C. F. G. S. DE DESARROLLO DE APLICACIONES MULTIPLATAFORMA

TÍTULO DEL PROYECTO: FOREST FIRE SIMULATOR

AUTOR/AUTORES: SERGIO PÉREZ SANZ

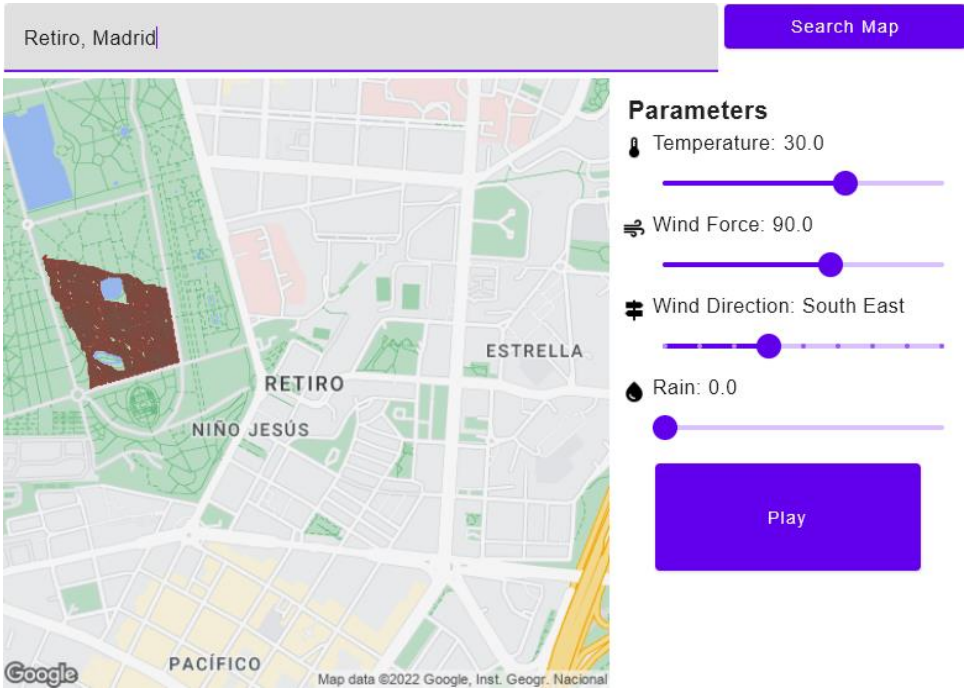
CURSO 2021/2022

INTRODUCCIÓN

¿Qué es Forest Fire Simulator?

Forest Fire Simulator simula el comportamiento de un incendio en una localización determinada. El resultado es una animación bidimensional que superpone pixeles rojos sobre el mapa simulando dicho incendio.

Los usuarios pueden elegir una ubicación incendiable (debe contener al menos un determinado porcentaje de zona verde para realizar una simulación significativa) y las condiciones climatológicas.



¿Por qué FFS?

La razón por la elijo esta aplicación es principalmente por su atractivo visual y la facilidad con la que se puede observar todo su potencial. Estas son las características perfectas para llevar a cabo el objetivo del proyecto: realizar una **aplicación multiplataforma de código único**.

OBJETIVOS

Tecnología multiplataforma

Con este proyecto deseo aprender a hacer una aplicación multiplataforma que utilice una tecnología común para todas ellas. Quiero que dicha tecnología no esté en mi repertorio actual y que hacer este proyecto me ayude a generalizar conocimientos de desarrollo de software, puesto que actualmente solo dispongo de conocimientos sobre Java/Spring/JavaFX.

Aprender Kotlin

Kotlin es un lenguaje que captó mi atención por sus soluciones a muchos de los problemas de Java. Pienso que tiene mucho futuro dentro del desarrollo multiplataforma y deseo estar cómodo programando en este lenguaje. Profundizo en por qué Kotlin es una tecnología interesante en el apartado Análisis de Tecnologías.

ALCANCE DEL PROYECTO

El proyecto busca tener una simulación de incendio funcional para cualquier ubicación que el usuario seleccione. La simulación se realizará sobre una versión simplificada a 3 colores del mapa de la zona. Estos colores serían: verde (campo), azul (agua) y amarillo (carreteras).

También se desea generar un objeto que resuma la ejecución y que se pueda importar en la aplicación para reproducirla de nuevo. Este objeto se puede subir a un servidor mediante una operación POST de API REST.

Se desea que la aplicación sea accesible desde al menos un PC (Mac, Windows y Linux) y desde un dispositivo móvil (iOS y Android).

Los datos que el usuario puede introducir son: Ubicación, precipitación, temperatura, dirección del viento y fuerza del viento.

JUSTIFICACIÓN

El motivo por el que quiero aprender una tecnología multiplataforma es por su futuro. La tecnología evoluciona para ofrecer comodidad a quien la utiliza, y los desarrolladores actualmente gastamos muchos recursos del desarrollo para ofrecer esta disponibilidad multiplataforma. Conocer las técnicas y las bases de estas tecnologías me parece muy importante para poder participar en proyectos que incluyan este paradigma, que insisto, creo que es el futuro del software.

Este proyecto busca asentar esas bases usando una aplicación simple y visual, que sirva para demostrar que lo importante es el contenido, no el dispositivo desde el que lo visualices. Las opciones barajadas para dicha aplicación fueron:

- Simulación de ecosistema: Un tablero en el que visualizar criaturas simples luchar por recursos escasos para sobrevivir o reproducirse.
- Ajedrez
- Forest Fire Model (FFM): Algoritmo lineal utilizado para estudiar el comportamiento del fuego frente a recursos inflamables y barreras naturales/artificiales.

La simulación del ecosistema presentaba un algoritmo parecido al FFM y presentaba menos utilidad social, de modo que fue descartado a favor del mismo. El ajedrez era ideal para el proyecto, pero desarrollar una IA competente de ajedrez sobrepasaba con creces el alcance del proyecto. De esta forma, el FFM fue elegido como la mejor opción, puesto que me permitía practicar tecnologías que ya conocía (API REST para obtención de mapas) a la vez que aprendía este nuevo paradigma.

ANÁLISIS

Análisis inicial del problema

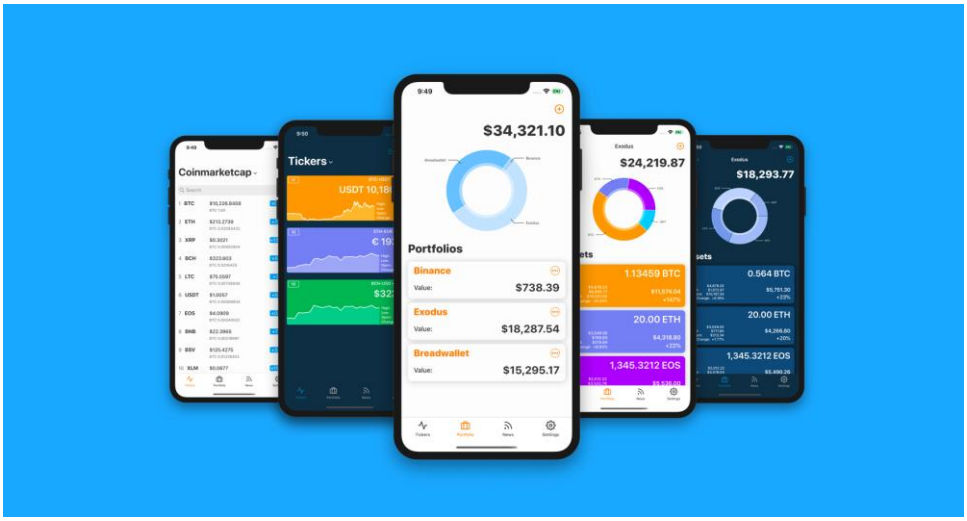
El principal problema al que nos enfrentamos es como **visualizar el algoritmo de manera adecuada**. La implementación del Forest Fire Model es muy sencilla sobre un mapa adecuado para ella, pero la generación de dicho mapa es el primer punto de dificultad. ¿Dónde obtengo un mapa de tres colores para su análisis? La solución a este problema será la Maps Static API, servicio REST que nos provee imágenes de mapas de Google Maps.

El segundo problema es **disponer de una tecnología multiplataforma** para programar la aplicación. Para ello voy a usar Jetpack Compose Multiplatform. Jetpack es la última tecnología utilizada en Android basada en Kotlin, y su versión multiplatform permite generar proyectos en los que reutilizamos los componentes de Android tanto en Web como en Escritorio. Todo ello compilando sobre la Java Virtual Machine.

ANÁLISIS DE TECNOLOGÍAS

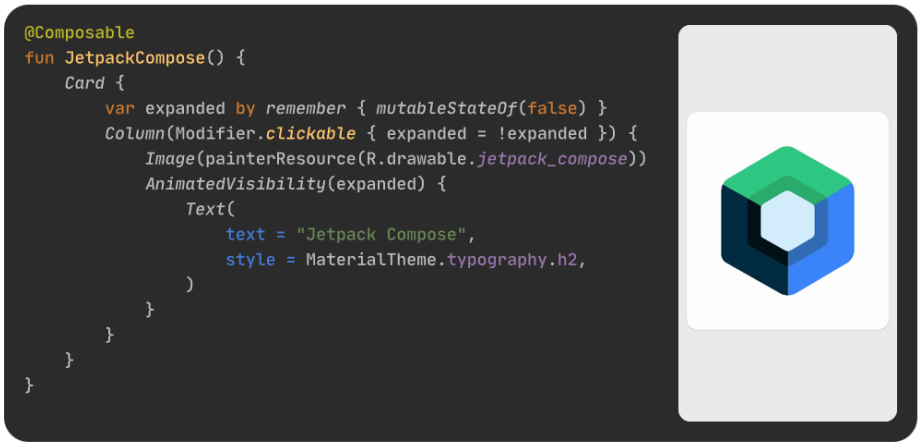
Varias tecnologías poseen el título de "Multiplataforma", pero las dos adecuadas para este proyecto son Flutter y Jetpack Multiplatform.

Flutter



Es una tecnología de interfaces declarativa que se basa en el lenguaje de programación Dart. Mediante la generación de Widgets reutilizables genera interfaces de usuario muy elegantes y aptas para todas las plataformas.

Jetpack Compose Multiplatform

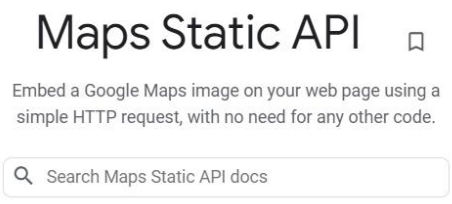


Lleva el paradigma de la composición de interfaces de Android a todas las plataformas. Está basado en Kotlin y requiere del SDK de Android. Genera interfaces de usuario acordes con el diseño Material. Estas interfaces están formadas de Componentes Composable, funciones de Kotlin que reciben un estado de la aplicación y representan dicho estado. Al cambiar el estado, se vuelve a llamar a la función para reflejar dicho cambio. Jetpack Multiplatform es la tecnología elegida por su afinidad con este proyecto al permitirme aprender Kotlin.

TECNOLOGÍAS DE GEOCODING

Para poder encontrar mapas a partir de coordenadas o cadenas de texto necesitamos un servicio de geocoding. Google ofrece un servicio llamado Google Cloud Platform en el que ofrece varias API relacionadas con sus servicios de Maps. Como alternativa, existe Here API, la cual ofrece APIs de localización muy potentes gratis para aplicaciones en desarrollo.

Maps Static API



Dentro del servicio Google Cloud Platform, la API que se adecua a las necesidades de este proyecto es Maps Static API. Recibe una llamada https en la que debemos especificar al menos una localización, nuestra API key, un nivel de vista (de 1 a 20) y un tamaño de la imagen para poder obtener una imagen de Google Maps con dichas especificaciones. Adicionalmente, podemos proveer de estilo a la imagen, elegir vista de mapa o de satélite o cambiar de formato de archivo.

Here API

Figure 1. Map image of central Berlin, Germany

La alternativa a la Static API de Maps es Here API. Ofrece varios servicios relacionados con los mapas y la localización. En concreto su Map Image API nos ofrece una imagen al igual que la anterior, pero esta vez sin estilizar. Es por este último detalle que decido utilizar la API de Maps.

LENGUAJE ELEGIDO: KOTLIN

He elegido Kotlin como lenguaje a aprender debido a varios motivos que voy a exponer ahora.

Kotlin nace con la idea de mejorar Java sin dejar de ser compatible con este. Esta ya es una primera premisa que me atrae, puesto que Java es el primer lenguaje que aprendí y hacer este salto cuesta mucho menos que a cualquier otro lenguaje. Kotlin se presenta como un lenguaje con soluciones para todos los problemas modernos: desarrollo móvil (principal lenguaje de Android), desarrollo front(Compose) y back(Ktor, Spring). Ofrece la potente orientación a objetos de Java junto a programación funcional. Simplifica el lenguaje de Java haciéndolo más legible y fácil de aprender, incluso llegando a parecer lenguaje natural en algunos casos. Kotlin también se

libra del molesto NullPointerException, no permitiendo que sus variables contengan valores nulos salvo que el programador así lo desee.

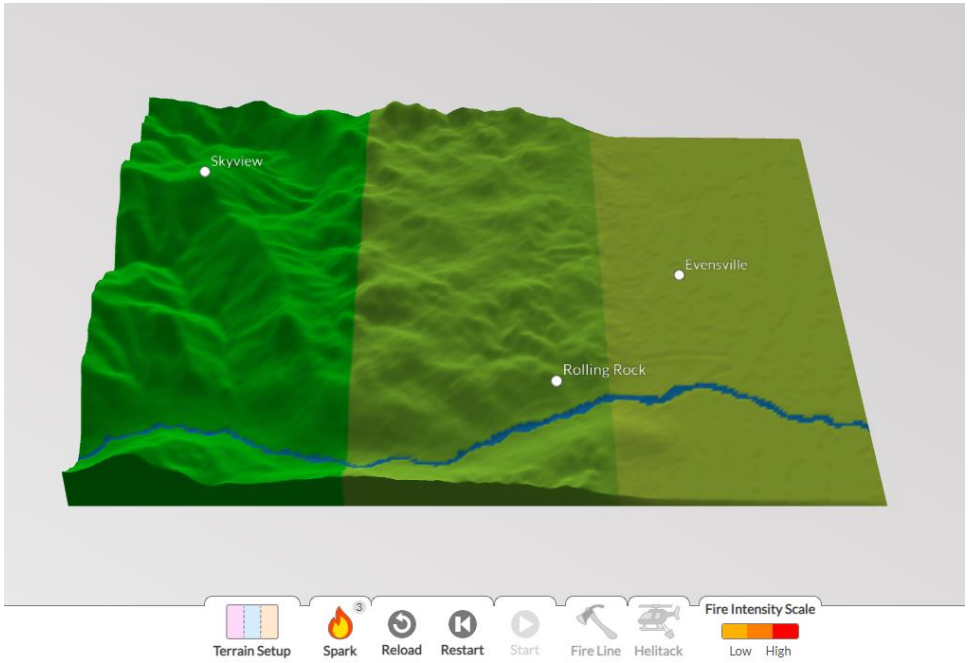
Las principales ventajas de las que me he aprovechado en este proyecto son:

- Nuevo sistema de clases: Puedes crear una clase "POJO" de java tan solo añadiendo "data" delante de una clase. También puedes implementar el patrón "Singleton" cambiando class por object. Adicionalmente, todos los atributos de la clase son accesibles como propiedades. Esto quiere decir que no es necesario que escribamos clase.getAtributo() a la hora de llamar a los atributos, podemos tan solo escribir "clase.atributo". Los métodos getter, setter, toString, etc. se generan automáticamente con la clase. Todos estos cambios han agilizado mucho mi programación del modelo en Kotlin.
- Corutinas de Kotlin: Kotlin ofrece una solución alternativa a los hilos clásicos: las corutinas. Kotlin define las corutinas como hilos ligeros, que permiten mucha mayor carga de tareas con mucha menos consumición de memoria por corutina. La documentación de Kotlin muestra cómo se pueden generar diez mil corutinas sin desestabilizar una máquina, dado que la tarea que ejecutan es muy ligera. Gracias a esta nueva forma de generar código asíncrono, la programación en Android es mucho más ágil e intuitiva y la asincronía se convierte en una herramienta fácil de utilizar.

PRODUCTOS SIMILARES

Wild Fire, Simulador 3D

Wild Fire ofrece un territorio de juegos donde poder simular incendios en una zona forestal de ejemplo. Es una herramienta de visualización muy potente que sirve para entender el comportamiento general de un fuego, pero no consigue personalizar la ejecución a la localización deseada. También ofrece parametrización climatológica, que es uno de los atractivos de mi aplicación.



Simulador ERVIN

El simulador ERVIN genera un entorno 3D donde podemos interactuar con un incendio forestal y entrenar a los servicios de emergencia en esa situación concreta. Este simulador permite generar distintos incendios y colocar distintos usuarios que actúan como bomberos. Su objetivo dista de entender como sucedería un incendio en una zona concreta, pues también se sucede en un mapa fijo.

REQUISITOS DEL PROYECTO

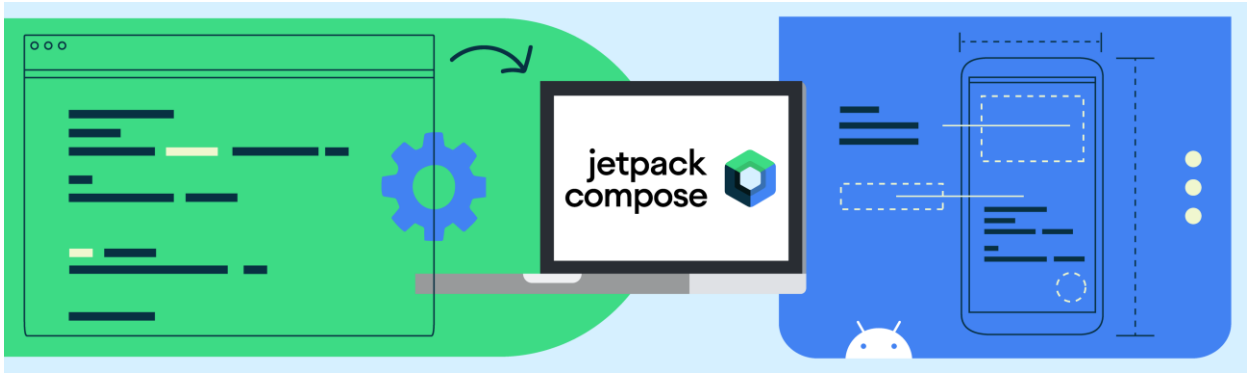
Requisitos funcionales

- Ejecutar una simulación sobre un mapa
- Elegir el punto inicial de la simulación con un clic
- Cambiar de mapa buscando una dirección o coordenadas
- Cambiar los parámetros de la ejecución

Requisitos no funcionales

- La aplicación debe funcionar en Android y Escritorio
- La distribución de los componentes debe adaptarse a cada plataforma
- El almacenamiento de simulaciones debe realizarse mediante una base de datos NoSQL

USO BÁSICO DE JETPACK COMPOSE



Jetpack Compose es una tecnología de interfaces de usuario declarativas que nos permite crear componentes visuales como si fueran funciones de programación. De esta manera, la función Button() te permite crear un botón, mientras que la función Column() te permite crear una

distribución para los componentes en pantalla. Estos dos componentes son ejemplos de todos los disponibles en esta tecnología. La mayoría de ellos provienen de componentes de Compose para Android.

Esta es la demo que aparece en la web de Compose.

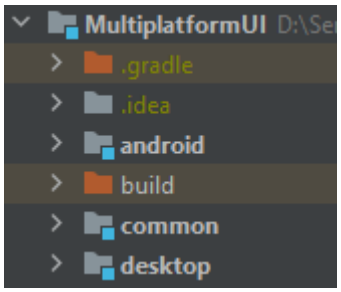
```
@Composable
fun JetpackCompose() {
    Card {
        var expanded by remember { mutableStateOf(false) }
        Column(Modifier.clickable { expanded = !expanded }) {
            Image(painterResource(R.drawable.jetpack_compose))
            AnimatedVisibility(expanded) {
                Text(
                    text = "Jetpack Compose",
                    style = MaterialTheme.typography.h2,
                )
            }
        }
    }
}
```

En esta demo podemos observar el paradigma básico de Compose: los Componentes reaccionan a cambios en el estado. El estado de este componente es el booleano expanded. El componente Card() tiene la opción de expandirse mediante el método AnimatedVisibility(). Este método recibe un booleano para determinar si debe expandirse o no. En caso de expandirse, ejecuta el código que tiene dentro (el componente Text()). Por último, si la columna recibe un clic, alternaremos el estado. Alternar el estado implica recomponer la IU, es decir, se vuelve a ejecutar cada componente que estuviera asociado a ese estado. En este caso, la animación del componente Card se volvería a ejecutar.

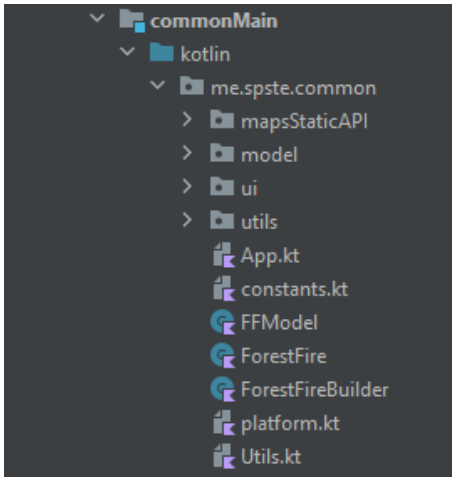
DISEÑO

ESTRUCTURA DEL PROYECTO

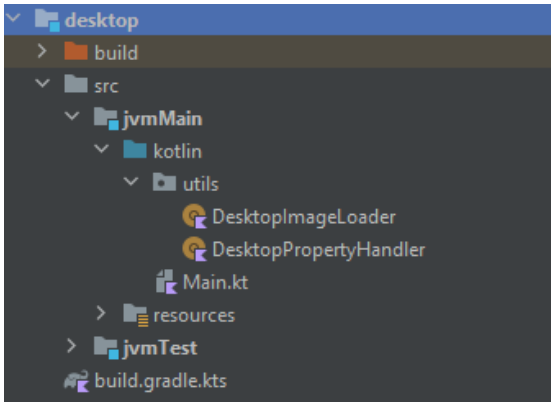
La aplicación está estructurada según la plantilla de Jetpack Multiplatform (se aplica al seleccionar el tipo de proyecto en IntelliJ). Esta plantilla separa el código fuente en tres subproyectos: Android, Common y Desktop. Cada uno de estos subproyectos tiene sus propias dependencias y su propia build de Gradle dependiente de la principal.

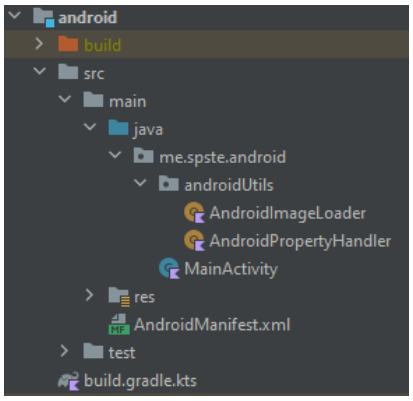


En Common debe encontrarse todo código fuente que deba estar disponible para cualquier plataforma. Por ejemplo, los componentes de la UI que he desarrollado, la lógica de la aplicación y los servicios de acceso a APIs.

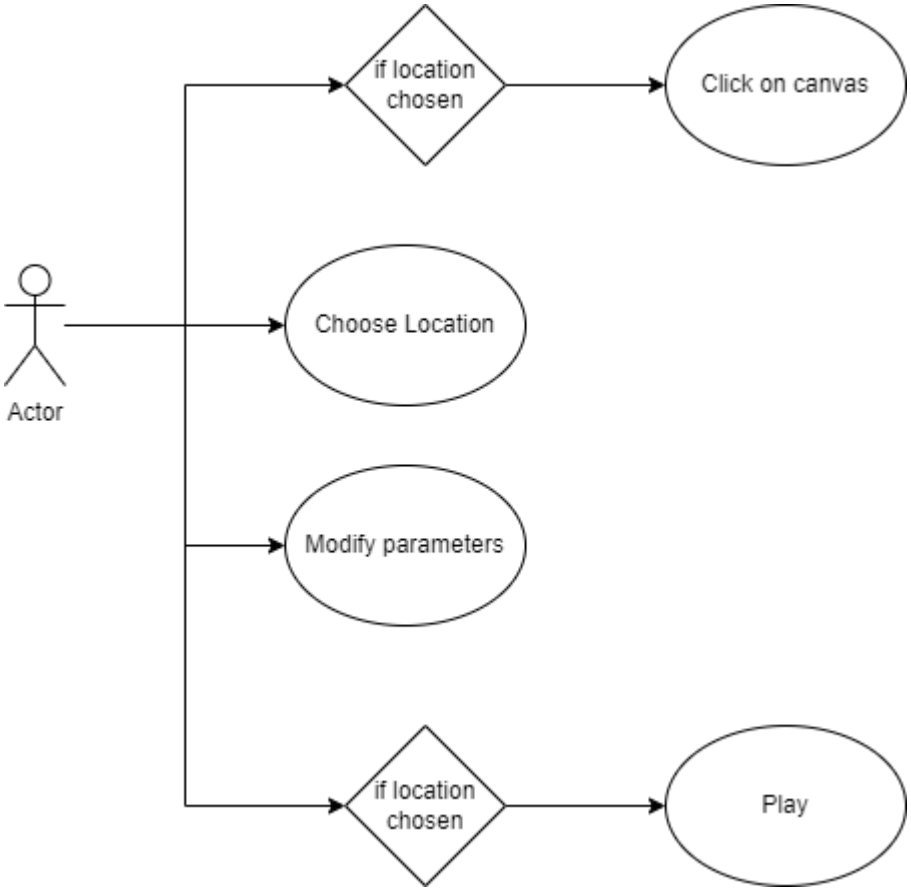


En Android y en desktop debe encontrarse el código propio de dicha plataforma necesario para funcionar (Manifest de Android, recursos propios de la plataforma, etc.)





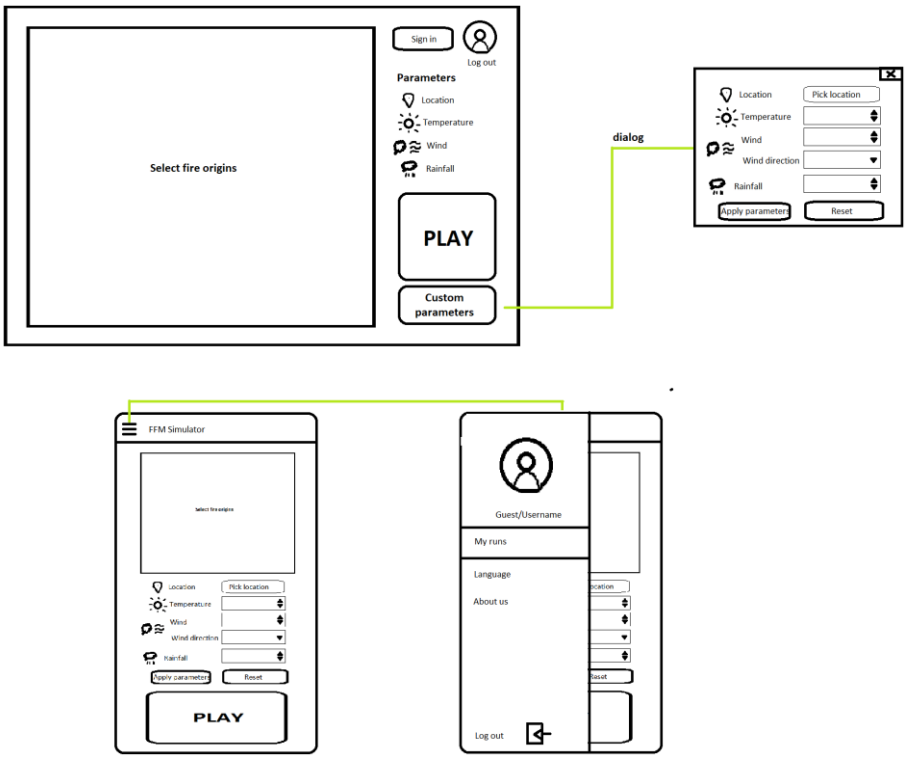
CASOS DE USO



DISEÑO DE LA INTERFAZ

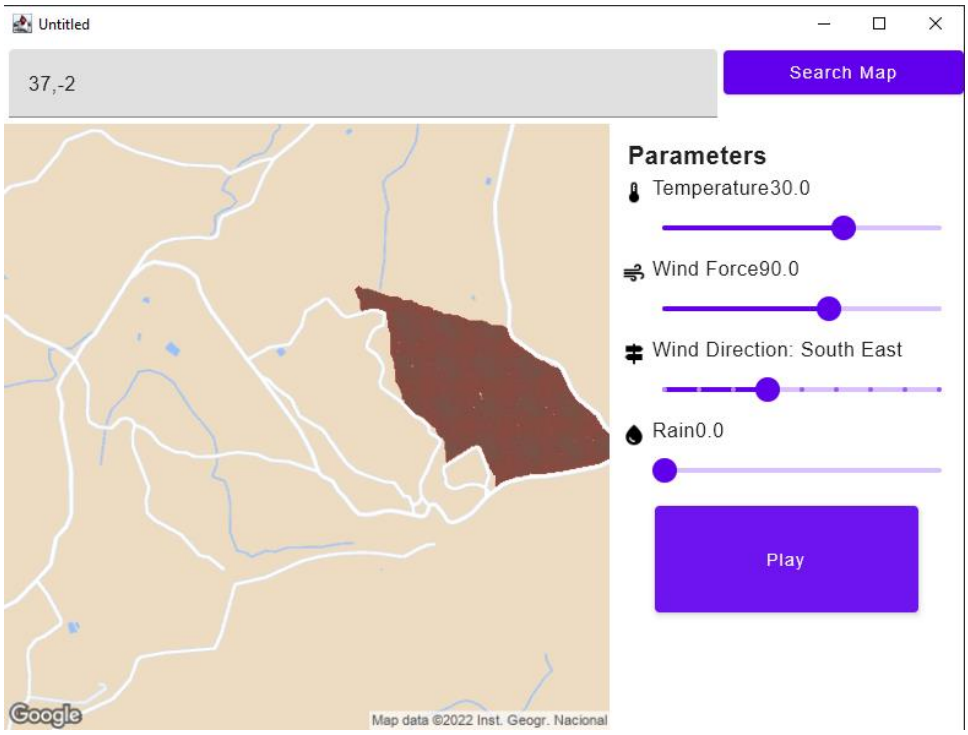
Diseños originales

Originalmente, la aplicación iba a ser mucho más amplia. Tanto el login como los menús en Android han acabado fuera del alcance, pero también ha habido cambios en el diseño de los componentes implementados.



Como podéis observar, originalmente el diseño de escritorio incluía un pop-up que permitía editar los parámetros desde un Dialog. Esta idea fue descartada por la diferencia innecesaria que generaba con el diseño en Android. Otro cambio fue la inclusión de Sliders en vez de DropDownMenus debido a que Jetpack Compose Desktop no incluye este componente aún.

Diseños actuales



En ellos podemos observar que el principal componente es la superficie del mapa para ambas plataformas. Adicionalmente encontramos un botón de "Reproducir" y una vista de parámetros. Cuando no existe un clic en el mapa, aparece un mensaje que indica que seleccionemos el lugar de origen del fuego.

Componentes

Todos los componentes se encuentran en el directorio ui dentro del proyecto common. Gracias a esto, podemos acceder a ellos desde los otros dos proyectos, Android y Desktop, para su correcta reutilización.

Botón Play

El botón de play es el componente más básico de Jetpack Compose. Recibe una función por parámetro que ejecutará cuando sea pulsado y recibe unos modificadores para ser adaptado a cualquier interfaz. Esta práctica nos permite que este botón se pueda utilizar en cualquier distribución de componentes.

```
@Composable
fun PlayButton(onClick: () -> Unit, modifier: Modifier) {
    Button(
        onClick = onClick,
        modifier = modifier
    ) { this: RowScope
        Text( text: "Play")
    }
}
```

Parámetros

Los parámetros reciben los datos de la ejecución y los métodos de actualización de dichos valores para presentar unos sliders con iconos que permiten modificar los datos de la ejecución.

```
@Composable
fun ParametersView(
    active: Boolean,
    builder: ForestFireBuilder,
    modifier: Modifier,
    loader: ImageLoader,
    onTemperatureChange: (Float) -> Unit,
    onWindValueChange: (Float) -> Unit,
    onWindDirectionChange: (Float) -> Unit,
    onRainfallChange: (Float) -> Unit
) {
    Column(modifier = modifier) { this: ColumnScope
        Text(text = "Parameters", fontSize = 20.sp, fontWeight = FontWeight.Bold, modifier = Modifier.padding(5.dp))
        TemperatureParameterView(builder.climate?.temperature, loader, onTemperatureChange, active)
        WindParameterView(builder.wind, loader, onWindValueChange, onWindDirectionChange, active)
        RainfallParameterView(builder.climate?.precipitation, loader, onRainfallChange, active)
    }
```

Simulador

El simulador está compuesto por dos capas: Imagen y Fuego.

Para poder realizar un componente con capas superpuestas utilizamos el componente Surface() de Jetpack Compose. Este componente superpone sus hijos. De esta forma, podemos mostrar la ejecución encima del mapa.

La ejecución se realiza en el componente Fire(). Fire contiene un lienzo (Canvas()) que permite dibujar líneas en posiciones concretas. Para representar los píxeles, el componente dibuja líneas de tamaño 1dp en la posición correspondiente al fuego en la imagen. Esta posición se calcula leyendo el mapa y capturando el tamaño actual de la ventana, pues es necesario escalar el tamaño de los píxeles si el usuario expande la ventana.

```
@Composable
fun Fire(imageBitmap: ImageBitmap, simulation: ForestFire?, variationIndex: Int, canvasSize: IntSize, click: Offset?) {
    val ratioHeight = canvasSize.height.toFloat() / imageBitmap.height.toFloat()
    val ratioWidth = canvasSize.width.toFloat() / imageBitmap.width.toFloat()
    Canvas(
        Modifier.aspectRatio( ratio: 1f,  matchHeightConstraintsFirst: true)
    ) {  this: DrawScope

        if ((click != null) && (click.x != -1f) && (click.y != -1f)) {
            drawLine(
                FIRE_COLOR,
                Offset(
                     x: click.x * ratioHeight,
                     y: click.y * ratioWidth
                ),
                Offset(
                     x: (click.x + 1) * ratioHeight,
                     y: (click.y + 1) * ratioWidth
                ),
                 strokeWidth: 5f
            )
        }
        if (simulation != null) {
            if (simulation.result != null) {
                for (i in 0  until  < variationIndex) {
                    simulation.result?.variation?.get(i)?.pixelUpdateList?.forEach {  it: PixelUpdate
                        val color = when (it.value) {
                            FIRE -> FIRE_COLOR
                            BURNT -> BURNT_COLOR
                            TREE -> TREE_COLOR
                            WATER -> WATER_COLOR
                            ROAD -> ROAD_COLOR
                            else -> TREE_COLOR
                        }
                        drawLine(
                            color,
                            Offset(
                                 x: it.x.toFloat() * ratioHeight,
                                 y: it.y.toFloat() * ratioWidth
                            ),
                            Offset(
                                 x: (it.x.toFloat() + 1) * ratioHeight,
                                 y: (it.y.toFloat() + 1) * ratioWidth
                            )
                        )
                    }
                }
            }
        }
    }
}
```

DISEÑO DE LA LÓGICA

Para generar las simulaciones, hay varias clases dentro del proyecto que trabajan entre ellas en un orden: Construcción parametrizada de la ejecución, Ejecución y generación del objeto reporte. Todo este proceso está encapsulado en una clase superior llamada FFModel.

FFModel

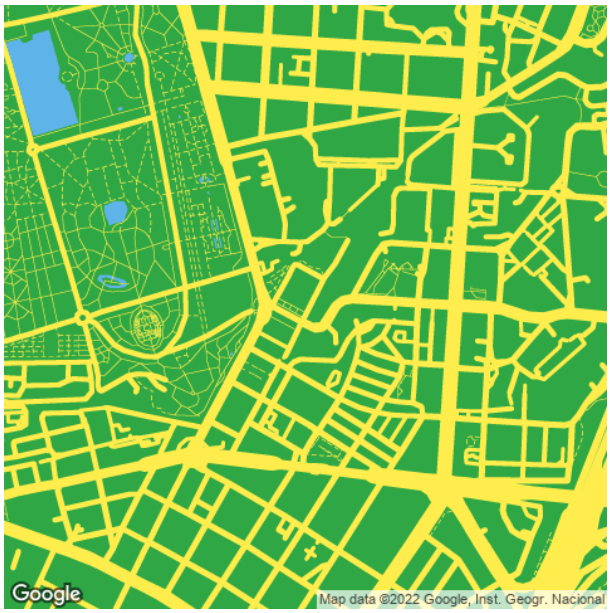
La lógica de la ejecución la realiza la clase FFModel. Esta clase se encarga de empaquetar toda la lógica del proyecto. Nos permite acceder al builder de ejecuciones, contener la ejecución actual, ejecutarla y reiniciar la lógica.

```
class FFModel(
    var analysisImage: ImageBitmap?,
    var propertiesHandler: PropertiesHandler
) {
    val builder = ForestFireBuilder()
    var run: ForestFire? = null
    var animationIndex = Animatable( initialValue: 0f)
```

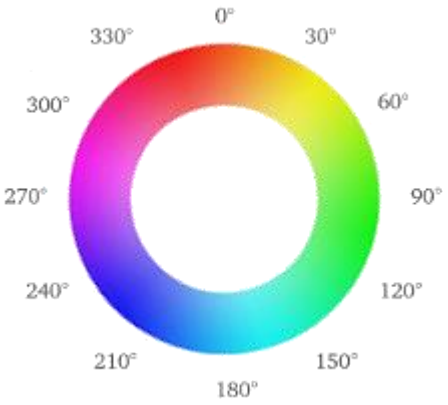
Para poder generar el mapa de enteros, se recibe una imagen en formato ImageBitmap. Este formato consiste en convertir una imagen en una matriz de píxeles. Gracias a este formato,

podemos analizar cada píxel de la imagen y determinar si es inflamable o barrera. Este análisis parece simple con la premisa de que la imagen está compuesta de 3 colores.

La obtención de la imagen se realiza a través de la API de Maps. La API permite recibir parámetros de estilo, con los que puedo reducir a 3 los colores del mapa. Como podéis ver en estas dos imágenes, una de ellas tiene un estilo aplicado y la otra solo tiene quitadas las etiquetas de algunos elementos.



Sin embargo, el png más definido que conseguí exportar contenía 250 colores distintos en una resolución 480x480. Esto se debe a que la intersección entre los colores genera píxeles de colores distintos. Para lidiar con este problema tuve que utilizar el formato de colores HSB. HSB me permite clasificar los colores según su "Hue" o "Tono". Los humanos distinguimos un color por esta característica, no por la cantidad de verde, rojo y azul que tenga un color. Esta es la rueda de colores HSB:



Siendo 0° el color rojo, se genera una relación entre Ángulo - Tono que nos permite identificar automáticamente el tono de un píxel. Gracias a ello, estos 250 colores pueden ser clasificados en amarillo, verde o azul. Ese es el trabajo de este método que convierte ImageBitmap en Matrices de enteros

```
fun generateMapFromImage(img: ImageBitmap, propHandler: PropertiesHandler) : MutableList<MutableList<Int>> {
    val pixelMap = img.toPixelMap()
    val intMap = mutableListOf<MutableList<Int>>()
    for(i in 0 ≤  until  < img.width) {
        intMap.add(mutableListOf())
        for (j in 0 ≤  until  < img.height) {
            val pixel = pixelMap[i, j]
            intMap[i].add(when(RGBtoHSB(pixel.red.toDouble(), pixel.green.toDouble(), pixel.blue.toDouble())[0]) {
                in 30f..75f -> ROAD
                in 75f.. 150f -> TREE
                in 150f..270f -> WATER
                else -> TREE
            })
        }
    }

    return intMap
}
```

Otro de los parámetros de esta clase es el **PropertiesHandler**. Debido a que Android y Escritorio lidian de manera distinta con la lectura de los archivos del almacenamiento, cada plataforma tiene un objeto PropertiesHandler asociado que le permite acceder a los valores almacenados.

ForestFireBuilder

Esta clase implementa el patrón builder para generar una ejecución valida de la simulación. Una simulación valida requiere: un mapa, viento, clima y un clic inicial. FFModel genera un builder al ser instanciado con el que empezar a construir una ejecución.

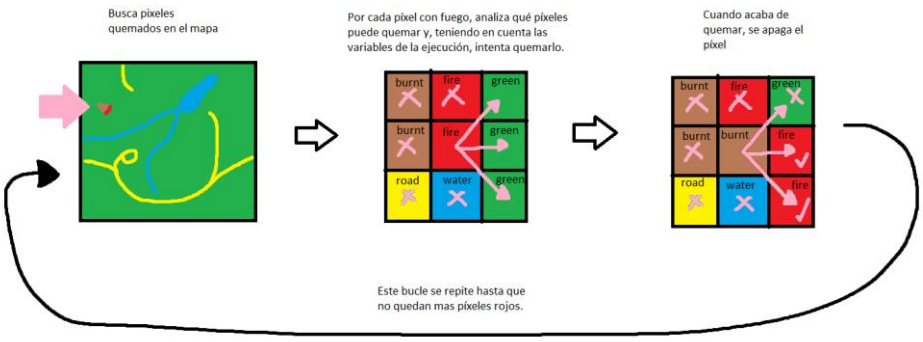
ForestFire

Esta clase contiene la lógica de la ejecución. La función run() ejecuta el código necesario para generar un objeto SimulationResult, que contiene la ejecución pulso a pulso, el mapa sobre el que se ejecuta y el resultado final. El mapa y el resultado final son matrices de enteros. La leyenda de estas matrices es:

- Fuego
- Agua
- Terreno
- Carretera
- Quemado

La ejecución pulso a pulso consiste en un mapa donde se almacenan los pulsos ordenados. Un pulso se modela mediante un entero (orden) y una lista de píxeles modificados. Juntando todos los pulsos en orden se puede generar la ejecución.

Este es un esquema básico de cómo funciona la ejecución.



La primera fase la realiza el método burnMap(). Cada ejecución de este método genera un objeto MapUpdate, el cual modela los pulsos del mapa. Una vez todo el pulso ha sido ejecutado, se guarda en este objeto en forma de Lista de Píxeles actualizados y se inserta en el SimulationResult final.

```
private fun burnMap(): MapUpdate {
    val burnQueue = mutableListOf<Array<Int>>()
    val updatedPixels = mutableListOf<PixelUpdate>()
    for (i in (0 ≤ until < newMap.size)) {
        for (j in (0 ≤ until < newMap[i].size)) {
            if (newMap[i][j] == FIRE)
                burnQueue.add(arrayOf(i, j))
        }
    }
    for (tile in burnQueue) {
        updatedPixels.addAll(expandTileFire(tile[0], tile[1]))
    }
    return MapUpdate(updatedPixels)
}
```

La segunda y tercera fase la realiza el metodo expandTileFire(). Este método devuelve un objeto PixelUpdate por cada píxel que haya cambiado. De esta forma, podemos guardar todos los píxeles que han sido actualizados en un pulso.

```
private fun expandTileFire(i: Int, j: Int): List<PixelUpdate> {
    val listPixelUpdate = mutableListOf<PixelUpdate>()
    for (x in (-1 ≤ .. ≤ 1)) {
        for (y in (-1 ≤ .. ≤ 1)) {
            if (isBurnable( i: i + x,  j: j + y)) {
                if (burnTile( mPos: i + x,  nPos: j + y, coordinatesToWindDirection(x, y)))
                    if (!(x == 0 && y == 0))
                        listPixelUpdate.add(PixelUpdate( x: i + x,  y: j + y, FIRE))
            }
        }
    }
    newMap[i][j] = BURNT
    listPixelUpdate.add(PixelUpdate(i, j, BURNT))
    return listPixelUpdate
}
```

Adicionalmente esta clase incluye funciones que permite detectar si un píxel es inflamable, si está dentro del tablero, decidir si un píxel arde o no a partir de los parámetros, etc.

SimulationResult

Como he mencionado previamente, el resultado de una ejecución se modela con el mapa inicial, un conjunto de evoluciones ordenadas por pulso, un mapa final y los parámetros de la ejecución. La clase encargada de este modelo es SimulationResult.

```
data class SimulationResult(val map:List<List<Int>>, val result:List<List<Int>>, val variation: Map<Int,MapUpdate>, val wind: Wind, val climate: Climate)

data class MapUpdate(val pixelUpdateList: List<PixelUpdate>)

data class PixelUpdate(val x: Int, val y: Int, val value: Int)
```

Esta clase está ideada para ser navegable en orden. Puedes ver el comienzo de una ejecución mirando el mapa inicial y el valor de la clave 0 en variation (que contiene el primer píxel ardiente). Puedes recorrer toda la ejecución pulso por pulso recorriendo variation y aplicando esos cambios al mapa original. Por último, puedes comprobar el resultado final directamente con result.

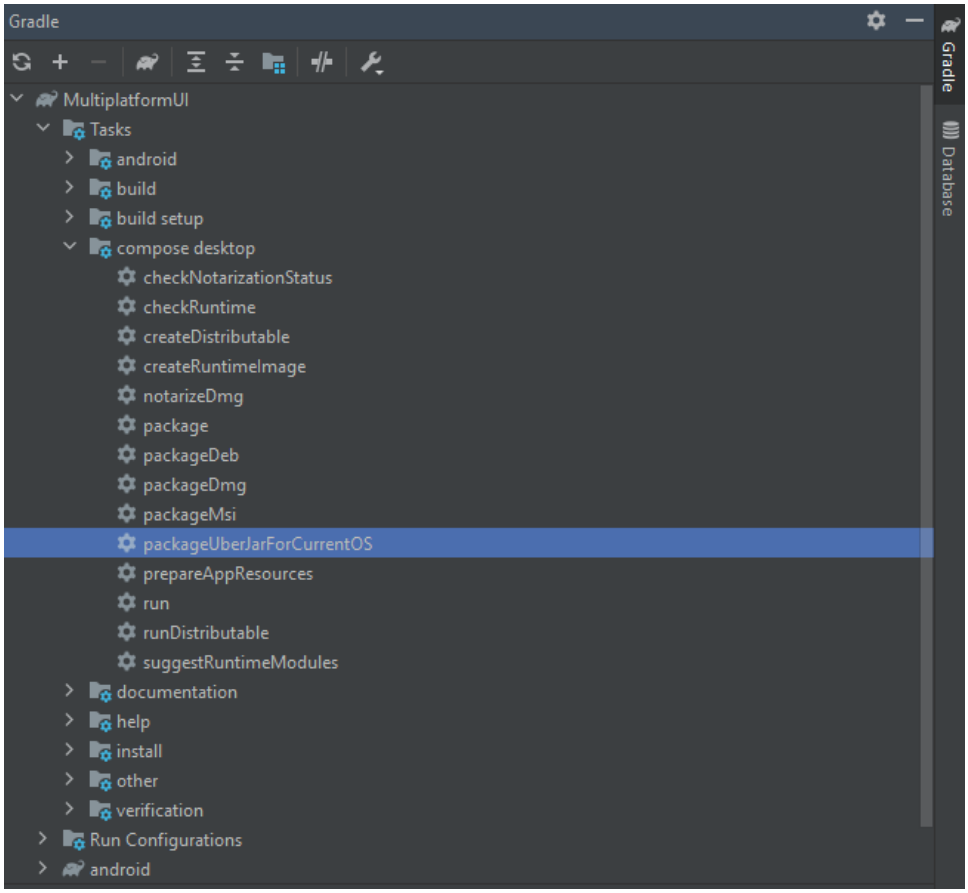
IMPLANTACIÓN

Debido a que las aplicaciones generadas son nativas, solo se realizará el empaquetado de la versión de escritorio.

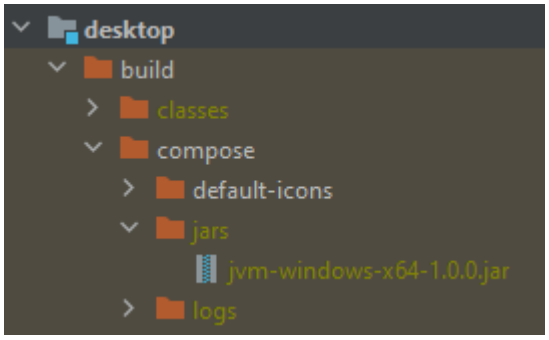
Para ello, utilizamos las tareas predefinidas de Gradle. Citando este enlace en el que hay disponible un tutorial extenso para realizar la distribución en entornos nativos:

> *packageUberJarForCurrentOS is used to create a single jar file, containing all dependencies for current OS. The task is available starting from the M2 release. The task expects compose.desktop.currentOS to be used as a compile/implementation/runtime dependency.*

Esto quiere decir que usemos la opción de gradle "packageUberJarForCurrentOS" para generar un jar con todas las dependencias necesarias. Es importante que la dependencia compose.desktop.currentOS este bien configurada, pero la propia configuración por defecto del proyecto nos establece esto correctamente.



Una vez ejecutamos la tarea de gradle, obtendremos el archivo .jar para su distribución. Este está disponible en la carpeta build de nuestro proyecto desktop.



CONCLUSIONES

CONCLUSIÓN RESPECTO A TECNOLOGÍAS UTILIZADAS

En este proyecto he aprendido a utilizar una nueva tecnología y por primera vez, dicha tecnología estaba aún en fase experimental. He aprendido lo que realmente significa aprender en estas circunstancias: significa que pocas personas han tenido los mismos problemas que tú. Por primera vez leer a fondo la documentación de la tecnología ha sido más efectivo para buscar soluciones que buscar el propio problema online. Esto ha supuesto varios quebraderos de cabeza y ha generado grandes retrasos en mis tiempos estipulados. Sin embargo, también me siento mucho más cómodo en esta tecnología que en ninguna otra que haya aprendido anteriormente, dado que es con diferencia la que más horas de trabajo ha recibido. Mi conclusión personal del proyecto es que quiero seguir utilizando Jetpack Compose Multiplatform para mis proyectos personales y, en un futuro, laborales.

Adicionalmente, agradezco mucho haber decidido empezar el proyecto en Kotlin, puesto que he adquirido mucha soltura en las bases de este lenguaje y su potencia y libertad son inigualables. Quiero seguir programando en este lenguaje para dominar las características más avanzadas que ofrece y poder sacar todo su potencial en mis aplicaciones.

CONCLUSIÓN RESPECTO AL SIMULADOR DE INCENDIOS

Respecto al simulador, finalizo con sensaciones agridulces. Pienso que es un proyecto precioso que consigue atraer la atención del cliente muy rápido y les muestra una ejecución sencilla y satisfactoria, el cual era mi objetivo inicial. Sin embargo, me gustaría haber podido optimizar mejor el algoritmo para que bajo ciertos parámetros la ejecución sea más realista. También me habría gustado poder dedicar más tiempo al estilizado de la aplicación, puesto que su estilo es bastante inicial. Pienso que, con más tiempo de desarrollo, la idea puede llegar a tener un uso real dentro de la de prevención de incendios, y me encantaría llevarla a dicho punto algún día. En resumen, las sensaciones son buenas pero los detalles me inspiran a seguir programando.

TRABAJO FUTURO

API REST

Me gustaría convertir cada ejecución del simulador en algo más. En el estado actual de la aplicación, una ejecución solo sirve para visualizar el resultado. Un servicio API Rest donde los usuarios pueden subir alguna ejecución que desean guardar y posteriormente reproducir es uno de los siguientes pasos para esta aplicación.

Para ello habría que implementar una nueva ventana dentro del simulador que permitiera acceder a las ejecuciones del usuario. También habría que incluir un login/registro.

La base de datos asociada a este servidor sería de tipo NoSQL. Esto se debe a que las ejecuciones se exportan en un objeto sencillo de almacenar en este tipo de bases de datos, pero muy tosco para bases de datos SQL.

WEB AND IOS SUPPORT

Jetpack Compose permite la opción de exportar a Web y a plataformas iOS. El proyecto web de Jetpack Compose necesita una generación de dependencias independiente, al igual que Android y Desktop. iOS por su parte solo requiere ciertas modificaciones a la build del proyecto de

Android. Añadir estas dos tecnologías me permitirían llamar al proyecto "Multiplataforma" con mejor sentido.

BIBLIOGRAFÍA

Map Style Preview (https://snazzymaps.com/editor)

Pagina utilizada para generar los estilos de los mapas.

Maps API (https://developers.google.com/maps/documentation/javascript/overview#maps_map_simple-typescript)

Documentación de la Maps Static API utilizada para recibir los mapas

Google Cloud Platform (https://console.cloud.google.com/)

Esta plataforma da soporte a la API mencionada previamente. Gracias a ella puedo generar claves API, comprobar el tráfico de llamadas a sus API, etc.

Documentación Jetpack Compose Android (https://developer.android.com/jetpack/compose)

Documentación Jetpack Compose Android. La mayoría de los tutoriales sobre componentes los he sacado de esta pagina

Ejemplos Jetpack Compose Multiplatform (https://github.com/JetBrains/compose-jb/tree/master/tutorials/)

Este GitHub contiene varios proyectos de ejemplo en esta tecnología a partir de los cuales pude aprender muchas de las técnicas necesarias para avanzar en el proyecto.

Ejemplos Compose Android (https://github.com/android/compose-samples)

Este GitHub contiene varios proyectos de ejemplo en Android que muestra el uso de muchos componentes de manera avanzada.