Table of Contents

Contents

Components

Extensible Bot

The `ExtensibleBot` class is the entry point for our framework. It's in charge of managing your extensions, and keeping track of commands.

To get started, you'll need to create an instance of `ExtensibleBot`. You can also subclass it if you need to add or change functionality. For example:

```
package com.kotlindiscord.bot

import com.kotlindiscord.bot.config.config
import com.kotlindiscord.bot.extensions.TestExtension
import com.kotlindiscord.bot.extensions.VerificationExtension
import com.kotlindiscord.kord.extensions.ExtensibleBot

/**
 * The current instance of the bot.
 *
 * It's often better to use named parameters if you need to be specific.
 * Both parameters take the same type, after all!
 */
val bot = ExtensibleBot(prefix = config.prefix, token = config.token)

/**
 * The main function. Every story has a beginning!
 *
 * @param args Array of command-line arguments. These are ignored.
 */
suspend fun main(args: Array<String>) {
    bot.addExtension(TestExtension::class)
    bot.addExtension(VerificationExtension::class)
    bot.start()
}
```

For more information on the public API for `ExtensibleBot`, feel free to take a look at the API docs.

Extensible Bot

Extensible Bot

The `ExtensibleBot` class is the entry point for our framework. It's in charge of managing your extensions , and keeping track of commands.

To get started, you'll need to create an instance of `ExtensibleBot`. You can also subclass it if you need to add or change functionality. For example:

```
package com.kotlindiscord.bot

import com.kotlindiscord.bot.config.config
import com.kotlindiscord.bot.extensions.TestExtension
import com.kotlindiscord.bot.extensions.VerificationExtension
import com.kotlindiscord.kord.extensions.ExtensibleBot

/**
 * The current instance of the bot.
 *
 * It's often better to use named parameters if you need to be specific.
 * Both parameters take the same type, after all!
 */
val bot = ExtensibleBot(prefix = config.prefix, token = config.token)

/**
 * The main function. Every story has a beginning!
 *
 * @param args Array of command-line arguments. These are ignored.
 */
suspend fun main(args: Array<String>) {
    bot.addExtension(TestExtension::class)
    bot.addExtension(VerificationExtension::class)
    bot.start()
}
```

For more information on the public API for `ExtensibleBot`, feel free to take a look at the API docs.

Extensions

Extensions

An `Extension` represents a single unit of functionality, encompassing related commands and event handlers, which can be easily defined using the DSL provided.

All extensions need to subclass the `Extension` class. They'll also need to override two things:

- The `setup` suspended function. This function is where all commands and event handlers should be registered.
- The `name` string. This is a unique identifier for the extension, and extension objects can be retrieved by name if needed.

You may register commands and event handlers using the `command` and `event` DSL functions respectively.

```
class TestExtension(bot: ExtensibleBot) : Extension(bot) {
    override val name: String = "test"

    override suspend fun setup() {
        // Register your commands and event handlers here.
    }
}
```

The `setup` function will be called exactly once, when the first `ReadyEvent` is received after connecting to Discord. This ensures that the `setup` function is able to retrieve objects from Discord.

Commands

Commands

A command is an action invoked by a user on Discord. They're the primary method of interacting with most bots, and require a lot of consideration. We've attempted to make things as easy and friendly to work with as possible.

You can define commands using the DSL functions available in extensions.

Defining Commands

Define your commands in your extension's `setup` function. Here's a basic example:

```
override suspend fun setup() {
    command { // this: Command
        name = "ping"

        action { // this: CommandContext
            message.channel.createMessage("Pong!")
        }
    }
}
```

Basic Options

All commands require a `name` and an `action`, defined similarly to the above example.

- The `name` of your command should be unique across all installed extensions, and is the name that is used to invoke the command on Discord.
- The `action` is the body of the command, which will be executed when a user invokes the command.

In addition to these required options, command support a further set of optional options:

- `aliases: Array<String>` - This is a list of alternative names that can be used to invoke the command. Note that if a command exists with a `name` that exists as an alias in another command, the command with the matching `name` takes priority.

  You can have as many aliases as you like.

- `description: String = """` - This is a description for your command, which will be displayed by the `help` command. Try to make it descriptive, while also keeping it short!

- `enabled: Boolean = true` - If you disable a command by setting this to `false`, it cannot be invoked and will not be shown in the `help` command. This can be modified at runtime, if required.

- `hidden: Boolean = false` - If you want it to be possible to invoke your command, but would still prefer it to be hidden from the `help` command, then you can set this to `true`.

- `signature: String = ""` - The command's signature represents how the arguments to the command should be formatted. For the sake of consistency, all optional arguments should be specified like this: `[name]`. Required arguments should use `<name>`. Lists can be either required or optional, but should contain an ellipsis - `<name ...>`.

> The signature will be shown in help commands, and may also be automatically generated from a data class. See the below section on command arguments for more information on signature generation.

Checks

You will likely want to write commands that are only invoked based on a set of conditions. In order to facilitate this in a manner that allows you to easily reuse these conditions, commands support a checks system.

```
suspend fun defaultCheck(event: MessageCreateEvent): Boolean {
    with(event) {
        return when {
            message.author?.id == bot.kord.getSelf().id -> false  // Check that we didn't
            message.author?.isBot == true               -> false  // Check that another b
            else                                        -> true
        }
    }
}

// ...

command {
    name = "ping"

    check(::defaultCheck)

    action {
        message.channel.createMessage("Pong!")
    }
}
```

If you don't need to make use of the check elsewhere, you can also specify them using a DSL syntax.

```
command {
    name = "ping"

    check {
        it.message.author?.id != bot.kord.getSelf().id &&  // Check that we didn't send t
        it.message.author?.isBot == false  // Check that another bot didn't send this mes
    }

    action {
        message.channel.createMessage("Pong!")
    }
}
```

You can have as many checks as you need - simply call `check` again for every check you wish to add. If you have multiple check functions to add, you may also specify them all as arguments for a single `check` call.

```
check(
    ::defaultCheck,  // Default checks we do for every command
    notChannelType(ChannelType.DM)  // Ensure the command isn't being invoked in a DM
)
```

Command Arguments

In order to make life as easy as possible, you have the option of creating a data class that represents the arguments for your command, allowing for automatic generation of the command signature and parsing of command arguments. For example:

```
data class SampleArgs(
    val optionalInt: Int = 1,  // An optional integer.
    val requiredUsers: List<User>  // A list of User objects.
)

command {
    name = "sample"

    signature<SampleArgs>()  // Automatically generate the command signature.

    action {
        val parsed = parse<SampleArgs>()  // Automatically parse the command arguments ir

        with(parsed) {  // this: SampleArgs
            message.channel.createMessage(
                "Arguments: optionalInt = $optionalInt | " +
                    "requiredUsers = ${requiredUsers.joinToString(", ") { it.username }}'
            )
        }
    }
}
```

In order to facilitate this manner of parsing, the parser follows these rules:

- First, split the message into arguments. Spaces separate arguments from each other, but users may surround a set of arguments with double quotes in order to have them treated as a single argument, preserving the spaces.
- For each parameter in the data class' primary constructor (not the properties defined in the body, if any): Take an argument and try to convert it to the correct type.
    - If the parameter is a List, attempt to convert all remaining arguments for the command.
    - If the argument could not be converted, but you gave the parameter a default value, skip it and try the next parameter with the same argument.
    - If the argument could not be converted and does not have a default value, the user will be presented with an error message.
- Once all the parameters have been converted, the data class is constructed using them. This is then returned from the `parse` function.

We also support an alternative syntax for specifying command parameters. If a parameter contains a colon, it's split into a key-value pair:

- The dataclass is searched for a parameter matching the key on the left side of the colon.
- If the dataclass contains a matching parameter, we convert the value on the right as normal and store it.
    - If the parameter is a list type, the value will be stored within the list. This means that multiple arguments can be specified this way, and they'll all be inserted into the same list.
    - If not, it'll be stored directly within the parameter.

The parser supports the following types:

- Basic types: `String`, `Regex`, `Boolean`, `Int`, `Short`, `Long`, `Float`, `Double`, `BigDecimal`, `BigInteger`.
- Kord types: `Channel`, `Guild`, `GuildEmoji`, `Role`, `User`.

Note: `GuildEmoji` and `Role` objects will be retrieved from the within the current guild. It is not possible to specify which guild to use at this time, but we may add support for that later.

Event Handlers

Event Handlers

Event handlers function similarly to commands, but they react to Kord events instead of command invocations on Discord. For a full list of Kord events, you can check out the Kord GitLab - but for the purposes of this page, we'll just be looking at a `MessageCreateEvent` - an event that is fired when someone sends a message.

Defining Event Handlers

Define your event handlers in your extension's `setup` function. Here's a basic example:

```
override suspend fun setup() {
    event<MessageCreateEvent> {  // this: EventHandler
        action {  // it: MessageCreateEvent
            with(it) {
                message.channel.createMessage("Thanks for your message, ${message.author
            }
        }
    }
}
```

Options

All event handlers require an `action`. The `action` is the body of the event handler - when Kord fires the corresponding event, this will be executed.

Additionally, just like commands, event handlers support checks.

Checks

You will likely want to write event handlers that are only executed based on a set of conditions. In order to facilitate this in a manner that allows you to easily reuse these conditions, event handlers support a checks system.

```
suspend fun defaultCheck(event: MessageCreateEvent): Boolean {
    with(event) {
        return when {
            message.author?.id == bot.kord.getSelf().id -> false  // Check that we didn't
            message.author?.isBot == true               -> false  // Check that another b
            else                                        -> true
        }
    }
}

// ...

event<MessageCreateEvent> {
    check(::defaultCheck)

    action {
        with(it) {
            message.channel.createMessage("Thanks for your message, ${message.author!!.me
        }
    }
}
```

You can have as many checks as you need - simply call `check` again for every check you wish to add. If you have multiple check functions to add, you may also specify them all as arguments for a single `check` call.

```
check(
    ::defaultCheck,  // Default checks we do for every event
    notChannelType(ChannelType.DM)  // Ensure the event isn't being fired for a DM
)
```

Note: Most of the bundled checks only support `MessageCreateEvent` events. If you need to support other events, feel free to write your own checks and submit them in a pull request!

Checks

Checks

The checks system provides a light filtering system for commands and event handlers. We've provided a set of general-purpose checks as part of the framework, to cover most general cases - but you're always able to write your own as well.

Writing Your Own Checks

While we've already covered writing custom checks inline in your commands and event handlers, often it'll be more useful to define these checks somewhere that they can be re-used.

Every check is simply a suspended function that takes a Kord event object and returns a `Boolean`. If the function returns `true` then the check has passed and processing can continue. If it returns `false` then the check has failed and processing stops at that point.

You can define your check as a simple function. For example:

```
suspend fun isNotBot(event: MessageCreateEvent): Boolean {
    with(event) {  // Make sure the message wasn't created by a bot.
        return message.author?.isBot == false
    }
}

suspend fun isNotDM(event: MessageCreateEvent): Boolean {
    with(event) {  // Make sure the message wasn't sent as part of a DM.
        return message.channel.asChannel().type != ChannelType.DM
    }
}
```

These checks can be passed directly to the `check` function in your command (if it operates on MessageCreateEvent objects) or event handler.

```
command {
    check(::isNotBot, ::isNotDM)
}
```

This is already pretty useful, but we could make our checks more wide-reaching.

Checks with parameters

In order to write a check which may need to match against another object, we can create a builder function. Let's look at `notChannelType`, which is a check provided with this framework.

```
fun notChannelType(channelType: ChannelType): suspend (MessageCreateEvent) -> Boolean {
    suspend fun inner(event: MessageCreateEvent): Boolean {
        with(event) {
            return message.channel.asChannel().type != channelType
        }
    }

    return ::inner
}
```

This function creates a closure around an inner function, and returns that inner function. We can make use of it like this:

```
event<MessageCreateEvent> {
    check(notChannelType(ChannelType.DM))
}
```

This allows us to create far more useful checks that can operate across a variety of options.

Generic checks

In some cases, you'll want to write checks that can deal with multiple types of event. In order to facilitate this, we provide a `CheckUtils` module containing a set of functions that can return Kord Behaviors for a given event. For example, a check written to support events that concern specific members might look like this:

```
fun hasRole(role: Role): suspend (Event) -> Boolean {
    suspend fun inner(event: Event): Boolean {
        val member = memberFor(event) ?: return false

        return member.asMember().roles.toList().contains(role)
    }

    return ::inner
}
```

We provide the following functions:

- `channelFor(event: Event) -> ChannelBehavior?`
- `guildFor(event: Event): GuildBehavior?`
- `memberFor(event: Event): MemberBehavior?`
- `messageFor(event: Event): MessageBehavior?`
- `roleFor(event: Event): RoleBehavior?`
- `userFor(event: Event): UserBehavior?`

These functions will return `null` for an unsupported event type. All the checks bundled with this framework support generic events - if a check receives an unsupported type, that check will simply fail. These failures will be logged at debug level, so enable SLF4J debug logging if you're not sure - or just use a debugger, I'm not your mom.

Combinators

If you don't want to simply require that all checks pass for a command or event handler, you can use a combinator check. We currently bundle two checks: `or` and `and`.

```
event<MessageCreateEvent> {
    check(or(  // Check that the channel is either...
        channelType(ChannelType.DM)  // A DM channel,
        channelType(ChannelType.GuildNews)  // Or a news channel.
    ))
}
```

Bundled Checks

For a full list of bundled checks, please take a look at the API documentation for the checks package.

Examples

Full Command Example

The below is a sample extension implementing a command, which makes use of and explains all the options available to you.

```
package com.kotlindiscord.kord.extensions.samples

import com.gitlab.kordlib.common.entity.ChannelType
import com.gitlab.kordlib.core.entity.User
import com.kotlindiscord.kord.extensions.ExtensibleBot
import com.kotlindiscord.kord.extensions.checks.notChannelType
import com.kotlindiscord.kord.extensions.extensions.Extension

class CommandFullSample(bot: ExtensibleBot) : Extension(bot) {
    override val name: String = "command-action-sample"

    override suspend fun setup() {
        /**
         * A data class representing the arguments for the command we'll define below.
         *
         * You don't have to define this in the setup function, it's just placed here fo
         * example.
         *
         * Arguments are attempted to be filled in the order they're defined in the data
         * primary constructor. If an argument has a default value then it's considered
         * so it will be skipped if type conversion fails.
         *
         * Typed [List] arguments are filled using the rest of the command's arguments,
         * of the time you'll only want to provide a list as the last argument for a com
         *
         * For more information on how this works, please check out the examples in the
         */
        data class SampleArgs(
            val optionalInt: Int = 1,  // An optional integer
            val requiredUsers: List<User>  // A list of User objects, to be converted fr
                                           // or mentions
        )

        val sampleCommand = command {  // You can optionally store the command object yo
                                       // if you need it elsewhere.
            aliases = arrayOf("test", "hello")  // Alternative names that can be used fo
                                                // command invocation.
            description = "A sample command. Outputs `Hello, world!'."  // Description f
                                                                        // command.
            enabled = true  // Whether the command is enabled; this can be changed at ru
                            // as well.
            hidden = false  // Whether to hide the command from help command listings.
            name = "sample"  // The name of the command.

            signature<SampleArgs>()  // Generate the signature from the given dataclass.
            signature = "[optionalInt] <requiredUsers ...>"  // Manually set the command
                                                             // signature instead.

            check(notChannelType(ChannelType.DM))  // Check that the message wasn't sent
                                                   // in a DM.
            check {
                // Check that the message wasn't sent by a bot.
                it.message.author?.isBot == false
            }

            action { // The body of the command, executed assuming the checks all pass.
                with(parse<SampleArgs>()) {  // Automatically parse command arguments in
```

```
                                // given data class.
                    message.channel.createMessage("Hello, world!")

                    // Since we're in a `with` block, we can directly use the parameters
                    // the data class.
                    message.channel.createMessage(
                        "Arguments: optionalInt = $optionalInt | " +
                            "requiredUsers = ${requiredUsers.joinToString(", ") { it.user
                    )
                }
            }
        }
    }
}
```

Full Command Example

Full Command Example

The below is a sample extension implementing a command, which makes use of and explains all the options available to you.

```
package com.kotlindiscord.kord.extensions.samples

import com.gitlab.kordlib.common.entity.ChannelType
import com.gitlab.kordlib.core.entity.User
import com.kotlindiscord.kord.extensions.ExtensibleBot
import com.kotlindiscord.kord.extensions.checks.notChannelType
import com.kotlindiscord.kord.extensions.extensions.Extension

class CommandFullSample(bot: ExtensibleBot) : Extension(bot) {
    override val name: String = "command-action-sample"

    override suspend fun setup() {
        /**
         * A data class representing the arguments for the command we'll define below.
         *
         * You don't have to define this in the setup function, it's just placed here fo
         * example.
         *
         * Arguments are attempted to be filled in the order they're defined in the data
         * primary constructor. If an argument has a default value then it's considered
         * so it will be skipped if type conversion fails.
         *
         * Typed [List] arguments are filled using the rest of the command's arguments,
         * of the time you'll only want to provide a list as the last argument for a com
         *
         * For more information on how this works, please check out the examples in the
         */
        data class SampleArgs(
            val optionalInt: Int = 1,  // An optional integer
            val requiredUsers: List<User>  // A list of User objects, to be converted fr
                                           // or mentions
        )

        val sampleCommand = command {  // You can optionally store the command object yo
                                       // if you need it elsewhere.
            aliases = arrayOf("test", "hello")  // Alternative names that can be used fo
                                                // command invocation.
            description = "A sample command. Outputs `Hello, world!'."  // Description f
                                                                        // command.
            enabled = true  // Whether the command is enabled; this can be changed at ru
                            // as well.
            hidden = false  // Whether to hide the command from help command listings.
            name = "sample"  // The name of the command.

            signature<SampleArgs>()  // Generate the signature from the given dataclass.
            signature = "[optionalInt] <requiredUsers ...>"  // Manually set the command
                                                             // signature instead.

            check(notChannelType(ChannelType.DM))  // Check that the message wasn't sent
                                                   // in a DM.
            check {
                // Check that the message wasn't sent by a bot.
                it.message.author?.isBot == false
            }

            action { // The body of the command, executed assuming the checks all pass.
                with(parse<SampleArgs>()) {  // Automatically parse command arguments in
```

```
                                // given data class.
                    message.channel.createMessage("Hello, world!")

                    // Since we're in a `with` block, we can directly use the parameters
                    // the data class.
                    message.channel.createMessage(
                        "Arguments: optionalInt = $optionalInt | " +
                            "requiredUsers = ${requiredUsers.joinToString(", ") { it.user
                    )
                }
            }
        }
    }
}
```

Full Event Handler Example

Full Event Handler Example

The below is a sample extension implementing an event handler, which makes use of and explains all the options available to you.

```
package com.kotlindiscord.kord.extensions.samples

import com.gitlab.kordlib.common.entity.ChannelType
import com.gitlab.kordlib.core.event.message.MessageCreateEvent
import com.kotlindiscord.kord.extensions.ExtensibleBot
import com.kotlindiscord.kord.extensions.checks.notChannelType
import com.kotlindiscord.kord.extensions.extensions.Extension

class EventHandlerSample(bot: ExtensibleBot) : Extension(bot) {
    override val name: String = "event-handler-sample"

    override suspend fun setup() {
        val eventHandler = event<MessageCreateEvent> {  // Define an event handler, optic
            // This one listens for new messages.
            check(notChannelType(ChannelType.DM))  // Check that the message wasn't sent
            // in a DM.
            check {
                // Check that the message wasn't sent by a bot.
                it.message.author?.isBot == false
            }

            check {
                it.message.author != null  // Ensure the message has an author (webhooks
            }

            action {  // The body of the event handler, executed assuming the checks all
                with(it) {  // The event object
                    // We know the author isn't null because of our check earlier!
                    message.channel.createMessage("Thanks for your message, ${message.aut
                }
            }
        }
    }
}
```