# Kazalo

[Kazalo 1](#_Toc199232577)

[Uvod – način za razvoj mobilnih aplikacij 2](#_Toc199232578)

[Android JatPack Compose 5](#_Toc199232579)

[Namestitev Android Studio IDE 5](#_Toc199232580)

[Uporabniški vmesnik (UI) 11](#_Toc199232581)

[Gradle 17](#_Toc199232582)

[UI 18](#_Toc199232583)

[Shrani v datoteko (JSON to File) 19](#_Toc199232584)

[API-s 20](#_Toc199232585)

[GSON 23](#_Toc199232586)

[JSON 24](#_Toc199232587)

[Toast 25](#_Toc199232588)

[Navigacija med ekrani 26](#_Toc199232589)

[FireBase 28](#_Toc199232590)

[Virtual devices 31](#_Toc199232591)

[Težave 32](#_Toc199232592)

[Viri 33](#_Toc199232593)

# 

# Uvod – način za razvoj mobilnih aplikacij

Danes obstaja več različnih **načinov za razvoj mobilnih aplikacij**, vsak s svojimi prednostmi, slabostmi in področji uporabe. Spodaj so glavni pristopi, najbolj pogosti primeri in priporočilo, s čim začeti, če začenjaš iz nič.

## **Glavne metode razvoja mobilnih aplikacij**

### 1. **Native razvoj (tako imenovane domorodne aplikacije – vir:islovar)**

Aplikacije so razvite posebej za posamezno platformo (Android ali iOS), z uporabo uradnih orodij.

#### Android:

* **Jezik**: Kotlin (ali Java)
* **Orodja**: Android Studio +
  + Jetpack Compose ali
  + XML

#### iOS:

* **Jezik**: Swift (ali Objective-C)
* **Orodja**: Xcode + SwiftUI ali UIKit

#### ➕ Prednosti:

* Najboljša zmogljivost in dostop do vseh funkcij naprave.
* Uradna podpora in redne posodobitve.

#### ➖ Slabosti:

* Razvijanje **ločeno za vsako platformo** (več dela, več stroškov).

### 2. **Cross-platform razvoj** (ena koda za več platform)

Aplikacije delijo velik del kode za Android **in** iOS.

#### **Flutter (Google)**

* Jezik: Dart
* Orodje: Android Studio, VS Code
* Hitro rastoč framework z lastnim renderiranjem UI.

#### **React Native (Meta/Facebook)**

* Jezik: JavaScript ali TypeScript
* Uporablja nativne komponente preko bridge-a.

#### **Kotlin Multiplatform (KMP)**

* Jezik: Kotlin
* Deli poslovno logiko med Android/iOS, UI pa je ločen.

#### ➕ Prednosti:

* Enoten razvoj za Android in iOS → **hitrejši razvoj**.
* Nižji stroški razvoja (eno ekipo za več platform).

#### ➖ Slabosti:

* Manjše zmogljivosti pri animacijah ali kompleksnih UI.
* Večji frameworki (več prostora, počasnejši zagon).

### 3. **Hybrid / Web aplikacije**

Gre za mobilne aplikacije, ki se izvajajo v WebView ali so progresivne spletne aplikacije (PWA).

#### **Ionic, Cordova, Capacitor**

* HTML, CSS, JavaScript
* Mobilne aplikacije znotraj spletne lupine.

#### **PWA (Progressive Web Apps)**

* Spletne strani, ki delujejo kot aplikacije (lahko brez povezave).

#### ➕ Prednosti:

* Uporaba spletnih tehnologij, ki jih že veliko ljudi pozna.
* Lahka implementacija osnovnih aplikacij.

#### ➖ Slabosti:

* Slaba integracija s strojno opremo.
* Nižje zmogljivosti in slabša uporabniška izkušnja.

### 4.AppInventor

### učenje programiranja,

### prototipi

### osnovne možnosti

## **Najbolj pogosti danes:**

1. **Jetpack Compose + Kotlin** za Android
2. **SwiftUI + Swift** za iOS
3. **Flutter** za obe platformi
4. **React Native** za JS/TS razvijalce

## **Kaj se učiti, če ne znaš nič?**

### Če želiš razvijati **Android aplikacije**:

* Uči se **Kotlin** + **Jetpack Compose**

uradno podprt, enostaven, čist, moderno zasnovan

### Če želiš delati za **oboje (Android + iOS)**:

* **Flutter** – zelo priljubljen za začetnike, veliko virov in primerov
* **React Native**, če že poznaš JS ali želiš vstop v svet spletnega + mobilnega razvoja

### Če te zanima **splet in aplikacije skupaj**:

* Začni z **HTML/CSS/JavaScript**
* Nato nadaljuj z **PWA** ali **Ionic** za preproste aplikacije

## Zaključek – priporočilo za začetnika:

| **Cilj** | **Priporočeno za učenje** |
| --- | --- |
| Samo Android | Kotlin + Jetpack Compose |
| Android + iOS | Flutter |
| Splet + enostavne aplikacije | HTML/CSS/JS + PWA ali Ionic |
| iOS samo | Swift + SwiftUI |

# Android JatPack Compose

**Jetpack Compose je Googlov priporočeni sodobni paket orodij za gradnjo domordnih uporabniških vmesnikov (UI) na Androidu. Poenostavi in pospeši razvoj UI za Android. Z manj kode, zmogljivimi orodji in intuitivnimi Kotlin API-ji lahko hitro oživiš svojo aplikacijo.**

Prednosti:

* Manj kode
* Intuitivno (deklarativni pristop – z opisom UV)
* Pospeši razvoj

So don't be just a spectator. I would like you to actually **type as I type as well**. So that's part of the whole learning process. And the other thing is **programming can be daunting at times**. And so when you feel overwhelmed, **it's okay to stop rest, go get some fresh air, coffee, tea**, whatever it is that you like to drink to just **rest a little bit**, okay? And once you've rested, you can come back and keep at it.

Native

**Predpogoj za tečaj**

* Osnove programiranja kateregakoli objektnega programskega jezika (Python, Kotlin, Java, C#, C++, JS , PHP, …)

**Kaj bomo obdelali:**

* Osnove programskega jezika Kotllin
* JetPak Compose

# Namestitev Android Studio IDE

[link](https://developer.android.com/studio)

Preveri sistemske zahteve

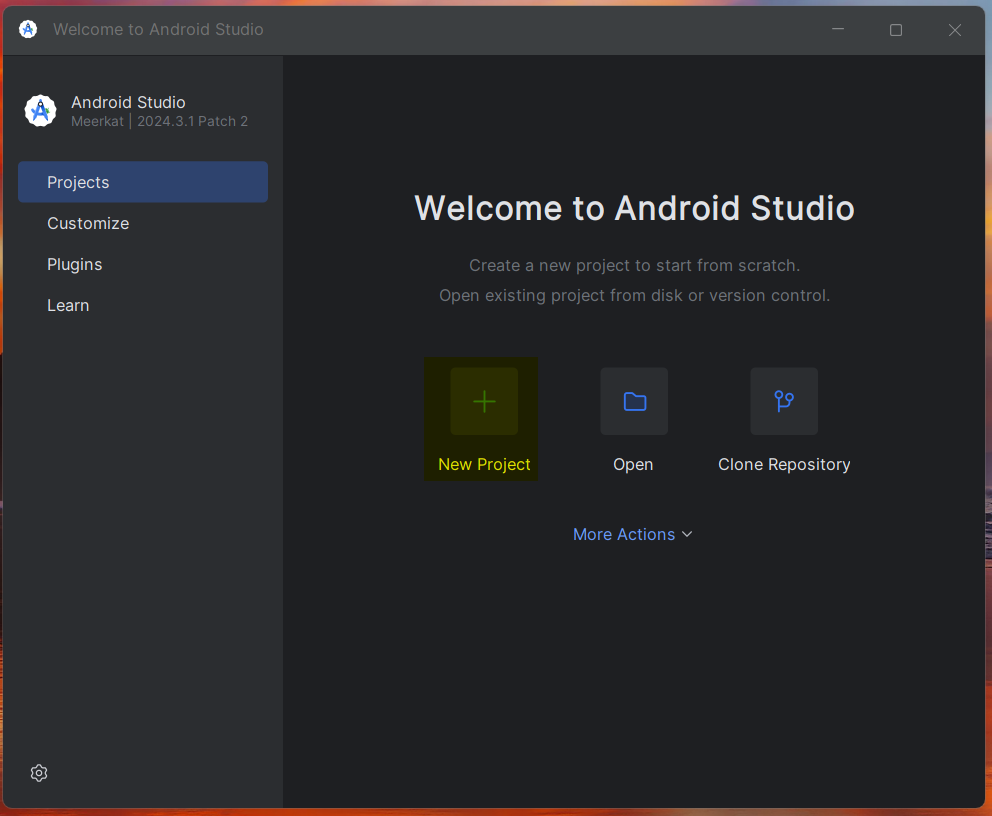
Zelo »zahteven« program

IDE se konstantno spreminja – zna biti rahlo drugačen od verzije do verzije

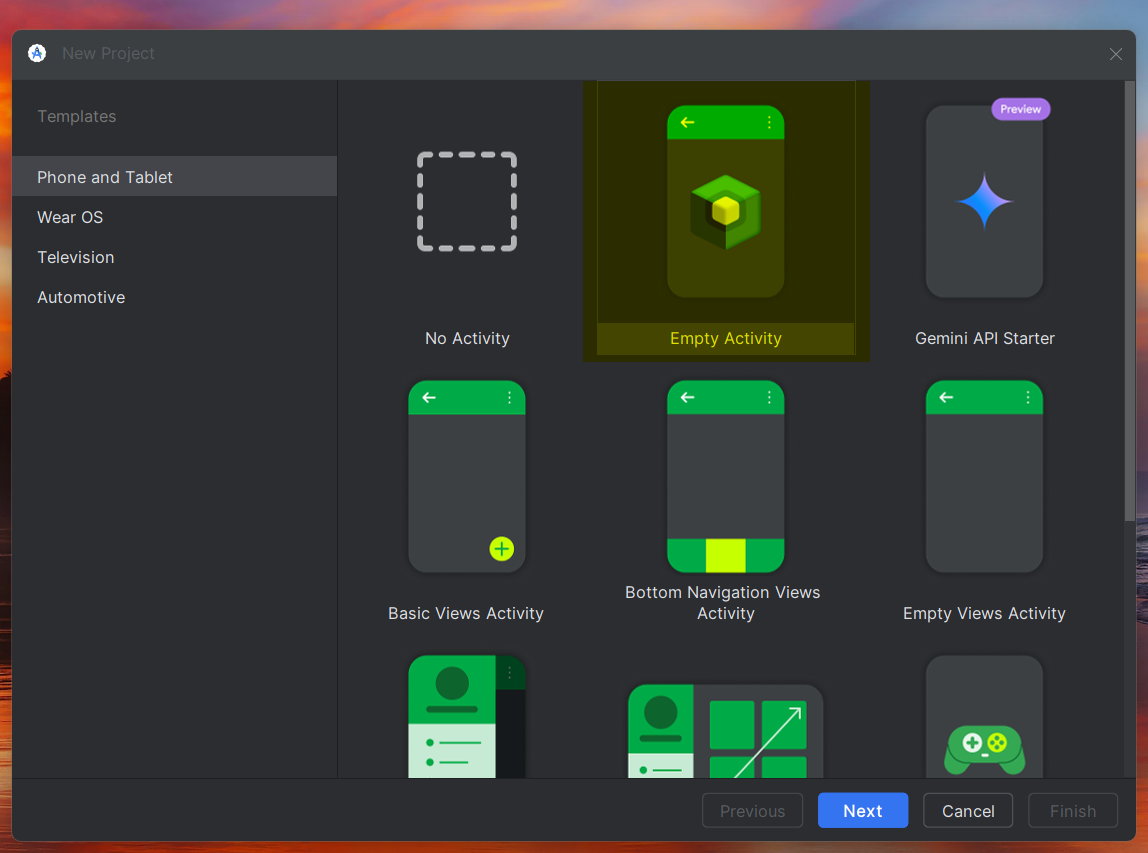
Po navadi namestimo razvojno orodje

Emulator za testiranje aplikacij (za pohitritev potrebujemo HAXM) – ustvari se virtualni računalnik, kjer bo tekla naša mobilna aplikacija

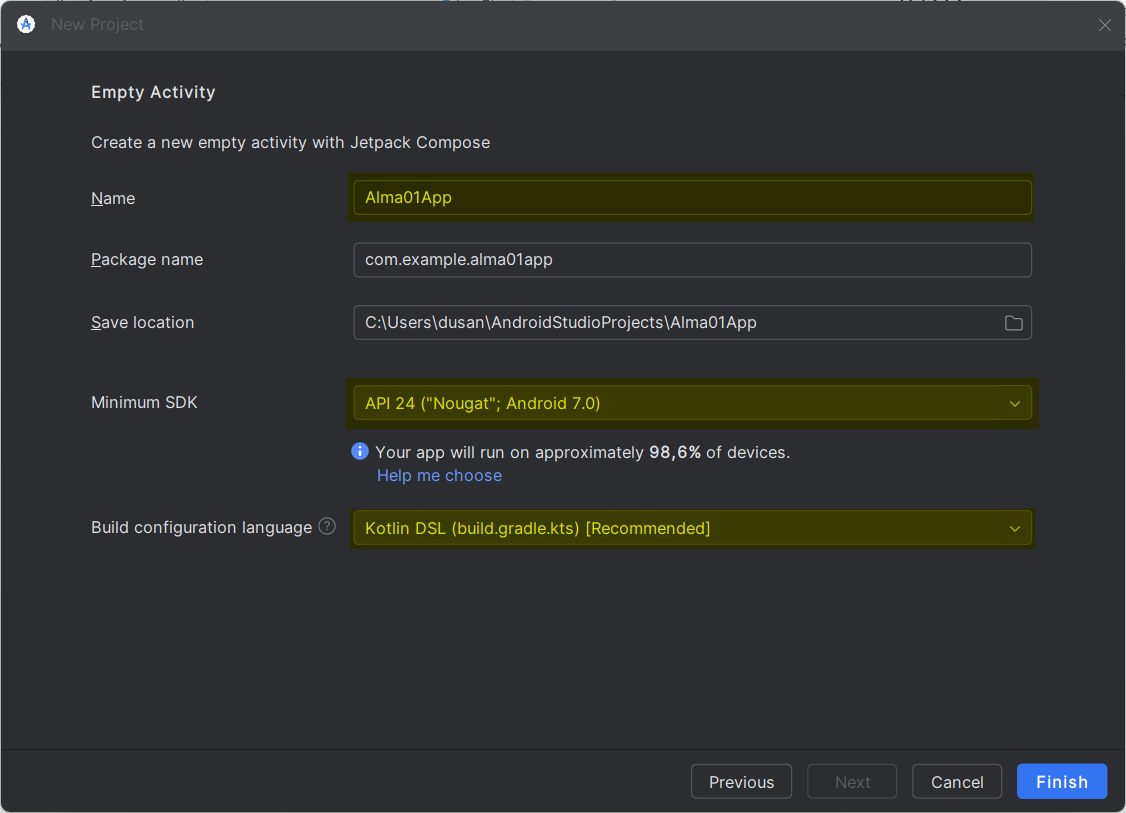
Ustvarjanje novega praznega projekta (New Project)



Izberemo : Empty Activity



New Project



Določimo ime projekta:

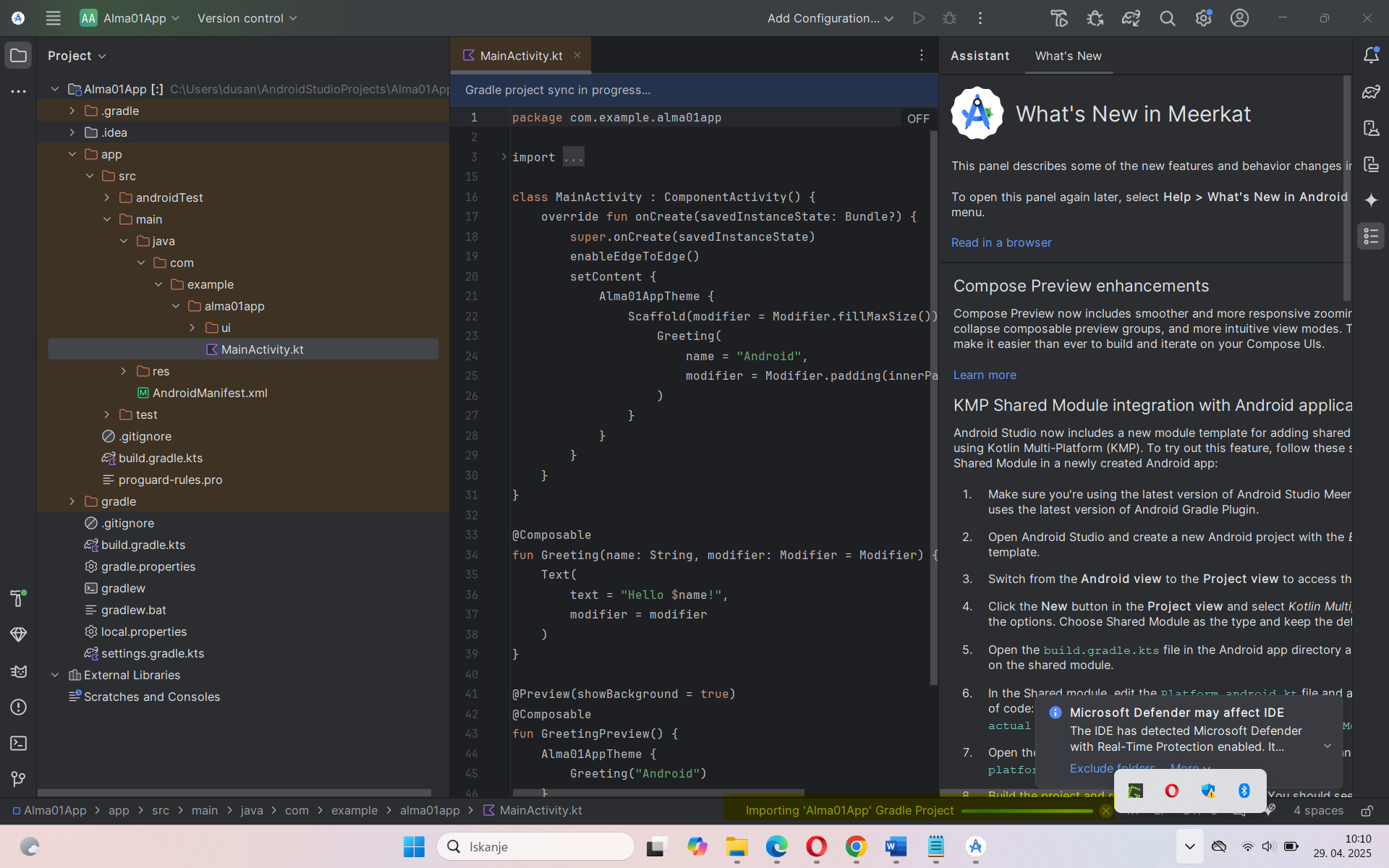
Minimum SDK. Višji (kasnejši) SDK kot bomo izbrali, **manj** naprav bo **kompatibilnih** z njim, lahko bo pa uporabljal **več** novejših **funkcij**.

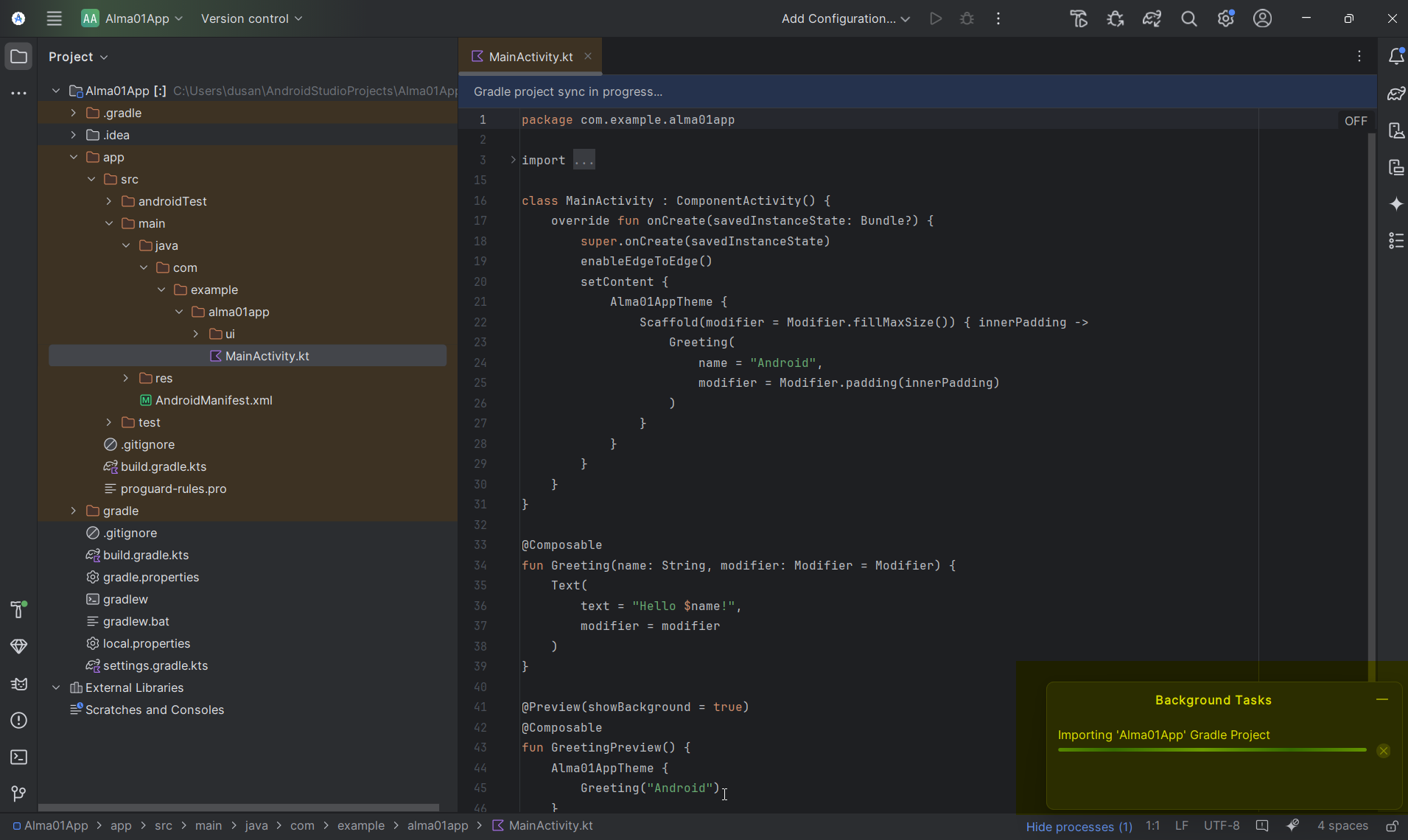
Build configuration language: Kotlin

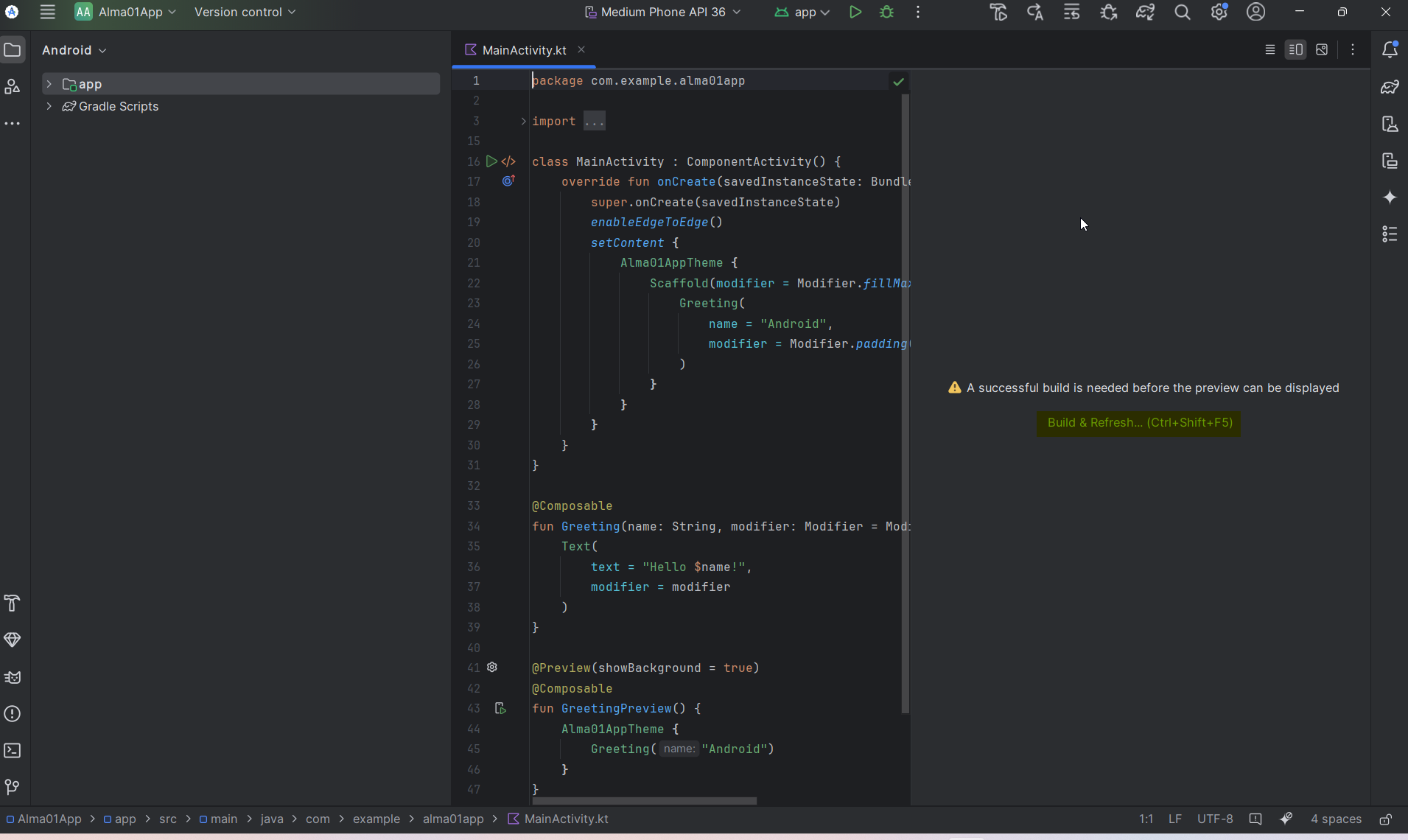
Potrdimo (Next)

Sedaj se naloži nov projekt. To **traja nekaj časa**. V desnem spodnjem delu zaslona je status nalaganja projekta. (importing …) .

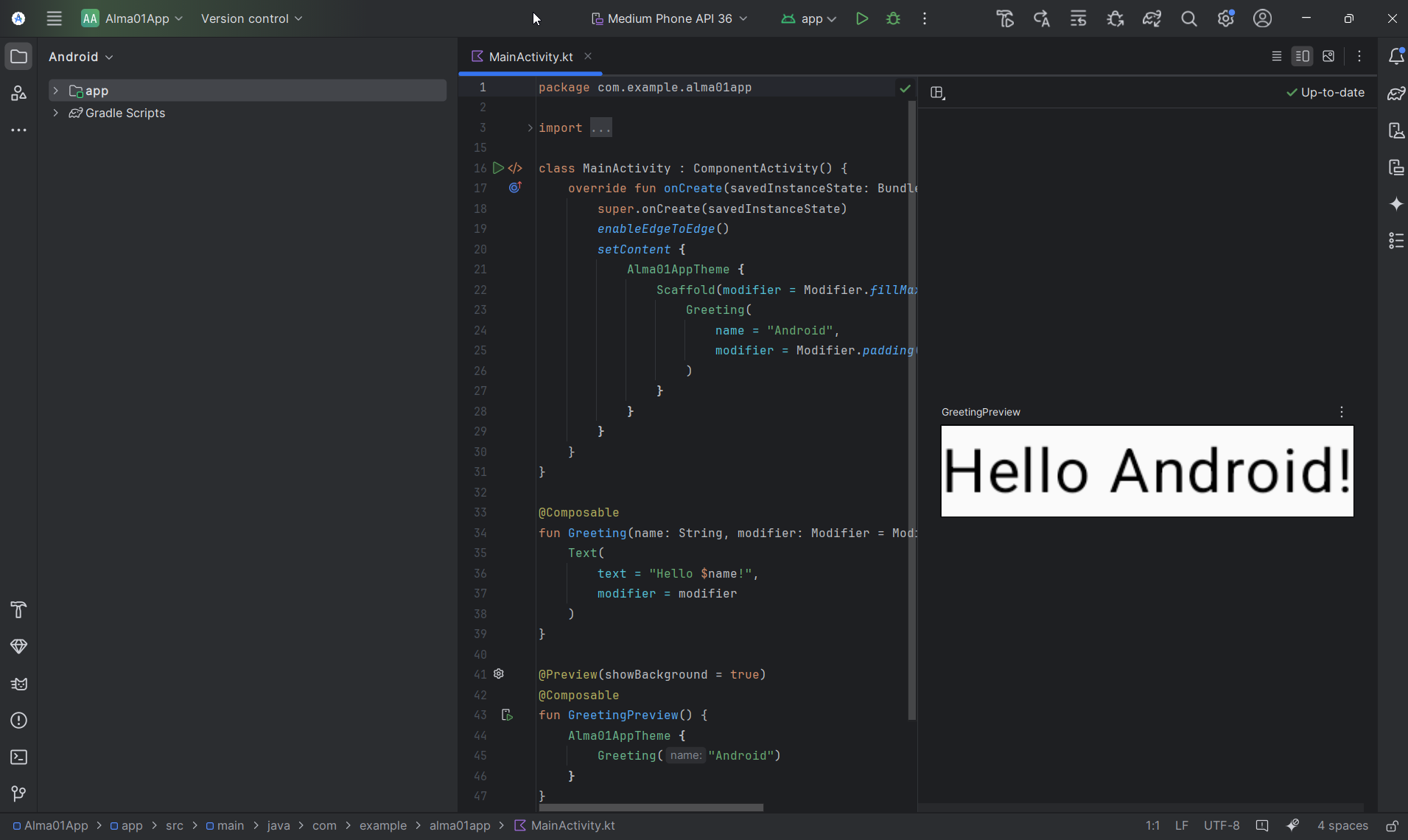
* Projekt se indeksira (indexing)
* Prenaša vse odvisnosti (dependencies)





Ko se projekt zgradi, lahko osvežimo projekt. 

Sledimo navodilom v desnem spodnjem kotu. Morda bomo morali potrditi pravica. Gremo v SDK manager.



Sedaj v programski kodi Kotlin spremenimo tekst znotraj klica funkcije v »Zdravo študent!«. Preverimo spremembo!

To kar vidimo je samo predogled (preview). S JetPack Compose **gradimo** (compose) uporabniški vmesnik. Gradimo ga s programsko kodo. To je bistvo.

Če želimo videti, kako bo aplikaciji v resnici izgledala, jo moramo zagnati v android emulatorju ali na resnični napravi. Emulator moramo najprej namestiti.

Namestimo lahko emulator, ki je del razvojnega okolja, lahko pa namestimo kakšen emulator drugega podjetja . Na primer BlueStacks.

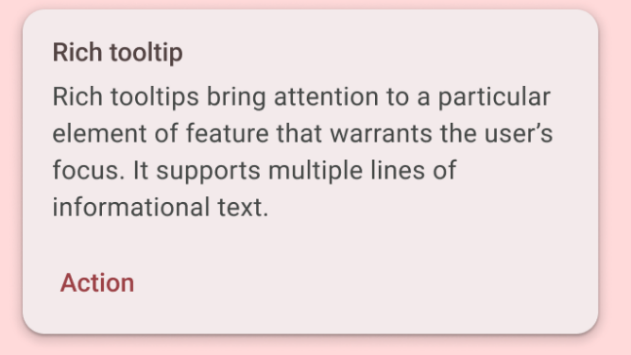
V Android Studio bomo namestimo emulator tako, da gremo v Android Virtual Device manager (AVD).

Ko nameščamo VD, moramo imeti nameščen API, za katerega smo izdelali projekt na začetku. Lahko ga prenesemo, ko nameščamo novi VD.

# Uporabniški vmesnik (UI)

UV aplikacije je to kar **vidimo** na ekranu:

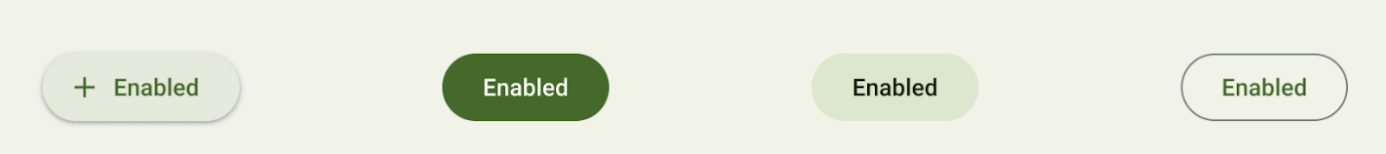
* Besedilo



* Slike
* Vnosna polja



* Gumbi



* Mnogi drugi tipi

**In kako so elementi postavljeni** na ekranu. Z elementi lahko uporabniki komunicirajo.

Vsem omenjenim elementom pravimo komponente uporabniškega vmesnika »UI components«.

## Kaj je Jetpack compose

Jetpack Copmpose je orodje za grajenje uporabniškega vmesnika za Android aplikacije. Compose **poenostavi** in **pohitri** razvoj uporabniških vmesnikov na način, da je:

* manj programske kode ( z manj programske kode naredimo več),
* »močnimi« orodji in
* intuitivnimi značilnostmi programskega jezika Kotlin.

Uporabniški vmesnik gradimo tako, da definiramo množico funkcij, ki jim rečemo **»composable functions«**, ki sprejmejo prametre in opišejo element za uporabniški vmesnik (UI element).

**»Composable function«**

Osnovni gradnik uporabniškega vmesnika v Compose.

Composable function:

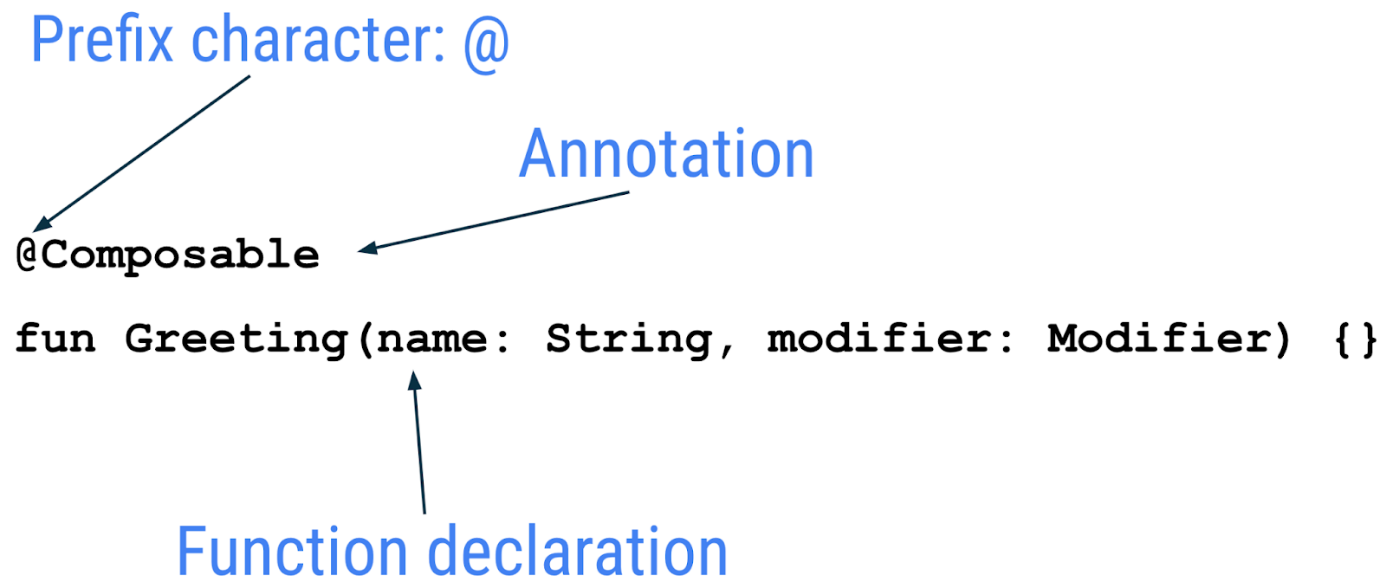
* opiše del uporabniškega vmesnika
* nič ne vrne
* prejme vhode (argumente) in ustvari nekaj, kar se odraža na ekranu

**Anotation (Opombe)**

Opombe so način dodajanja dodatnih informacij o programski kodi. Te informacije pomagajo orodjem, kot je prevajalnik Jetpack Compose, ter drugim razvijalcem, da bolje razumejo kodo aplikacije.

Anotacija se uporabi tako, da pred njenim imenom dodaš znak **@** na začetek deklaracije, ki jo želiš označiti. Različnim elementom kode – kot so **lastnosti**, **funkcije** in **razredi** – lahko dodamo opombo - anotacijo.

|  |
| --- |
| class MainActivity : ComponentActivity() {  override fun onCreate(savedInstanceState: Bundle?) {  super.onCreate(savedInstanceState)  enableEdgeToEdge()  setContent {  Alma02AppTheme {  Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->  Greeting(  name = "Dusan",  modifier = Modifier.padding(innerPadding)  )  }  }  }  }  }  @Composable  fun Greeting(name: String, modifier: Modifier = Modifier) {  Text(  text = "Hello $name!",  modifier = modifier  )  }  @Preview(showBackground = true)  @Composable  fun GreetingPreview() {  Alma02AppTheme {  Greeting("Android and iPhone")  }  } |



**Anotacije s parametri**

Anotacije lahko sprejemajo **parametre**. Ti parametri dajejo dodatne informacije orodjem, ki anotacije obdelujejo.

Spodaj je nekaj primerov anotacije @Preview, **z in brez parametrov**:

|  |
| --- |
| @Preview  @Composable  fun MyComponentPreview() {  MyComponent()  } |

|  |
| --- |
| @Preview(name = "Light Mode", showBackground = true)  @Composable  fun MyLightPreview() {  MyComponent()  } |

Kaj to pomeni?

* @Preview je anotacija, ki pove Android Studiu, da naj **prikaže predogled** funkcije v orodju (Preview pane).
* Parametri kot name in showBackground omogočajo:
  + **poimenovanje predogleda**
  + **dodajanje ozadja**
  + prikaz v **temni ali svetli temi**, itd.

Funkcija Composable je označena z anotacijo @Composable. **Vse composable funkcije morajo imeti to anotacijo.**

Ta anotacija obvesti **prevajalnik** Compose, da je ta funkcija namenjena pretvarjanju podatkov v uporabniški vmesnik (UI).

Naj te spomnimo: **prevajalnik (compiler)** je poseben program, ki vzame kodo, ki si jo napisal, jo pregleda vrstico po vrstico in jo prevede v nekaj, kar računalnik razume – v **strojni jezik**.

**Nekaj opomb o composable funkciji:**

Jetpack Compose je zgrajen okoli **composable funkcij**. Te funkcije ti omogočajo, da uporabniški vmesnik (UI) aplikacije definiraš **programsko**, tako da opišeš, **kako naj izgleda**, namesto da bi se ukvarjal z natančnim postopkom njegove gradnje.

Da ustvariš composable funkcijo, preprosto dodaš anotacijo **@Composable** pred ime funkcije.

|  |
| --- |
| @Preview(showBackground = true) @Composable fun BirthdayCardPreview() {     HappyBirthdayTheme {         Greeting("Android")     } } |

### Composable funkcije lahko kličejo druge composable funkcije

V tem primeru **preview funkcija** kliče drugo composable funkcijo z imenom Greeting().

Opazil boš, da ima ta funkcija poleg @Composable tudi še **drugo anotacijo – @Preview**, ki vsebuje **parameter**.

Ta anotacija @Preview pove Android Studiu, naj funkcijo prikaže v **orodju za predogled** (Preview pane), kar omogoča hitrejši razvoj in sprotno preverjanje izgleda aplikacije.

Več o **parametrih anotacije @Preview** boš spoznal kasneje v tečaju.

### **Imena composable funkcij**

Funkcija v Compose, ki **ne vrača ničesar** in ima anotacijo @Composable, **mora biti poimenovana v Pascal zapisu** (Pascal case).

### 🔤 Kaj je Pascal case?

* **Vsaka beseda se začne z veliko začetnico**.
* Primeri:
  + ✅ GreetingCard()
  + ✅ UserProfileImage()
  + ❌ greetingCard() (to je camelCase)

### 🧠 Pravila poimenovanja composable funkcij:

#### ✅ **MORA biti samostalnik:**

* Dobro: DoneButton()
* Dobro: RoundIcon()

#### ❌ **NE sme biti glagol ali glagolska zveza:**

* Slabo: DrawTextField()  
  → Ne opisuje komponente, ampak dejanje

#### ❌ **NE sme biti “preimenovana predlog” (npr. funkcija, ki temelji na povezavi, prostoru ipd.):**

* Slabo: TextFieldWithLink()

#### ❌ **NE sme biti pridevnik ali prislov:**

* Slabo: Bright(), Outside()  
  → Ne vemo, kaj ta funkcija sploh predstavlja

### ✅ **Lahko vsebuje opisni pridevnik pred samostalnikom:**

* Primer: RoundIcon()  
  → Pridevnik + samostalnik = jasno, kaj komponenta predstavlja

### 📌 Zakaj je to pomembno?

* Sledi konvenciji Compose → bolj **čitljiva**, **prepoznavna** in **standardizirana** koda
* Olajša delo tebi in drugim razvijalcem

Želiš primer napačnih in pravilnih imen v pravi kodi?

# Gradle

Gradle is a **build automation tool** used to **compile**, **build**, **test**, and **deploy** code in many programming languages — most notably Java and Kotlin, and it's the standard build system for Android development.

Gradle is build script. Is a code!

**What does Gradle** do in Android?

In Android projects, Gradle:

* Compiles your Kotlin or Java code.
* Packages your app into an .apk or .aab.
* Downloads and manages dependencies (like Jetpack libraries).
* Handles different build variants (debug/release).
* Lets you automate tasks (e.g., running tests, minifying code).

**Definition**

Task runner.

Runs a right task in right order to achive a certain result.

Results are:

* App runable
* Install on device
* See on device

We do not just run kotlin code and see result on device. There are intermediate steps.

Gradle is a build tool that we use for Android developement to automate the process of buildeing and publishing apps.

Gradle can be written in

* Groovy (old version)
* Kotlin (new version)

Vir: <https://www.youtube.com/watch?v=RCRQlz78wCg&t=116s>

# UI

|  |  |  |
| --- | --- | --- |
| Column |  |  |
| Row |  |  |
| Text |  |  |
| Button |  |  |
| OutlinedTextField |  |  |
| HorizontalDivider |  |  |
| Button |  |  |
| RadioButton |  |  |
| LazyColumn |  |  |

# Shrani v datoteko (JSON to File)

|  |
| --- |
| fun saveStudentsToFile(context: Context, studentiList: List<Student>) {  val jsonString = Json.encodeToString(studentiList)  val file = File(context.filesDir, "student\_list.json")  file.writeText(jsonString) }   fun loadStudentsListFromFile(context: Context): List<Student> {  val file = File(context.filesDir, "student\_list.json")  return if (file.exists()) {  val jsonString = file.readText()  Json.decodeFromString(jsonString)  } else {  emptyList()  } } |

# API-s

Uporabili bomo javni brezplačni API za izdelavo kvizov.

<https://quizapi.io/>

API pomeni Application Programming Interface – torej vmesnik za programsko povezovanje aplikacij. Gre za nabor pravil in orodij, ki omogočajo, da programski deli med seboj komunicirajo.

**Po domače:**

API je kot natakar v restavraciji:

* Gostišče (tvoja aplikacija) pove natakarju (API), kaj želi.
* Natakar (API) gre v kuhinjo (strežnik), prinese podatke (naročeno jed) in jih dostavi nazaj aplikaciji.

**HTTP REST API**

To je najpogostejša oblika API-jev, ki se uporablja prek interneta. Temelji na protokolu HTTP (isti protokol, kot ga uporabljajo spletne strani).

REST pomeni:

REpresentational State Transfer – arhitekturni slog za delo s spletnimi storitvami.

REST API uporablja naslove URL in HTTP metode za izvajanje operacij:

| **HTTP metoda** | **Namen** | **Primer uporabe** |
| --- | --- | --- |
| GET | Pridobi podatke | GET /users → seznam uporabnikov |
| POST | Ustvari nove podatke | POST /quiz → dodaj nov kviz |
| PUT | Posodobi obstoječe podatke | PUT /user/5 → posodobi uporabnika 5 |
| DELETE | Izbriši podatke | DELETE /quiz/3 → izbriši kviz 3 |

**Primer uporabe REST API v Android aplikaciji:**

* Aplikacija želi **prikazati kviz vprašanja**.
* Pošlje **GET** zahtevek na URL npr. https://quizapi.io/api/v1/questions.
* API vrne podatke v **JSON** obliki.
* Aplikacija te podatke **pretvori v objekte** in jih prikaže uporabniku.

**Kako uporabljamo API v praksi?**

V Android aplikaciji uporabiš:

1. URL, HttpURLConnection ali knjižnico kot Retrofit/Volley.
2. Pošlješ zahtevek (GET, POST...).
3. Prebereš odziv (JSON).
4. Pretvoriš JSON v objekte (npr. z Gson).
5. Prikažeš rezultate v aplikaciji.

**Sestavljanje GET zahteve:**

|  |
| --- |
| https://quizapi.io/api/v1/questions?apiKey=YOUR\_API\_KEY&category=docker&difficulty=Medium&limit=10&tags=Docker |

<https://quizapi.io/api-config>

YOUR\_API\_KEY zamenjaj s svojim osebnim ključem (personal token)

<https://quizapi.io/register>

|  |
| --- |
| [  {  "id": 10267,  "question": "You need to troubleshoot a slow-performing Linux server. Which command can help you identify high CPU usage by processes?",  "description": "Explore tools for monitoring and diagnosing performance issues on a Linux server.",  "answers": {  "answer\_a": "htop",  "answer\_b": "ps",  "answer\_c": "uptime",  "answer\_d": "vmstat",  "answer\_e": null,  "answer\_f": null  },  "multiple\_correct\_answers": "false",  "correct\_answers": {  "answer\_a\_correct": "true",  "answer\_b\_correct": "false",  "answer\_c\_correct": "false",  "answer\_d\_correct": "false",  "answer\_e\_correct": "false",  "answer\_f\_correct": "false"  },  "correct\_answer": null,  "explanation": "The 'htop' command provides an interactive process viewer that shows CPU and memory usage.",  "tip": "Use 'htop' for a more user-friendly and dynamic way to monitor system processes.",  "tags": [  {  "name": "Linux"  }  ],  "category": "Linux",  "difficulty": "Easy"  },  {  "id": 3168,  "question": "Which command is used to flush the DNS cache in Ubuntu?",  "description": "Flushing the DNS cache is necessary to resolve issues related to outdated or incorrect DNS information.",  "answers": {  "answer\_a": "Use 'sudo systemd-resolve --flush-caches'",  "answer\_b": "Run 'dnsflush'",  "answer\_c": "Type 'cleardns'",  "answer\_d": "Use 'net-clear-cache'",  "answer\_e": null,  "answer\_f": null  },  "multiple\_correct\_answers": "false",  "correct\_answers": {  "answer\_a\_correct": "true",  "answer\_b\_correct": "false",  "answer\_c\_correct": "false",  "answer\_d\_correct": "false",  "answer\_e\_correct": "false",  "answer\_f\_correct": "false"  },  "correct\_answer": null,  "explanation": "To flush the DNS cache, use the 'sudo systemd-resolve --flush-caches' command.",  "tip": null,  "tags": [  {  "name": "Ubuntu"  }  ],  "category": "Linux",  "difficulty": "Medium"  },    …  ] |

|  |
| --- |
| suspend fun fetchQuizQuestions(context: Context): List<QuizQuestion> = withContext(Dispatchers.IO) {  val apiKey = "lLQaPLOfjLBHPs3r7HldkjWzgP81OxvOpL3CyD0O"  val stVprasanj = 5  val url = URL("https://quizapi.io/api/v1/questions?apiKey=$apiKey&limit=$stVprasanj&category=Linux")  val connection = url.openConnection() as HttpURLConnection  try {  connection.requestMethod = "GET"  connection.connectTimeout = 5000  connection.readTimeout = 5000  // preveri, če je povazava uspešna  if (connection.responseCode == HttpURLConnection.HTTP\_OK) {  // bufferedReader() ... prebere telo HTTP odziva (HTTP response) in vrne String  // use() {} ... tok se zapre ko preberemo String  val input = connection.inputStream.bufferedReader().use { it.readText() }  // parsanje JSON Stringa  Json.decodeFromString<List<QuizQuestion>>(input) // ta vrednsot se vrne!!! – zadnji izraz v funkciji (single-expression function)  } else {  throw RuntimeException("HTTP error code: ${connection.responseCode}")  }  } finally {  connection.disconnect()  }  } |

**Dodatne zahteve**

Dovoljenje dostopa do interneta. V manifest datoteki AndroidManifest.xml

|  |
| --- |
| <?xml version="1.0" encoding="utf-8"?>  <manifest xmlns:android="http://schemas.android.com/apk/res/android"  xmlns:tools="http://schemas.android.com/tools">  <uses-permission android:name="android.permission.INTERNET"/>  <application  android:allowBackup="true"  android:dataExtractionRules="@xml/data\_extraction\_rules"  android:fullBackupContent="@xml/backup\_rules"  android:icon="@mipmap/ic\_launcher"  android:label="@string/app\_name"  android:roundIcon="@mipmap/ic\_launcher\_round"  android:supportsRtl="true"  android:theme="@style/Theme.Vaja07Naloga02quiz"  tools:targetApi="31">  <activity  android:name=".MainActivity"  android:exported="true"  android:label="@string/app\_name"  android:theme="@style/Theme.Vaja07Naloga02quiz">  <intent-filter>  <action android:name="android.intent.action.MAIN" />  <category android:name="android.intent.category.LAUNCHER" />  </intent-filter>  </activity>  </application>  </manifest> |

# JSON

Dodaj »plugin« in »library« v libs.versions.toml

|  |
| --- |
| [versions]  kotlinx-serialization = "1.6.3"  kotlin = "1.9.0"  [plugins]  kotlin-serialization = { id = "org.jetbrains.kotlin.plugin.serialization", version.ref = "kotlin" }  [libraries]  kotlinx-serialization-json = { module = "org.jetbrains.kotlinx:kotlinx-serialization-json", version.ref = "kotlinx-serialization" } |

Dodaj odvisnost (dependency) in plugin v datoteko build.gradle.kts

|  |
| --- |
| plugins {  alias(libs.plugins.kotlin.serialization)  dependencies  {  …  implementation(libs.kotlinx.serialization.json) |

|  |
| --- |
| fun loadStudentsListFromFile(context: Context): List<Student> {  val file = File(context.filesDir, "student\_list.json")  return if (file.exists()) {  val jsonString = file.readText()  Json.decodeFromString(jsonString)  } else {  emptyList()  }  } |

|  |
| --- |
| fun saveStudentsToFile(context: Context, studentiList: List<Student>) {  val jsonString = Json.encodeToString(studentiList)  val file = File(context.filesDir, "student\_list.json")  file.writeText(jsonString)  } |

# Toast

Toast v Androidu uporabimo, kadar želimo uporabniku prikazati kratko sporočilo, ki se prikaže na zaslonu za nekaj sekund in nato samodejno izgine. Toast je nevsiljiv način obveščanja – uporabnik ne rabi ukrepati (ni gumbov, ne prekine dela z aplikacijo).

### Zakaj uporabimo Toast?

* Za hitro povratno informacijo uporabniku:
  + »Podatki shranjeni«
  + »Napaka pri povezavi«
  + »Pritisnili ste gumb«
* Ne želimo prekiniti poteka uporabe aplikacije.

### Kaj moraš vključiti?

1. Context – običajno je to this, applicationContext ali context iz Composable funkcije.
2. Sporočilo (niz besedila).
3. Trajanje prikaza – Toast.LENGTH\_SHORT (2 sekundi) ali Toast.LENGTH\_LONG (3,5 sekunde).

|  |
| --- |
| import android.widget.Toast  fun App() {  val context = LocalContext.current  Toast.makeText(context, "Izpolnite vsa polja", Toast.LENGTH\_SHORT).show()  } |

# Navigacija med ekrani

S funkcijo **NavHost** definiramo graf za navigacijo po ekranih naše mobilne aplikacije. S tem povemo:

* Katere ekrane ima naša aplikacija.
* Kako med njimi navigiramo.

1. Dodati moramo odvisnost v build.gradle (:app)

|  |
| --- |
| implementation("androidx.navigation:navigation-compose:2.7.7") |

1. Vključiti moramo razred

|  |
| --- |
| import androidx.navigation.compose.rememberNavController |

1. Ustvarimo Composable funkcijo s poljubnim imenom. Običajno jo imenujemo MyAppNavHost.
   1. Ustvarimo objekt, ki hrani navigacijo med ekrani in se ohrani med rekompozicijo Ta objekt hrani:
      1. Graf z navigacijo med ekrani
      2. Zgodovino klikov – klikanja med ekrani (back stack)
      3. Trenutno destinacijo - viden ekran
      4. Argumente, ki jih lahko pošljemo »ekranu« (če jih pošiljamo)
      5. Shranjena stanja (če je omogočeno)
   2. Objekt napolnimo s podatki. Uporabimo funkcijo NavHost.
      1. Začetni ekran (ime).
      2. Ekranom določimo imena (String), na katera se bomo kasneje sklicevali in imena funkcij, ki jih bomo klicali!
   3. Definiramo Composable ekrane s parametrov NavController.
      1. Dodamo kodo za premik na drugo stran. Uporabimo funkcijo navigate("ime\_ekrana")

|  |
| --- |
| @Composable  fun MyAppNavHost() {  val ime\_konktrolerja = rememberNavController()  NavHost(navController = ime\_konktrolerja, startDestination = "ime\_prvega\_ekrana") {  composable("ime\_prvega\_ekrana") { Ime\_prve\_funkcije(ime\_konktrolerja) }  composable("ime\_drugega\_ekrana") { Ime\_prve\_funkcije(ime\_konktrolerja) }  ...  }  }  @Composable  fun Ime\_prve\_funkcije (navController: NavController) {  Button(onClick = { navController.navigate("ime\_drugega\_ekrana") }) {  Text("Prestavi se na drugi ekran")  }  }  @Composable  fun Ime\_druge\_funkcije (navController: NavController) {  Button(onClick = { navController.navigate("ime\_prvega\_ekrana") }) {  Text("Prestavi se na prvi ekran")  }  }  … |

|  |
| --- |
| class MainActivity : ComponentActivity() {  override fun onCreate(savedInstanceState: Bundle?) {  super.onCreate(savedInstanceState)  setContent {  MyAppNavHost()  }  }  }  @Composable  fun MyAppNavHost() {  val navC = rememberNavController()  NavHost(navController = navC, startDestination = "home") {  composable("home") { HomeScreen(navC) }  composable("second") { SecondScreen(navC) }  }  }  @Composable  fun HomeScreen(navC: NavController) {  Column(  modifier = Modifier  .statusBarsPadding()  ) {  Text("This is Home Screen")  Button(onClick = { navC.navigate("second") }) {  Text("Go to Second Screen")  }  }  }  @Composable  fun SecondScreen(navC: NavController) {  Column(  modifier = Modifier  .statusBarsPadding()  ) {  Text("This is Second Screen")  Button(onClick = { navC.navigate("home") }) {  Text("Go to home Screen")  }  }  } |

# FireBase

Potrebujemo google račun.

Firebase konzola

**Ustvarjanje projekta**

1. Ustvarimo projekt v konzoli Firebase (Create Project in Firebase Console).
2. Ime projekta.
3. Nič drugega.

**Dodajanje nastavitev za Firebase v našo aplikacijo.**

1. Izberemo projekt
2. Kliknemo na ikono Android (Get started by adding Firebase to your app).
3. Dodamo Android package name
   1. Gradle (app nivo) – build.gradle.kts (Module :app)
   2. Išči applicationId in ga kopiramo v vnosno polje na v konzoli Firebase
4. Kopiramo file google-services.json v root mapo app (Android view)
   1. Konfiguracije vrednosti za Firebase SDK. S temi podatki se naša aplikacija lahko poveže na Firebase in tam na app (project) – ki smo jo ravnokar naredili.
5. Dodajmo Firebase SDK v naš Android projekt (code base)
   1. V datoteko libs.versions.toml dodamo

|  |
| --- |
| [plugins]  google-services = { id = "com.google.gms.google-services", version = "4.4.1" }  [libraries]  firebase-database = { group = "com.google.firebase", name = "firebase-database-ktx", version = "20.3.1" }  firebase-bom = { group = "com.google.firebase", name = "firebase-bom", version = "32.7.2" } |

* 1. V build.gradle.kts (module-level) dodamo

|  |
| --- |
| plugins {  alias(libs.plugins.google.services)  }  dependencies {  implementation(platform(libs.firebase.bom))  implementation(libs.firebase.database)  } |

Great — you’re ready to **move from local file storage to shared Firebase data**, so that all users of your app access the **same list of students** from the cloud.

## ✅ Goal

Replace:

saveStudentsToFile(context, studenti)

loadStudentsListFromFile(context)

With:

saveStudentsToFirebase(studenti)

loadStudentsFromFirebase { ... }

## 🔧 1. Add Firebase to your project

### 🔹 a) Add google-services.json

* In [Firebase Console](https://console.firebase.google.com/):
  + Create project → Add Android app
  + Set your applicationId
  + Download google-services.json and place it in:
  + app/src/main/google-services.json

### 🔹 b) Add to libs.versions.toml

[plugins]

google-services = { id = "com.google.gms.google-services", version = "4.4.1" }

[libraries]

firebase-database = { group = "com.google.firebase", name = "firebase-database-ktx", version = "20.3.1" }

firebase-bom = { group = "com.google.firebase", name = "firebase-bom", version = "32.7.2" }

## 🧩 2. Update build.gradle.kts (module-level)

plugins {

alias(libs.plugins.google.services)

}

dependencies {

implementation(platform(libs.firebase.bom))

implementation(libs.firebase.database)

}

## 🧠 3. Enable Firebase Realtime Database

In Firebase Console:

* Go to "Build > Realtime Database"
* Click "Create database"
* Select test mode (or configure secure rules later)
* Create a default database

## 🧬 4. Modify your save and load functions

### 🔹 Student model must be Firebase-compatible:

@Serializable

data class Student(

val id: Int = 0,

val ime: String = "",

val priimek: String = "",

val spol: String = ""

)

### 🔹 Save to Firebase:

fun saveStudentsToFirebase(studentiList: List<Student>) {

val db = FirebaseDatabase.getInstance().getReference("students\_shared")

db.setValue(studentiList)

}

### 🔹 Load from Firebase:

fun loadStudentsFromFirebase(onLoaded: (List<Student>) -> Unit) {

val db = FirebaseDatabase.getInstance().getReference("students\_shared")

db.get().addOnSuccessListener { snapshot ->

val list = snapshot.children.mapNotNull { it.getValue(Student::class.java) }

onLoaded(list)

}.addOnFailureListener {

onLoaded(emptyList())

}

}

## 🔁 5. Replace usage in StudentiApp

Replace:

saveStudentsToFile(context, studenti)

with:

saveStudentsToFirebase(studenti)

And replace:

val loaded = loadStudentsListFromFile(context)

studenti.clear()

studenti.addAll(loaded)

with:

loadStudentsFromFirebase { loaded ->

studenti.clear()

studenti.addAll(loaded)

}

## ✅ Done!

Now your app:

* Uses Firebase Realtime Database
* Shares the same student list among **all users**
* Requires no local storage or file saving

Would you like to:

* Add real-time syncing (updates live when another user changes it)?
* Restrict who can edit (e.g., authenticated users)?  
  Let me know and I can guide you next.

# Virtual devices

Kako priklopiti Bluestacks na Android studiuo?

1. Dodaj ADB v path, da boš lahko zaganjal ukaz v terminalu. Nahaj se v tej mapi

C:\Users\[user]\AppData\Local\Android\Sdk\platform-tools;

1. Zaženi adb devices
2. Poveži s adb connect 127.0.0.1:5555

# Težave

|  |  |  |
| --- | --- | --- |
| Kako vidim napako | Sporočilo | Rešitev |
| Podčrtan MainActivity | Cannot access 'androidx.core.view.KeyEventDispatcher.Component' which is a supertype of 'com.example.vaja05naloga02seznamtrgovinademo.MainActivity'. Check your module classpath for missing or conflicting dependencies. | File → Sync Project with Gradle Files  Build → Clean Project  Build → Rebuild Project |
| Funkcija podčrtana |  | Postavi na funkcijo  ICTRL + B  Postavi se na definicijo funkcije  Poišči ime paketa |

# Viri

<https://developer.android.com/codelabs/basic-android-kotlin-compose-text-composables?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-compose-unit-1-pathway-3%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-compose-text-composables#1>

Udemy - Android Jetpack Compose: The Comprehensive Bootcamp - [link](https://www.udemy.com/course/kotling-android-jetpack-compose-)