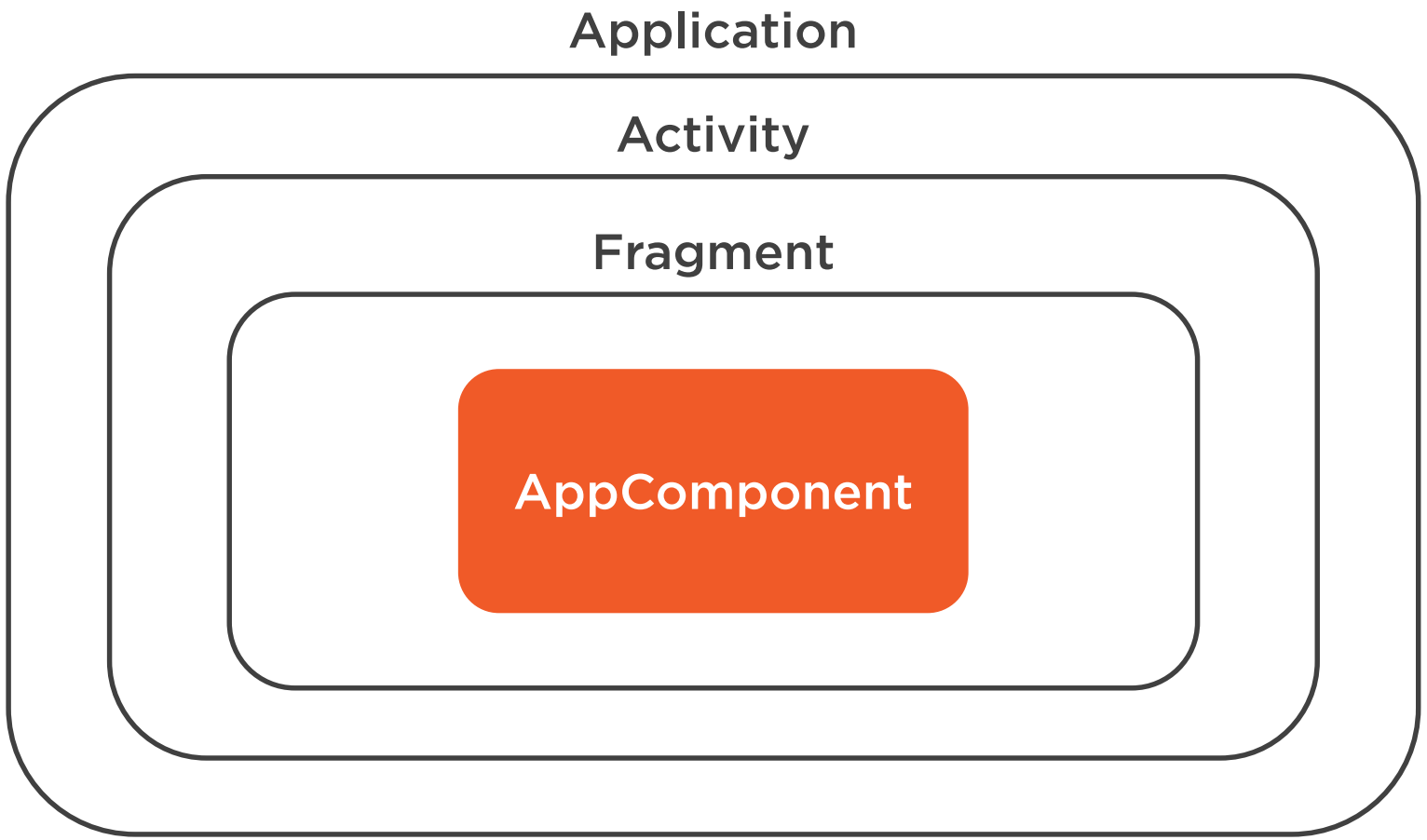
Layering Our Component



Layering Our Component

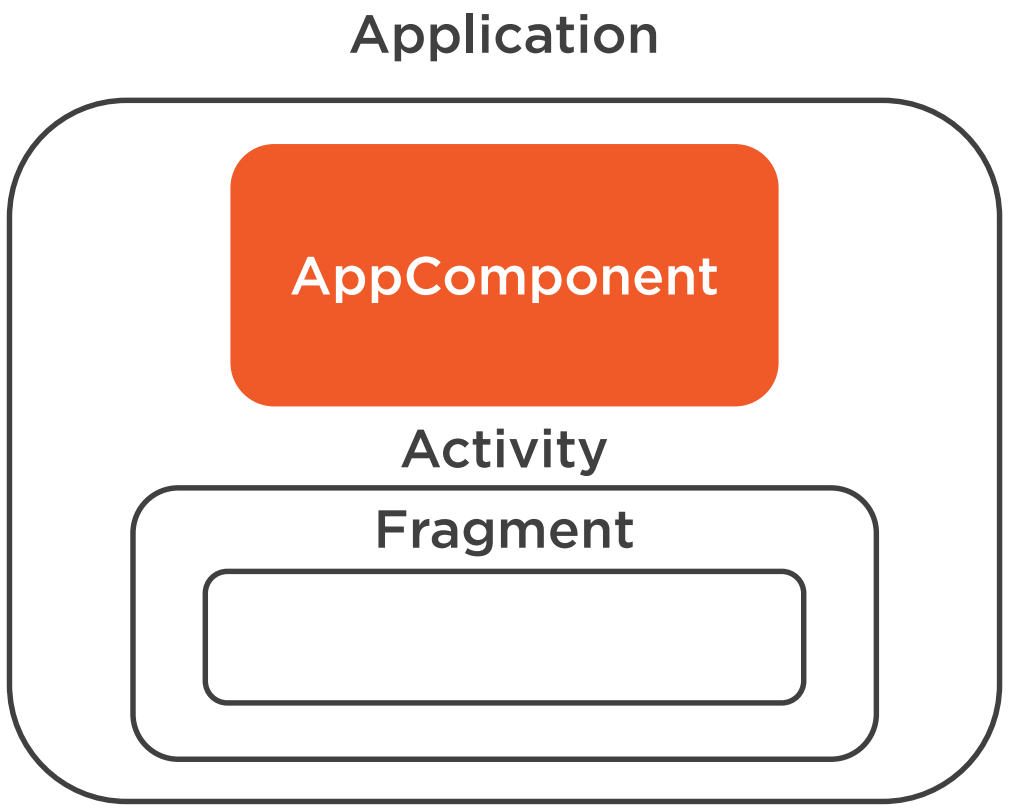


Layering Our Component

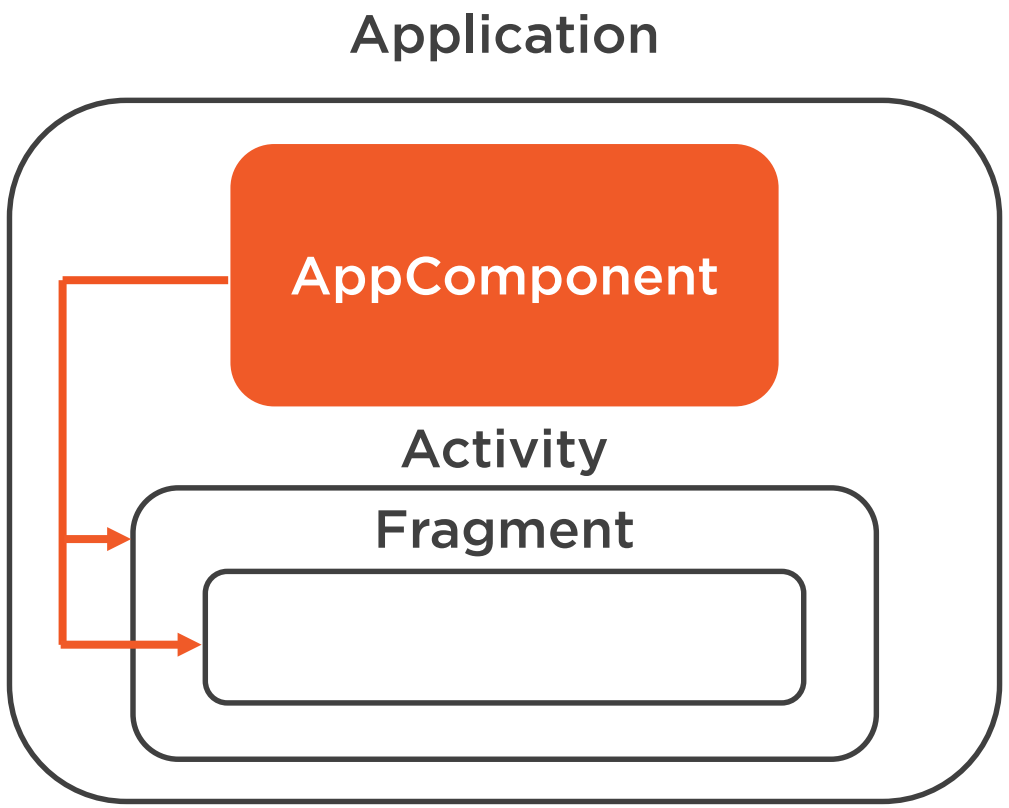


How do we solve this?

Layering Our Component



Layering Our Component

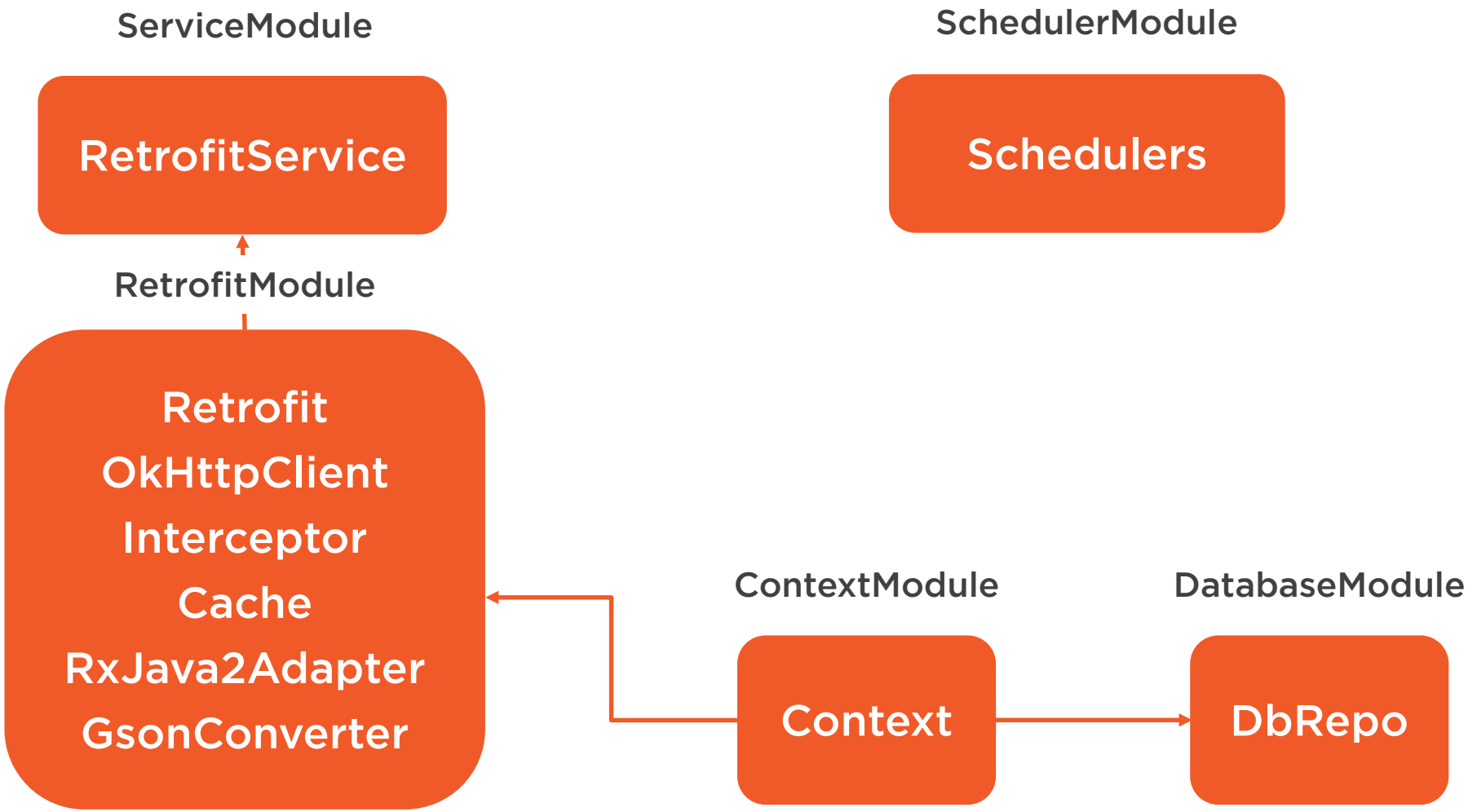


Summary

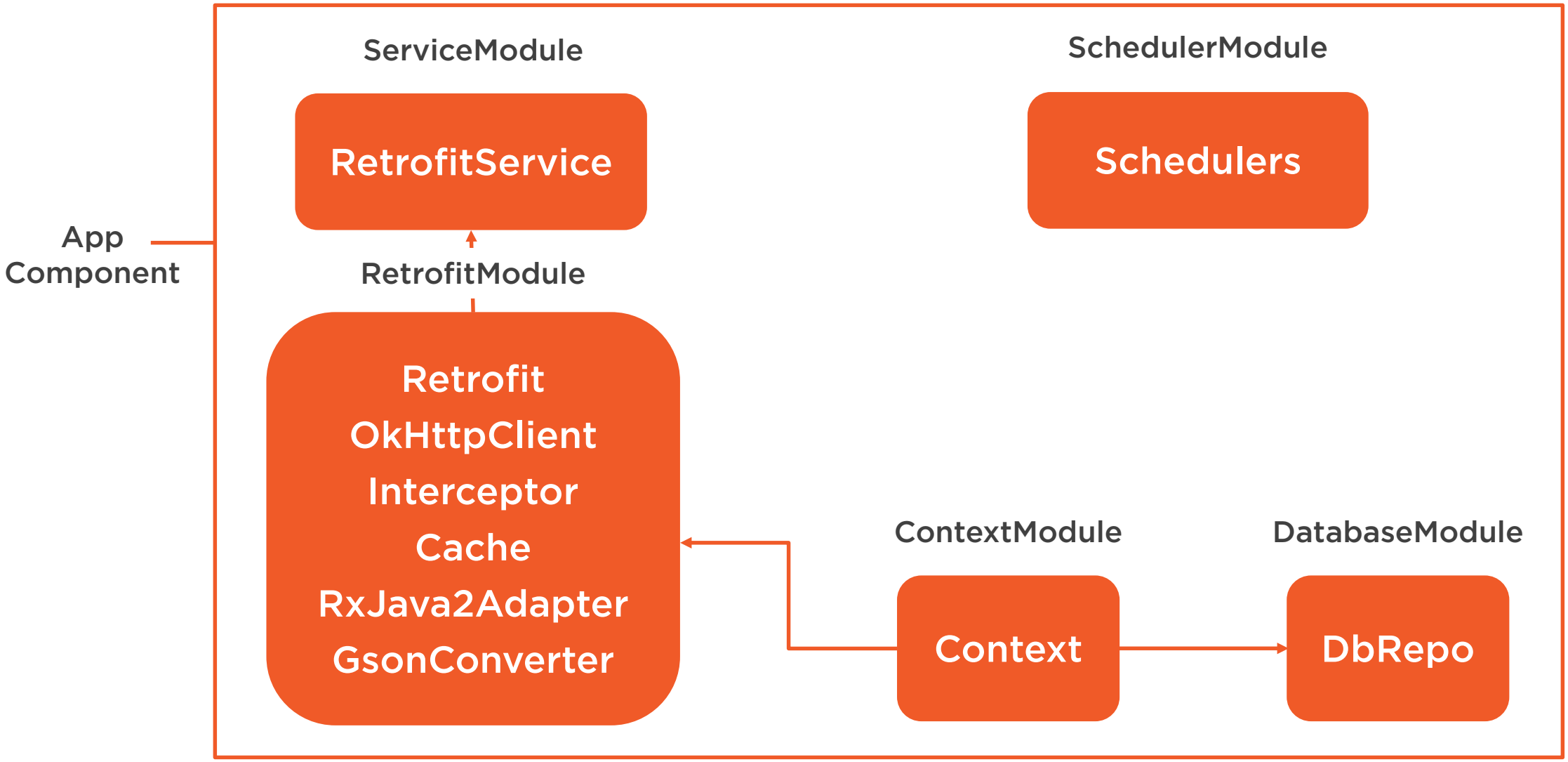
Component layering

- Where to place a component

Our Current Component



Our Current Component

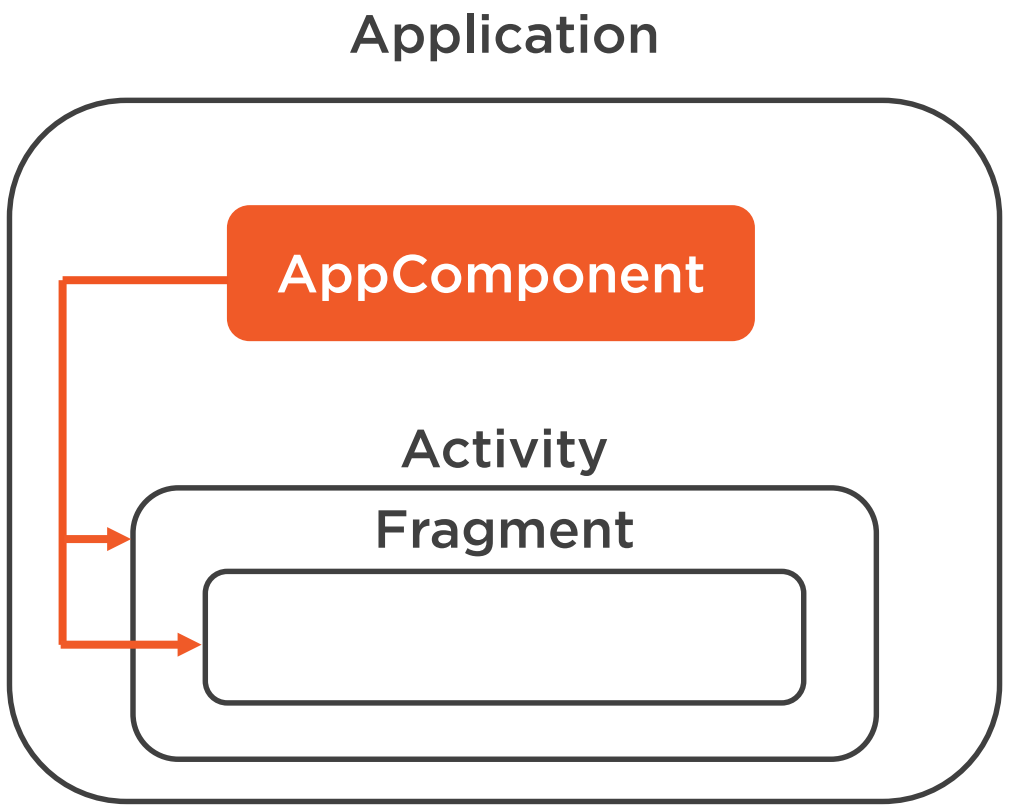


Our Presenter Constructors

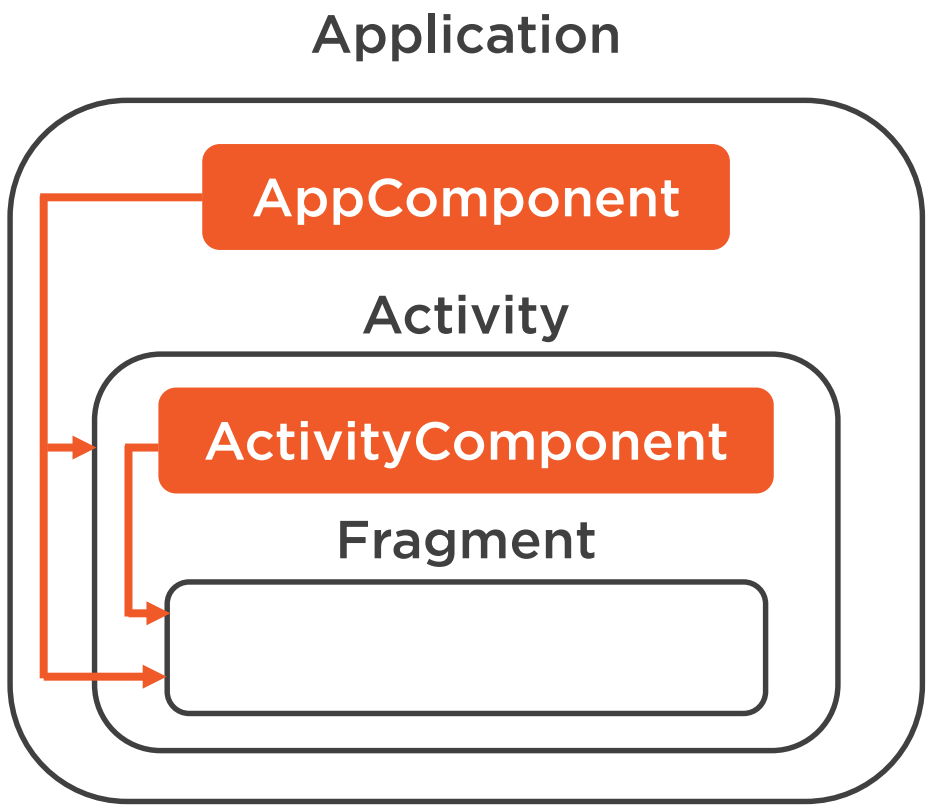
```
class EpisodeDetailPresenter @Inject constructor(
      private val dbRepo: DbRepo,
      private val schedulersBase: SchedulersBase)


class EpisodeListPresenter @Inject constructor(
      private val episodeService: EpisodeService,
      private val schedulers: SchedulersBase,
      private val dbRepo: DbRepo)
```

Layering Our Components

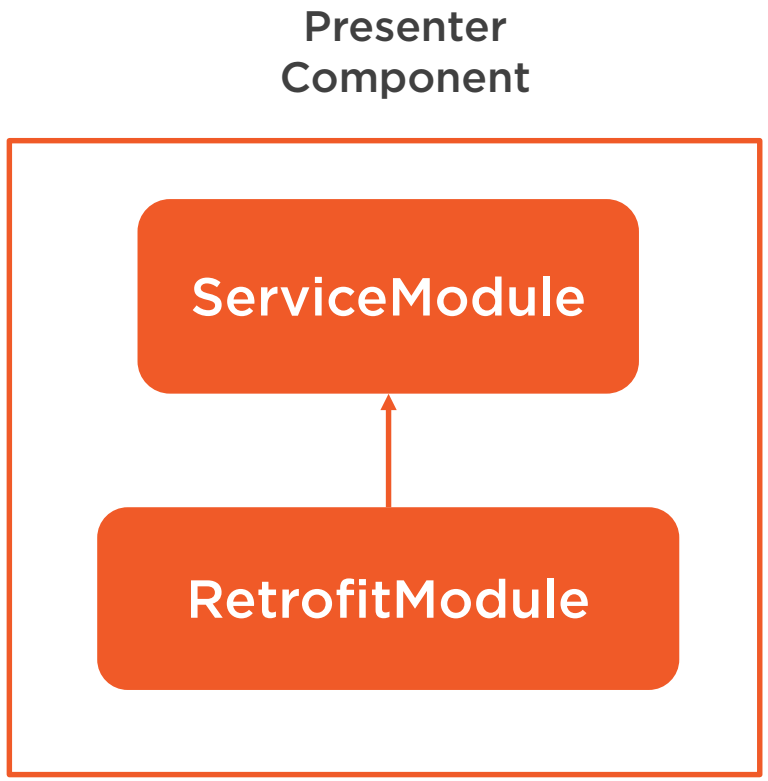


Layering Our Components

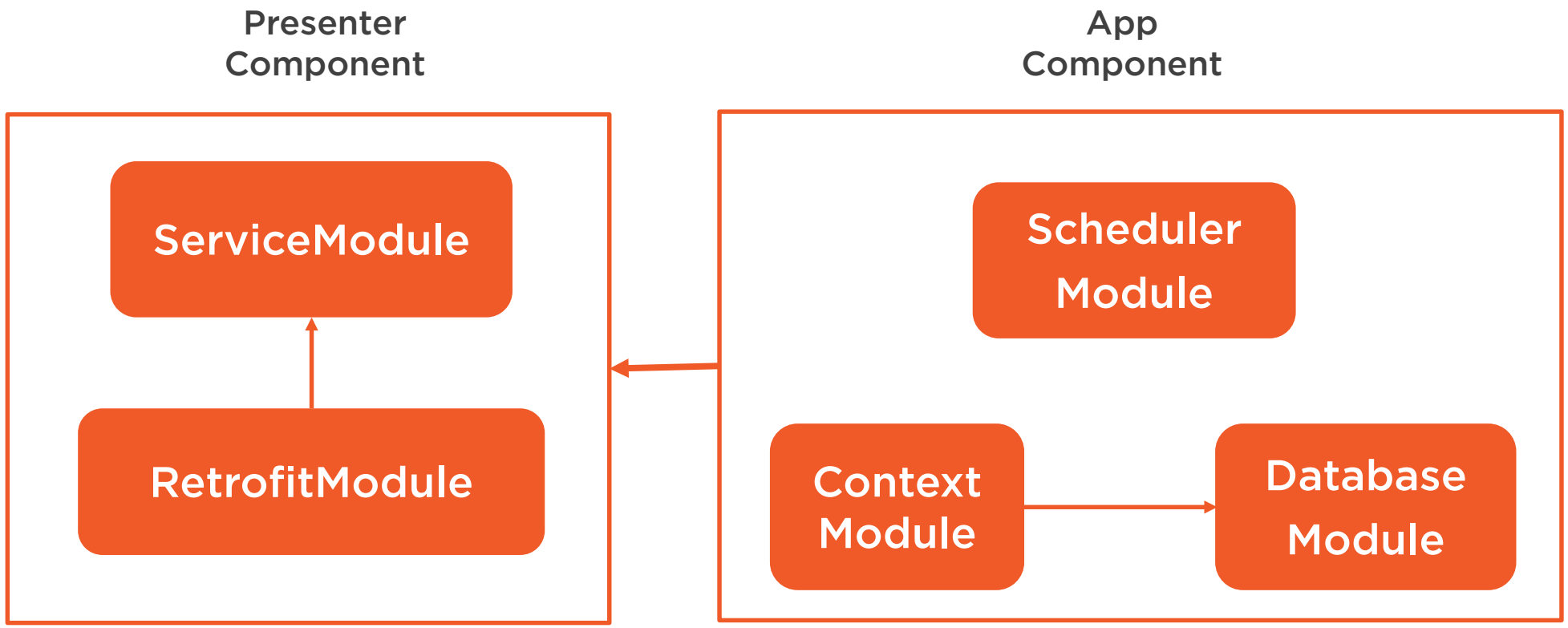


How do create multiple layered components?

Visualizing a Subcomponent



Visualizing a Subcomponent



Summary

Subcomponents

- Importance of layering components
- What they are
- How to create and use one