Building Desktop Applications

CS 346: Application Development

Features

What makes a desktop application?

Modern Features

1. Graphical User Interface.

Interactive graphical interfaces are a major part of a modern application. Your application should support these features:

- Interaction using standard controls e.g., buttons, panels, images, scrollbars, and so on. We'll discuss how to use standard controls from a GUI toolkit.
- Rich data, animations and other design elements that add to the aesthetics and appeal of the platform.

Modern Features

2. Keyboard + mouse interaction

Interaction based on keyboard and mouse (pointing device) support:

- Users should be able to use the keyboard for navigation & common tasks. You should support keyboard shortcuts aka “hotkeys” when possible.
- The mouse should be used for most selection tasks, and for manipulating data i.e., a right-click context menu, or dropdown menu.
- Support standard navigation conventions for the platform. e.g., window manipulation including window and content resizing.
- Undo/redo cycle i.e., being able to explore the interface by performing and potentially undoing actions.

Modern Features

3. Rich Data Manipulation

Users expect to be able to manipulate data in a variety of ways:

- Cut-Copy-Paste. You should be able to use these commands in both desktop and mobile applications to manipulate text and image data.
- Drag-Drop. When an application requires you to move data from one place to another, you should be able to drag and drop it. e.g., dragging an image from your file system into a dialog box.

Compose > Desktop

Desktop-specific GUI considerations.

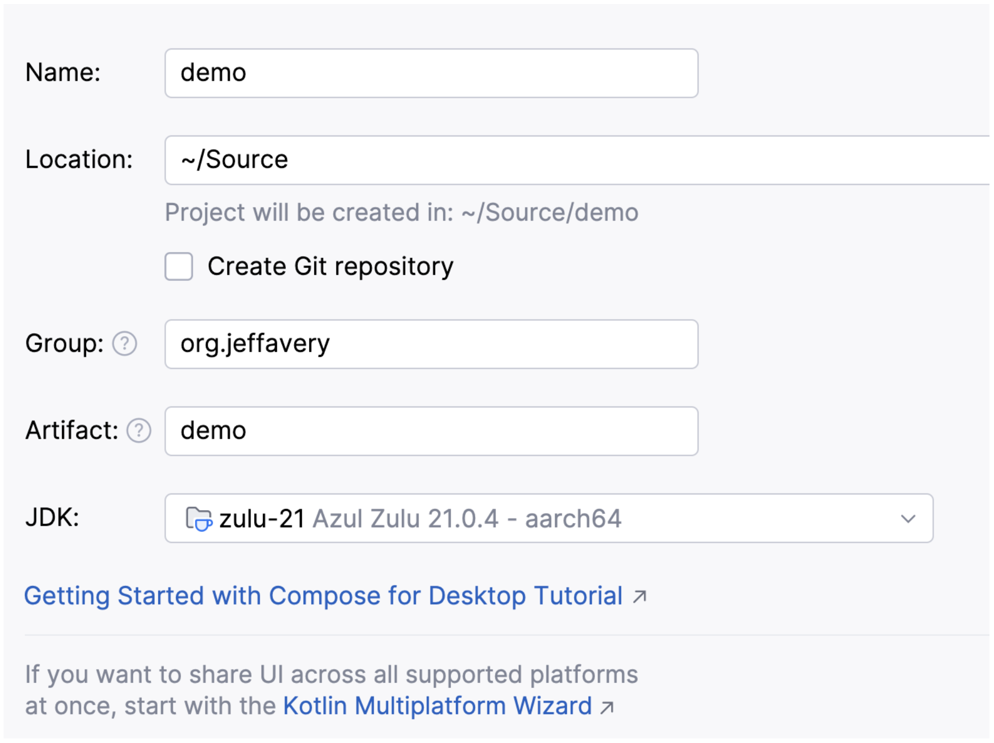
Creating a desktop project

Use IntelliJ IDEA

- Install the plugin for **Compose Multiplatform IDE Support**
- File > New Project > Compose Multiplatform

Do NOT use the Kotlin Multiplatform Wizard

- KMP uses a different project format
- If you don't know what this wizard is…. Ignore this comment!



Name:	demo
Location:	~/Source
	Project will be created in: ~/Source/demo
	☐ Create Git repository
Group:	org.jeffavery
Artifact:	demo
JDK:	zulu-21 Azul Zulu 21.0.4 - aarch64

Getting Started with Compose for Desktop Tutorial ↗

If you want to share UI across all supported platforms at once, start with the Kotlin Multiplatform Wizard ↗

Install Compose Dependencies

If you have an **existing** project, you need to add dependencies + compiler plugins. New projects will already include this.

libs.versions.toml

```
[plugins]
kotlin-jvm = {id = "org.jetbrains.kotlin.jvm", version.ref = “2.0.20"}
jetbrains-compose = {id = "org.jetbrains.compose", version.ref = “1.6.11"}
compose-compiler = {id = "org.jetbrains.kotlin.plugin.compose", version.ref = “2.0.20"}
```

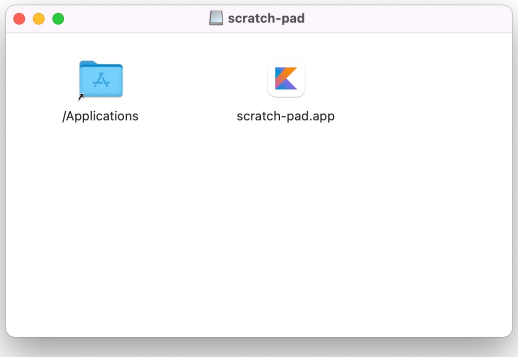
build.gradle.kts

```
plugins {
    alias(libs.plugins.jetbrains.compose)
    alias(libs.plugins.compose.compiler)
}
dependencies {
  implementation(compose.desktop.currentOs)
}
```

Gradle Tasks for Desktop

Use the Gradle menu (`View` > `Tool Windows` > `Gradle`).

Command	What does it do?
`Tasks` > `build` > `clean`	Removes temp files (deletes the `/build` directory)
`Tasks` > `build` > `build`	Compiles your application
`Tasks` > `compose desktop` > `run`	Executes your application (builds it first if necessary)
`Tasks` > `compose desktop` > `package`	Create an installer for your platform!



Application Structure

A desktop GUI application is just a regular application that uses Compose.

It needs to:

- use a **main method** as its entry point,
- declare a **top-level application scope**,
- declare one or more **windows** within that application scope.

Nothing new here!

```
fun main() = application {
    Window(
        title = "Minimum Window",
        onCloseRequest = ::exitApplication
    ) {
        Text("Hello Compose!")
    }
}
```

samples/desktop/compose-demo -> run **MinimumWindow** main method

Window position/size

Create a `WindowState` for the Window composable and pass in the appropriate values.

```kotlin
fun main() {
    application {
        Window(
            title = "WindowState",
            state = WindowState(
                position = WindowPosition(0.dp, 0.dp),
                size = DpSize(300.dp, 200.dp)
            ),
        onCloseRequest = ::exitApplication
        ) {
            Text("This is a window")
    }
}
```

samples/desktop/compose-demo -> run **WindowState** main method

Adding Menus

```kotlin
fun main() = application {
    Window(onCloseRequest = ::exitApplication) {
        App(this, this@application)
    }
}

@Composable
fun App(
    windowScope: FrameWindowScope,
    applicationScope: ApplicationScope
) {
    windowScope.MenuBar {
        Menu("File", mnemonic = 'F') {
            val nextWindowState = rememberWindowState()
            Item(
                "Exit",
                onClick = { applicationScope.exitApplication() },
                shortcut = KeyShortcut(
                    Key.X, ctrl = false
                )
            )
        }
    }
}
```

Keyboard Input

```
fun main() = application {
    Window(
        title = "Key Events",
        state = WindowState(width = 500.dp, height = 100.dp),
        onCloseRequest = ::exitApplication,
        onKeyEvent = {
            if (it.type == KeyEventType.KeyUp) {
                println("Window handler: " + it.key.toString())
            }
        }
    ) {
        MaterialTheme {
            Frame()
        }
    }
}
```

It's just a new event type
Captured at window level, so
no focus issues.

Mouse Input

```kotlin
Box(
    modifier = Modifier
        .background(Color.Magenta)
        .fillMaxWidth(0.9f)
        .fillMaxHeight(0.2f)
        .combinedClickable(
            onClick = { text = "Click! ${count++}” },
            onDoubleClick = { text = "Double click! ${count++}” },
            onLongClick = { text = "Long click! ${count++}” }
        )
)
```

Mouse Movement

```kotlin
var color by remember { mutableStateOf(Color(0, 0, 0)) }

Box(
    modifier = Modifier
        .wrapContentSize(Alignment.Center)
        .fillMaxSize()
        .background(color = color)
        .onPointerEvent(PointerEventType.Move) {
            val position = it.changes.first().position
            color = Color(position.x.toInt() % 256, position.y.toInt() % 256, 0)
        }
)
```

Drag-Drop Interaction

Compose supports drag and drop with two modifiers:

- `dragAndDropSource`: Specifies a composable as the starting point.
- `dragAndDropTarget`: Specifies a composable that accepts the dropped data

e.g., to enable users to drag an image in your app, create an image composable and add the `dragAndDropSource` modifier. To set up a drop target, create another image composable and add the `dragAndDropTarget` modifier.

```
Modifier.dragAndDropSource {
    detectTapGestures(onLongPress = {
        // Transfer data here.
    })
}
```

https://developer.android.com/develop/ui/compose/touch-input/user-interactions/drag-and-drop

Creating an installer

Gradle tasks are built-in for this!

`Tasks > Compose desktop > packageDistributionforCurrentOS`

You need to use a Mac to build a macOS installer, Windows to build a Windows installer etc.

- No other platform specific code i.e., your app should run everywhere.
- When building installers, target the platforms that you have access to.