



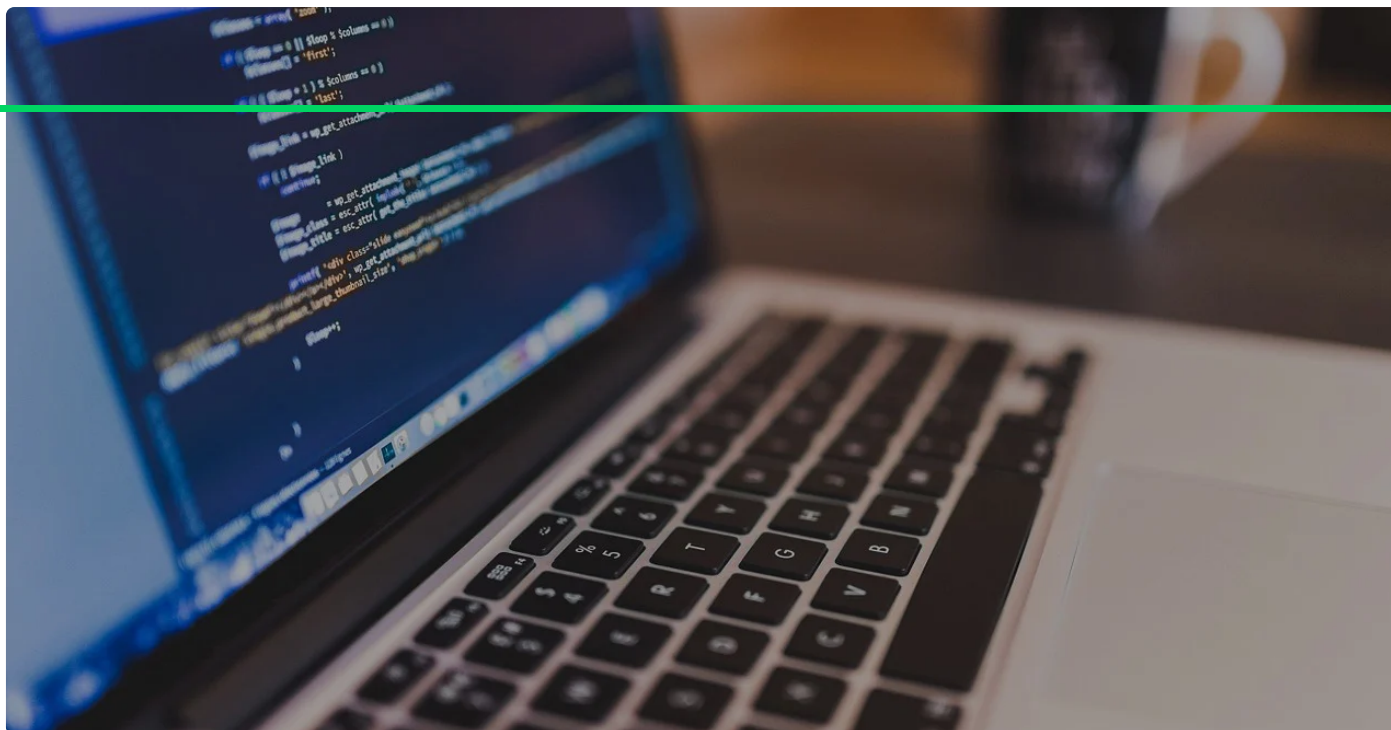
How to Handle Resources in Kotlin Multiplatform



Krzysztof Kansy

Aug 23, 2022 • 9 min read





When we work with a user-facing application we would like to use resources in our application like colors, texts, images, etc.

Currently, Kotlin Multiplatform does not provide a resource management solution as it is in the case of native Android applications. Because of it, we want to provide you with a guide with recommendations on how to achieve a good, scalable solution for app resources.

In this article, I'll try to demonstrate some of the solutions we've discovered for some key [Kotlin multiplatform resources](#).

Configuration for Android

In most cases, the second solution for Android wouldn't work without the line below. Place this block of code in `build.gradle.kts` in the module where you're planning to keep resources.

```
android {  
    sourceSets["main"].apply {  
        res.srcDirs("src/androidMain/res", "src/commonMain/resources")  
    }  
}
```

Colors

The Material library offers the color schemes `lightColors` and `darkColors`. You should create your own data class for colors only if you have a custom palette that doesn't follow those from the Material library. Here's an example of how you can use custom colors in your app.

```
@Immutable  
data class Colors(  
    ...  
)
```

```
    val primary: Color,  
    val onPrimary: Color  
)  
  
private val LightColors = Colors(  
    primary = Color(0xFF3D7FF9),  
    onPrimary = Color(0xFFFFFFFF)  
)  
  
@Composable  
fun ApplicationTheme(content: @Composable () -> Unit) {  
    MaterialTheme(  
        colors = MaterialTheme.colors.copy(  
            primary = LightColors.primary,  
            onPrimary = LightColors.onPrimary  
        ),  
        content = content  
    )  
}
```

Dimension

The approach for the dimension resource is similar to the solution above for colors. Look at the example below.

```
@Immutable
data class Dimens(val margin: Dp)

val smallDimens = Dimens(margin = 10.dp)

val mediumDimens = Dimens(margin = 12.dp)

val largeDimens = Dimens(margin = 14.dp)

val LocalDimens = staticCompositionLocalOf { smallDimens }

@Composable
fun ApplicationTheme(windowSize: WindowSize, content: @Composable () -> Unit) {
    val dims = when (windowSize) {
        WindowSize.COMPACT -> smallDimens
        WindowSize.MEDIUM -> mediumDimens
        WindowSize.EXPANDED -> largeDimens
    }
    CompositionLocalProvider(LocalDimens provides dims) {
        content()
    }
}

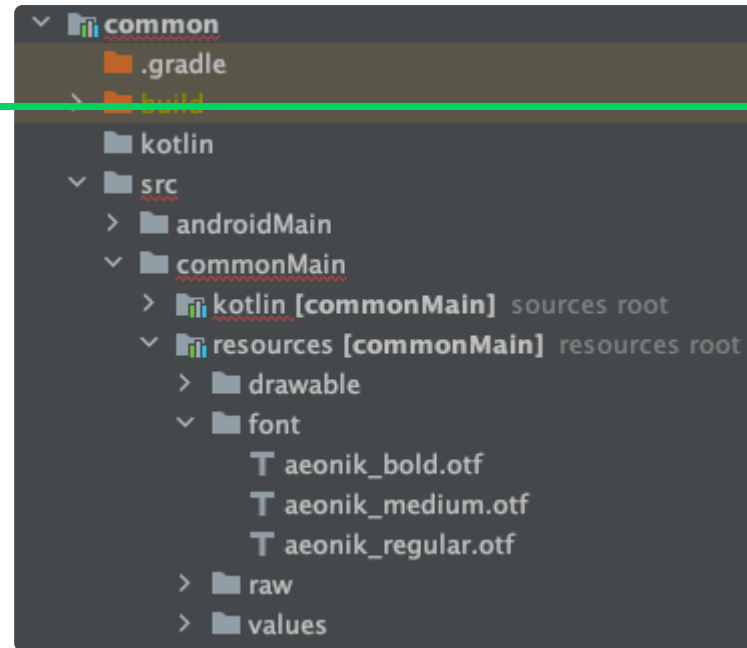
@Composable
fun ApplicationScreen() {
    ApplicationTheme(WindowSize.COMPACT) {
        Column(Modifier.padding(LocalDimens.current.margin)) {
```

```
}  
}  
}
```

Fonts

You can read fonts from the file by using the `fontResources` method. You'll need to create `FontFamily` and assign it to `defaultFontFamily`. Then, you just need to provide your custom font to the theme.





Example of Fonts usage

The code below shows a simple implementation for using custom fonts.

@Composable

```
private fun CustomTypography() = Typography(  
    defaultFontFamily = FontFamily(  
        fontResources("aeonik_regular.otf", FontWeight.Normal, FontStyle.Normal),  
        fontResources("aeonik_medium.otf", FontWeight.W500, FontStyle.Normal),  
        fontResources("aeonik_bold.otf", FontWeight.Bold, FontStyle.Normal)
```

```
)  
)  
  
@Composable  
fun ApplicationTheme(content: @Composable () -> Unit) {  
    MaterialTheme(  
        typography = CustomTypography(),  
        content = content  
    )  
}
```

FontResources Implementation

Every platform has a different mechanism to retrieve fonts from its files. Thanks to the `expect / actual` feature, you can use specific implementations.

commonMain/kotlin/FontResources

```
import androidx.compose.runtime.Composable  
import androidx.compose.ui.text.font.Font  
import androidx.compose.ui.text.font.FontStyle  
import androidx.compose.ui.text.font.FontWeight
```

omposable


```
expect fun fontResources(  
    font: String,  
    weight: FontWeight,  
    style: FontStyle  
): Font
```

androidMain/kotlin/FontResources

```
@Composable  
actual fun fontResources(  
    font: String,  
    weight: FontWeight,  
    style: FontStyle  
): Font {  
    val context = LocalContext.current  
    val name = font.substringBefore(".")  
    val fontRes = context.resources.getIdentifier(name, "font", context.packageName)  
    return Font(fontRes, weight, style)  
}
```

desktopMain/kotlin/FontResources

```
import androidx.compose.ui.text.font.Font  
import androidx.compose.ui.text.font.FontStyle  
import androidx.compose.ui.text.font.FontWeight  
import androidx.compose.ui.text.platform.Font
```

```
actual fun fontResources(  
    font: String,  
    weight: FontWeight,  
    style: FontStyle  
): Font = Font("font/$font", weight, style)
```

Strings

There's an issue of how to handle translations when it comes to Strings. We found two ways in handling strings translations: with and without automation.

First solution: Without automation

This solution is quite simple but it requires implementing everything manually.

```
@Immutable  
data class StringResources(  
    val appBackground: String = ""  
)  
  
stringResourcesEn() = StringResources(  

```

```
    appBackground = "App background"
)

fun stringResourcesDe() = StringResources(
    appBackground ="App hintergrund"
)

val LocalStringResources = staticCompositionLocalOf { stringResourcesEn() }

@Composable
fun ApplicationTheme(content: @Composable () -> Unit) {
    val stringResources = when (Locale.current.language) {
        "DE" -> stringResourcesDe()
        else -> stringResourcesEn()
    }
    CompositionLocalProvider(LocalStringResources provides stringResources) {
        content()
    }
}

@Composable
fun ApplicationScreen() {
    ApplicationTheme {
        Text(text = LocalStringResources.current.app_background)
    }
}
```

Second solution: With automation

The second way will require you to use the tool that defines strings in traditional .xml. Then, by using Gradle, this can automatically generate all required classes with useful extensions.

- The first step is to create .xml files with strings and put them in the appropriate directories for the language (e.g. values-de, values-en).

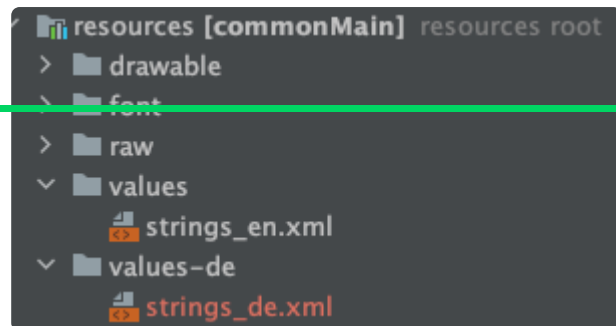
resources/values/strings_en.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<resources>
    <string name="app_background">App background</string>
</resources>
```

resources/values-de/strings_de.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<resources>
    <string name="app_background">App hintergrund</string>
</resources>
```

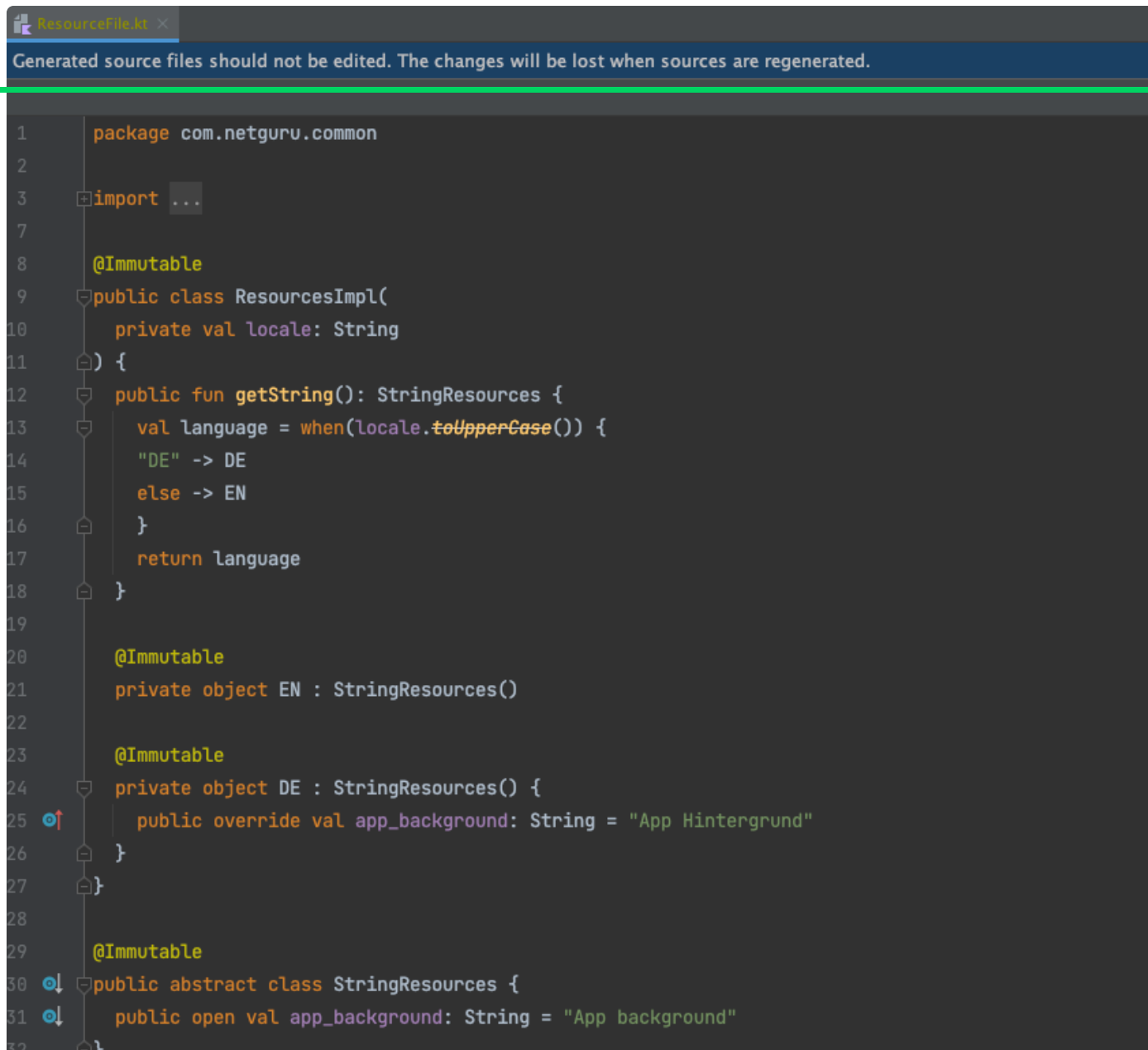




Strings location

- The next step is to run the `resourceGeneratorTask`.





```
1 package com.netguru.common
2
3 import ...
4
5
6
7
8 @Immutable
9 public class ResourcesImpl(
10     private val locale: String
11 ) {
12     public fun getString(): StringResources {
13         val language = when(locale.toUpperCase()) {
14             "DE" -> DE
15             else -> EN
16         }
17         return language
18     }
19
20     @Immutable
21     private object EN : StringResources()
22
23     @Immutable
24     private object DE : StringResources() {
25         public override val app_background: String = "App Hintergrund"
26     }
27 }
28
29 @Immutable
30 public abstract class StringResources {
31     public open val app_background: String = "App background"
32 }
```

The example of the general

- Lastly, we need to

```

13
14 @ReadOnlyComposable
15 @Composable
16 public fun getString(): StringResources = LocalResources.current.getString()

```

```

val LocalResources = staticCompositionLocalOf { ResourcesImpl("EN") }

@Composable
fun ApplicationTheme(content: @Composable () -> Unit) {
    CompositionLocalProvider(LocalResources provides ResourcesImpl(Locale.current.language)) {
        content()
    }
}

@Composable
fun ApplicationScreen() {
    ApplicationTheme {
        Text(text = getString().app_background)
    }
}

```

Drawables



Similarly, the two solutions when dealing with drawables involve one with automation and the other without.

First solution: Without automation

The first approach is to duplicate the same drawable for each platform. Then, you need to implement the `PainterRes` object for each platform by using a specific way of reading the drawable. The code below is an example of how you can implement the `PainterRes` object for desktop and Android platforms. Look carefully as you'll notice that we're referencing to a drawable by `path` on the desktop but by `resource id` on Android.

PainterRes implementation

commonMain/Kotlin/PainterRes.kt

```
import androidx.compose.runtime.Composable
import androidx.compose.ui.graphics.painter.Painter

expect object PainterRes {
    @Composable
    fun loginBackground(): Painter
}
```


desktopMain/kotlin/PainterRes.kt

```
import androidx.compose.runtime.Composable
import androidx.compose.ui.graphics.painter.Painter
import androidx.compose.ui.res.painterResource

actual object PainterRes {
    @Composable
    actual fun loginBackground(): Painter = painterResource("images/login_background.png")
}
```

androidMain/kotlin/PainterRes.kt

```
import androidx.compose.runtime.Composable
import androidx.compose.ui.graphics.painter.Painter
import androidx.compose.ui.res.painterResource

actual object PainterRes {
    @Composable
    actual fun loginBackground(): Painter = painterResource(R.drawable.login_background)
}
```

Here's an example when using the solution above.

```
@Composable
fun ApplicationScreen() {
    ApplicationTheme {
```

```
        Image(painter = PainterRes.loginBackground())  
    }  
}
```

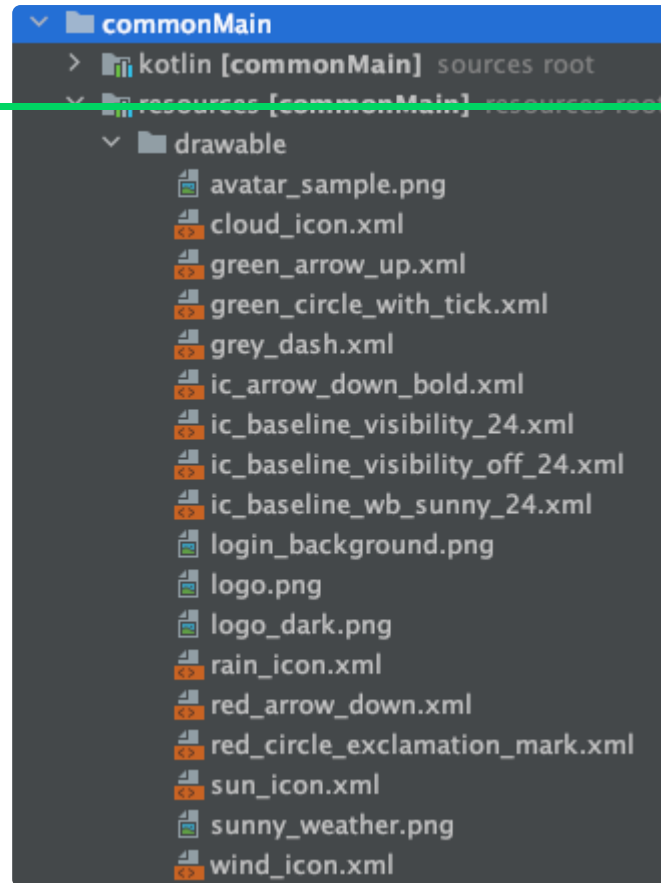
Now, imagine that you have 100 drawables in your app. First, you must duplicate every drawable for each platform. Then, you'll need to implement `PainterRes`. This approach requires a lot of manual work.

Second solution: With automation

The other way uses a well-known code generation mechanism. A bit of a different method using `expect / actual` will help you simplify this solution.

1. Put all drawables under `drawable/`.





2. Then, run the `resourceGeneratorTask`.



```

@Immutable
public class ResourcesImpl(
    private val locale: String
) {

    public fun getDrawableResources() = DrawableResources

    @Immutable
    public object DrawableResources {
        public val red_circle_exclamation_mark: String = "red_circle_exclamation_mark.xml"
    }

    @ReadOnlyComposable
    @Composable
    public fun getDrawables(): DrawableResources = LocalResources.current.getDrawableResources()

```

3. Finally, you need to provide a drawable down the composition tree. By using `imageResources`, you'll be able to get the drawable.

```

val LocalResources = staticCompositionLocalOf { ResourcesImpl("EN") }

@Composable
fun ApplicationTheme(content: @Composable () -> Unit) {
    CompositionLocalProvider(LocalResources provides ResourcesImpl(Locale.current.language)) {
        content()
    }
}

```

```
}  
  
@Composable  
fun ApplicationScreen() {  
    ApplicationTheme {  
        Image(painter = imageResources(image = getDrawables().red_circle_exclamation_mark))  
    }  
}
```

Most of the work will be done by the Gradle task. You just need to provide the drawable down the tree.

ImageResources implementation

commonMain/kotlin/imageResources

```
import androidx.compose.runtime.Composable  
import androidx.compose.ui.graphics.painter.Painter  
  
@Composable  
expect fun imageResources(image: String): Painter
```

androidMain/kotlin/imageResources

```
import androidx.compose.runtime.Composable  
import androidx.compose.ui.graphics.painter.Painter
```

```
import androidx.compose.ui.platform.LocalContext
import androidx.compose.ui.res.painterResource

@Composable
actual fun imageResources(image: String): Painter {
    val context = LocalContext.current
    val name = image.substringBefore(".")
    val drawable = context.resources.getIdentifier(name, "drawable", context.packageName)
    return painterResource(drawable)
}
```

desktopMain/kotlin/imageResources

```
import androidx.compose.runtime.Composable
import androidx.compose.ui.graphics.painter.Painter
import androidx.compose.ui.res.painterResource

@Composable
actual fun imageResources(image: String): Painter = painterResource("drawable/$image")
```

Conclusion



As you can see, because it's not particularly obvious how to handle resources in Kotlin Multiplatform, we had to come up with some creative workarounds. Admittedly, resourceGeneratorTask plays a vital role as it does most of the work for us. When these things happen, it's important to think of solutions that are as simple as possible to implement.

TAGS

Android

Kotlin-Multiplatform



MORE POSTS BY THIS AUTHOR

Krzysztof Kansy

Former Android Developer at Netguru



We're Netguru!



At Netguru we specialize in designing, building, shipping and scaling beautiful, usable products with blazing-fast efficiency

TRUSTED BY:



Netguru S.A

Nowe Garbary Office Center
ul. Małe Garbary 9
61-756 Poznań, Poland

VAT-ID: PL7781454968

REGON: 300826280

Follow Us

Bē




KRS: 0000745671

hello@netguru.com

Digital Acceleration Editorial

Boost innovation with insights for change

Submit

☐ I agree to receive marketing communication from
Netguru. 

Certificates:

Certified

Corporation



Partnerships:

 Microsoft



 Google Cloud
Partner

Recognized by:





©2023 Netguru S.A. All rights reserved.

