Chapter 1

Kreator

Parallel to the development of the main project — and in coordination with it —, I have started working on a framework to allow well-integrated dependency injection for kotlin.

Although solutions of varying functionality and quality have existed for years — such as Kodein or KOIN for kotlin, and many others for the jvm in general —, I have found them not to work in accordance of how I would prefer such a library or framework to behave.

1.1 Introduction

Dependency injection in object oriented environments is usually manifested in providing objects to other objects, where the latter use some accessible services of the former. This pattern can be immensely useful in reducing coupling and module dependency, which in turn allow for easier unit testing, better project transparency, and more.

Types of injection

Conventionally, dependency injection is separated into three main types: *constructor-*, *setter-*, and *interface injection*. These cover most of the options commonly used in the java word, but — while using for-java libraries is possible, — due to more powerful language capabilities, solutions in the kotlin ecosystem use different methods. The most important features of kotlin that endorse different patterns are strong immutability support (`val`), simplified 'setter constructor' definition — both incentivizing an in-constructor initialization of dependencies — and first class singleton support (`object`), which allows for easier programmatic injection. Setter- and interface injection is, of course, still possible (through nullable or `lateinit` properties), but result in a reduction of code quality.

Categorization

DI is compromised of two necessary phases: resource declaration and injection. The former is the process in which dependencies are identified for the framework, while injection is the practice of actually providing the requested dependencies, optionally initializing them in the process.

Most available dependency injection solutions work in what I would categorize into one of two ways: declarative- or imperative injection. Declarative injection uses annotations (or in the past, commonly XML configuration) to identify resources, and injects these at marked injection points, such as setters of annotated properties. This kind of injection relies on managed component handling to detect when an injection should occur. On the other hand, imperative injection frameworks require components to be registered at startup to be injectable, and they can be requested via functions.

1.2 Design

Kreator is based on a mixed system where resource identification is done in a declarative way using annotations, while resource injection is implemented in an imperative manner through global functions (`inject*`). This aims to capture the best both words — easy to understand definition and idiomatic injection.

I have focused on supporting constructor injection, because as it allows for easy non-nullable, immutable dependency properties, it is the best type of injection. To achieve this, Kreator is heavily based on kotlin's default method parameters, which allow passing dependencies manually, or falling back to injection. The main design principle is that all components (classes) should take their dependencies as constructor parameters — usually with the type of their provided service interface, and usually using kotlin's unified constructor-property definition construct —, and provide default values for them via the injector functions. It is also an important thing that subtypes of classes shall have constructor parameters for their parents' constructor parameters as well (with default injection) to keep manual dependency passing possible. This is unfortunately easy to forget when extending a component, but is necessary to enable passing dependecies transitively. This design makes kreator a fairly "magic-free" framework, where resource declarations are transparent and injection is cleanly done.

userService.kt

```
open class UserService(
    private val userRepo: UserRepo = inject(),
    private val accountRepo: AccountRepo = inject()
) {
    open fun create(name: String, balance: Int) {...}
}

class RemoteUserService(
    private val provider: ConnectionProvicder = inject(),

    userRepo: UserRepo = inject(),
    accountRepo: AccountRepo = inject()
) : UserService(userRepo, accountRepo) {

    override fun create(name: String, balance: Int) {...}
}
```

Even though dependency injection is the goal of the framework, kreator's injection mechanism is really just resource provision — provided resources are only "injected" to modules because the user connects the providers to the constructors. Because of this, kreator can perhaps more correctly be referred to as a resource management- and provision framework. It is perfectly valid to use injector functions in method bodies or property initializers (or inside constructors), but doing so breaks the possibility of manual dependency passing, effectively hides these used services, and therefore is ill-advised. Of course it sometimes still can be useful, especially when initialization cannot happen by hand anyway, such as in singleton property initializers or in the main function.

Injection function

Injection is completed using three base functions, `inject`, `injectOpt`, and `injectAny`, whose behavior differ slightly. These functions all take advantage of kotlin's (limited) reified generics to provide type-safety. They are designed in a way to be easily extended by building atop of them in a third-party library. All three functions follow the same qualification filtering algorithm, but in the case of some edge cases return different results:

Standard injection Using the standard injection logic, `inject` returns a non-null object, or throws an `InjectionException` if no resource matches the given injection qualifiers, or if multiple resources match the qualifiers with the same precedent.

Optional injection Works like standard injection, except returns `null` if no matching resource can be found.

Any injection Differs from the standard injection in that if the resource qualifiers are ambiguous, one of the matching resources are returned arbitrarily.

Arity

Each resource provider has an assigned arity that describes how it should be used by the injection framework.

- **Per request** The producer should be invoked on every injection request.
- **Singleton** The producer should be invoked at the time of the first request, the produced value should be stored and returned at future injection requests.
- **Singleton autostart** Similar to singleton, but the producer should be invoked at the time of the initialization (resource discovery phase) of the injection framework.

1.3 Injection qualifiers

Injection qualifiers are meta data connected to the injectable resources, which are used to select a resource for a given injection request. Such a request is a call of one of the injection functions. Its selector inputs are the implicitly defined — through generics — injection type and an optional tag value.

Tag

Tags are the most straightforward qualifiers assigned to resources. They are custom strings that are tested based on their equality to the (optional) query tag, and a resource definition can have multiple of them. At injection, the caller may pass a tag with which resources must be marked to be eligible. As these string can have any value, it is up to the programmers to create standard, meaningful tags to be used in a project. Tags should be used as a way of describing different behavior between resources of a type, and therefore often can be grouped by some attribute.

Examples: *file*, *db*, *in–mem*; *direct*, *cached*; *blocking*, *async*; *secure*, *non–secure*

Environment

The environment describes a hierarchical (tree organized) setting, in which a given resource can be used. An environment value is also set for the program at runtime using the `KREATOR_ENV` environmental variable. This qualifier is unique, as it not only filters resources, but also groups them into classes. These classes are looked through in a specific order to find matching resources. The groups are as follows:

As the default environment (`""`) is a parent of all others, if no value is set, all — otherwise eligible — resources can be injected.

Environments are use to define environment dependent behavior, most commonly to create test stubs, fakes, and mocks that should not be used in production. They can also be used to make the program work seamlessly on any host, e.g. on different cloud hosts or operating systems. Environments can also be used to provide implementation with different error processing or logging configurations.

Env. group	Example resources	Lookup
Exact match	`"test.unit"`	First
Sub env.	`"test.unit.junit"`	Second
Sup env.	`"test"`, `""`	Third
Neither	`"dev"`, `"prod.local"`	Never

Groups and examples for the program environment `"test.unit"`

Test resource limitation is achieved in the main project — in part — using environments, as the maven surefire (tester) plugin is configured to execute unit tests under `"test.unit"`.

Default

The default flag is used to select a resource when multiple are eligible for injection. It is a simple boolean value, false by default. Default resources always take precedence over non-default ones.

1.4 Resource declaration

In contrast to some DI frameworks (such as most declarative ones, including java CDI) not all classes that comply with some rules (e.g. having no-arg constructors) are automatically eligible for injection. As more common in programmatic systems, resources must be marked explicitly as injectable — but using annotations. The four annotations associated with resource declaration are `InjectableType`, `Injectable`, `TestInjectable`, and `NotInjectableFor`.

`InjetableType` This annotation marks types that act as a target for injection. It is intended to be used on interfaces that define the behavior of a type of services.

`Injectable`, `TestInjectable` These define resource providers with their respective configuration. They may be placed on no-argument functions, -constructors or classes with such constructors. Their configuration defines the qualifiers of the resource (environment, tags, defaultness), its arity, and their injectable types that define for which types can the resource be provided. By default (if the types are left empty), a resource is injectable for its own type (without is needing to be marked injectable type) and all of its parent injectable types. If types are set explicitly, the resource can only by injected for the set types.

A single provider can have multiple injectable annotations, each with their own set of qualifiers to establish different roles for the resource.

`TestInjectable` is a simple type-safe helper whose only job is to prefix all environments with `"test"`, the most commonly used environment (other than `""`).

`NotInjectableFor` This annotation acts as a disqualifier for specific types. It is to be placed on providers that are annotated with `Injectbale` (or `TestInjectable`) and have a lot of injectable parent types, but are not intended to be injectable for all of them. It allows the type qualifier to stay non-defined (implicitly all injectable supertypes), and to list only a handful of supertypes as not injectable for.

1.5 Property injection

As an independent submodule, kreator contains a property configuration framework that allows for string, integer, and boolean properties to be defined in various sources, such as environment variables, property files, or structured config files. Like the resource injection parts, this module is configured via environmental variables (source type and optionally source location), supports optional and non-optional properties, is type-safe, and provides default no-op sources if configuration is not present.

This system is relied on by the main project for environment dependent configuration of the data source, runtime actor handlers, test games, and test data manager scripts.

Sample config – `tulkas.conf`

```
Client {
    Log {
        base-dir: /tulkas/log/
    }
    Engine {
        script-path: /tulkas/engine-runtime-client/start.sh
        redirect-out: true
    }
    Bot {
        script-path: /tulkas/bot-runtime-client/start.sh
        redirect-out: true
    }
}

Server {
    Database {
        DriverClass: org.hsqldb.jdbc.JDBCDriver
        ConnectionString: jdbc:hsqldb:mem:tulkasDB
        Username: tulkas
        Password: tulkas

        Script {
            Init:  /sql/create.sql
            Fill:  /sql/insert_test_data.sql
            Clear: /sql/clear_tables.sql
            Drop:  /sql/delete.sql
        }
    }
}
```

Example usage – `dbConnection.kt`

```
@InjectableType
interface ConnectionSource {
    operator fun invoke(): Connection
}

@Injectable(
    arity = SINGLETON_AUTOSTART, default = true, tags = ["jdbc"]
)
class JdbcConnectionSource(
    driverClass: String
            = property("Server.Database.DriverClass"),
    private val connectionString: String
            = property("Server.Database.ConnectionString"),
    private val username: String
            = property("Server.Database.Username"),
    private val password: String
            = property("Server.Database.Password")
) : ConnectionSource {
    init {
        Class.forName(driverClass) // Load external class
    }

    override operator fun invoke(): Connection =
        getConnection(connectionString, username, password)
}
```

1.6 Project structure

Kreator is separated into four modules all targeting their respective jar files – *annotation*, *api*, *core*, and *property*.

Annotation contains necessary types for resource declaration, it is a very minimal library that can be used by any frameworks or libraries that wish to provide optional kreator–based injection support. It only contains declarative parts, therefore if a dependency of a program is built on it, but the program itself decides not to use kreator, no disadvantage will happen.

API defines the injetion functions with default no–op implementations, therefore it too can be used in a library without the it affecting all users of that library, only the injections will fail (return null on optional injections).

Core includes the default implementation of the implementation logic defined by *api* and is necessary for injection.

Property defines and implements all the aforementioned property configuration capabilities.