# Kotlin Toolchain

CS 346 Application Development

https://student.cs.uwaterloo.ca/~cs346

# Contents

# Introduction

# Build Systems

A build system is a system that manages the process of delivering software. This includes compilation, linking, testing, packaging and other required steps. Historically, build systems have been created to target specific programming languages e.g., Cargo for Rust, or Bazel for C++.

Characteristics of a useful build system:

- It provides consistency in builds and build results. It ensures that you get the same output every time.
- It is expressive so that you can define any custom tasks e.g., zip a file, run unit tests and check outputs.
- It integrates with other systems so that you can delegate responsibility e.g., remote test under a different OS.
- You can automate the entire build process to avoid user errors.

Kotlin is well-supported by a number of different build systems, including Maven, and Gradle [1]. Recently, JetBrains has released the Kotlin Toolchain, a lightweight declarative build system for Kotlin.

# Kotlin Toolchain

Gradle is the standard build system for Kotlin e.g., used by Google internally. However, it's complex, brittle and it can be challenging to learn.

We're instead going to use the Kotlin Toolchain (KT). It's a standalone command-line application designed to "unify building, formatting, testing, and multiplatform project setup". [2]

- It's designed for cross-platform development i.e., building multiple targets in the same project using KMP.
- It's much simpler to configure than Gradle, with a consistent project structure.
- It has more expressive build output, so you can find and fix build failures more quickly and easily.

> **Tip**
>
> You can use Gradle if you wish, but I recommend the Kotlin Toolkit for course projects. All course samples have been converted to use it!

# Getting Started

# Installation

The first step is to install the Kotlin Toolchain, either as a standalone CLI application, or just as a plugin for IntelliJ IDEA. We recommend installing both.

To install:

1. Visit the Kotlin Toolchain website.
2. Follow the installation instructions.

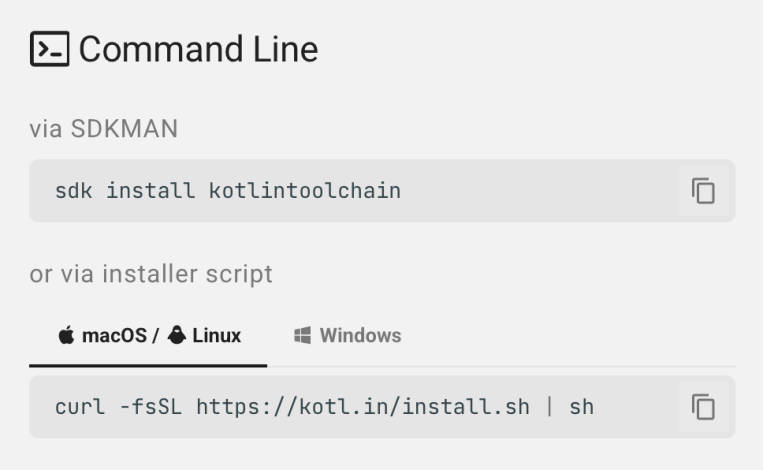



Figure 1: Installation instructions at the time this slide was written. Check the website to make sure you have the most up-to-date instructions.

# Installation

To check your installation, open a shell and type `kotlin`. You should see a number of options.

```
~/Downloads/project $ kotlin
Usage: kotlin [<options>] <command> [<args>]...

Options:
  -v, --version               Show the version and exit
  --root=<path>               Kotlin project root (deprecated)
  --log-level=(debug|info|warn|error|off)
                              Console logging level (default: INFO)
  --shared-cache-dir=<path>   Path to the cache directory shared
                              between all Kotlin projects (default:
                              /Users/jaffe/Library/Caches/JetBrains/Kotlin)
  --shared-caches-root=<path>
                              Path to the cache directory shared
                              between all Kotlin projects (deprecated)
  --build-output=<path>       Root directory for build outputs. By
                              default, this is the build directory
                              under the project root. (deprecated)
  -h, --help                  Show this message and exit

Debugging options:
  --profile           Profile the Kotlin CLI with the Async Profiler
                      (https://github.com/async-profiler/async-profiler)
                      . The path to the snapshot file is determined
                      by --profiler-snapshot-path.
```

Figure 2: The top of the output from the `kotlin` command. We'll explore these tasks in the upcoming slides.

# Creating a Project

Kotlin requires a project: a specific folder structure, with configuration files describing how your source code is structured.

You can create a project from the CLI, or from within IntelliJ. We'll demonstrate each of these options below.

### CLI

To create a new project from the command-line, do the following:

1. Open a shell.
2. Create a project folder.
   - this should be an empty folder.
   - you can also `git clone` an empty project and use that.
3. `cd` into the project folder.
4. Type `kotlin init` and follow the prompts.

# Creating a Project

## IntelliJ

To create a new project from within IntelliJ IDEA:

1. Launch IntelliJ IDEA.
2. `File` -> `New` -> `Project`.
3. Fill in the details in the dialog:

    - Name: name of your project
    - Location: folder e.g., Git working
    - Build System: Kotlin
    - Project Template: Type of project

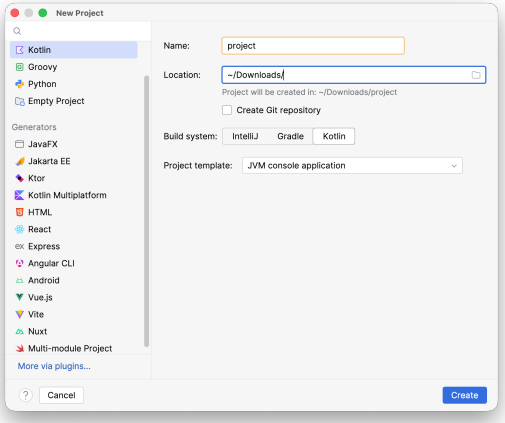



Figure 3: Standard New Project dialog

> **Tip**
>
> It doesn't matter which way you create a project, the resulting folders and config files are the same, based on the `project template`.

# Creating a Project

You have a choice of `Project template` when creating a project. The following types of projects are useful in this course:

Standalone projects

- Android application (Jetpack Compose)
- iOS Application (Compose Multiplatform)
- JVM GUI application (Compose Multiplatform)
- JVM console

Multi-target projects

- Compose Multiplatform application
- Multiplatform CLI application

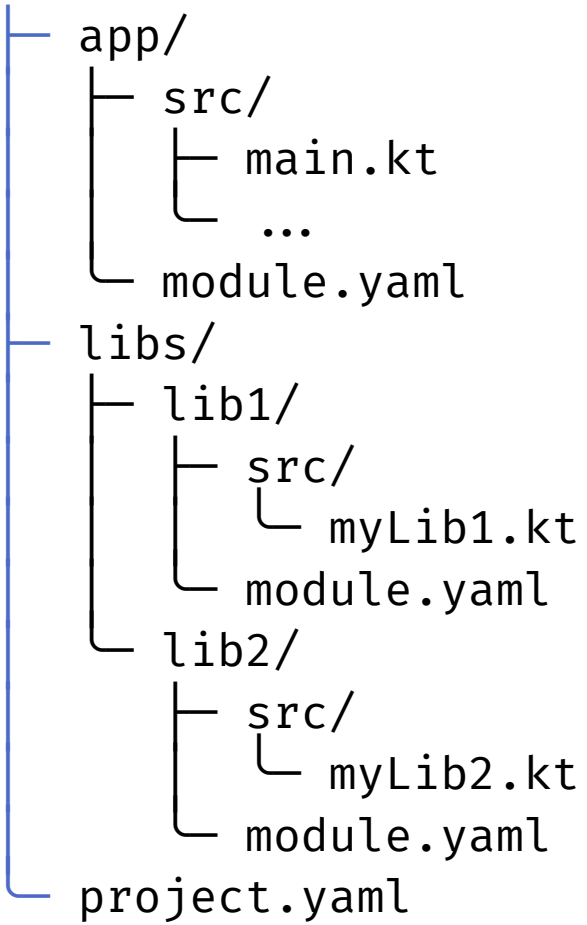
> **Tip**
>
> We will use `JVM console` in the following slides. Later sections will explore more advanced configurations.

# Projects & Modules

A Kotlin project has one or more details, each in its own subdirectory. A top-level `project.yaml` describes the project structure, including a list of modules. Each module has its own `module.yaml` file, source code and unit test directory.

This is a multi-module project:

- `project` is the project folder.
- `app` is the the first module.
  - `src` contains client code
  - `test` contains unit tests
  - `module.yaml` contains client details.
- `libs` contains modules `lib1` and `lib2`.
  - `src` contains all source code
  - `test` contains unit tests
  - `module.yaml` contains server details.
- `project.yaml` contains project details.



# Projects & Modules

## Configuration files

The `project.yaml` file contains all of the project configuration information, including which modules to include.

```
modules:
  - ./app
  - ./libs/lib1
  - ./libs/lib2
```

The `module.yaml` file contains module information i.e., what should be built. It also describes dependencies: how modules are related to one another (in this case, our `app` module imports the other two libs).

```
module.yaml:
  product: jvm/app
  dependencies:
    - ./libs/lib1
    - ./libs/lib2
```

# Build Tasks

Kotlin has a large number of build tasks. Common commands include:

```
$ ./kotlin
Usage: kotlin [<options>] <command> [<args>]...

Commands:
 *build                  Compile and link all code in the project
  check                  Run checks in the project
 *clean                  Remove the projects build output and caches
 *init                   Initialize a new Kotlin project
 *package                Package the project artifacts
  publish                Publish modules to a repository
 *run                    Run your application
 *show                   Show information about the project
  task                   Run a task and its dependencies
 *test                   Run tests in the project
  tool                   Run a tool
  update                 Update the Kotlin Toolchain
```

# Build Tasks

## kotlin build

To build your code from the CLI, type `./kotlin build` from the project folder.

```
~/Downloads/project $ ./kotlin build
00:01.153 INFO  :project:compileJvm       Compiling module 'project' for platform 'jvm'...
00:02.387 INFO  :project:compileJvmTest   Compiling module 'project' for platform 'jvm'...
Build successful
~/Downloads/project $
```

Figure 4: The compiler produces useful build output from the CLI.

To build from IntelliJ IDEA, select `Build` -> `Build Project` from IntelliJ menu.

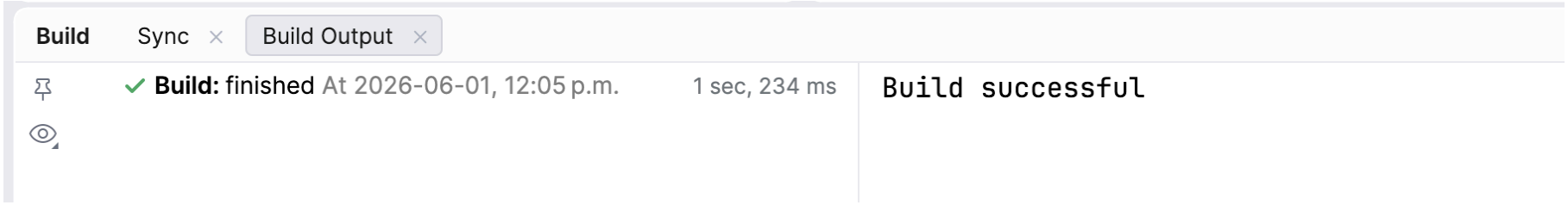



Figure 5: The IDE is functionally the same, with some formatting differences..

# Build Tasks

## kotlin run

To run your project from the CLI, type `./kotlin run` in the project folder.

To run your project in IntelliJ IDEA, open the source file, locate the `main` method, and press the `play` button. Alternatively, you can select `Run > Run` from the IntelliJ menu.

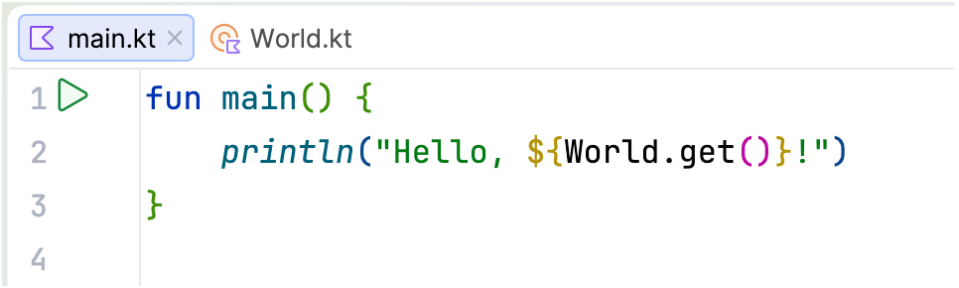



Figure 6: This works for most projects.

> **Note**
>
> If you are working in IntelliJ, it will incrementally compile your code in the background. Running will recompile as-needed before execution.

# Project Templates

# JVM Console Application

This is a standalone console application, with no graphical interface included. To add TUI functionality, you would need to add additional libraries like mosaic or kotter.

To build this type of project, choose `JVM console application` from the project template dropdown.

- The target is a console application for Windows, macOS or Linux.
- The starter project contains just the standard library, no external dependencies and no graphical user interface.

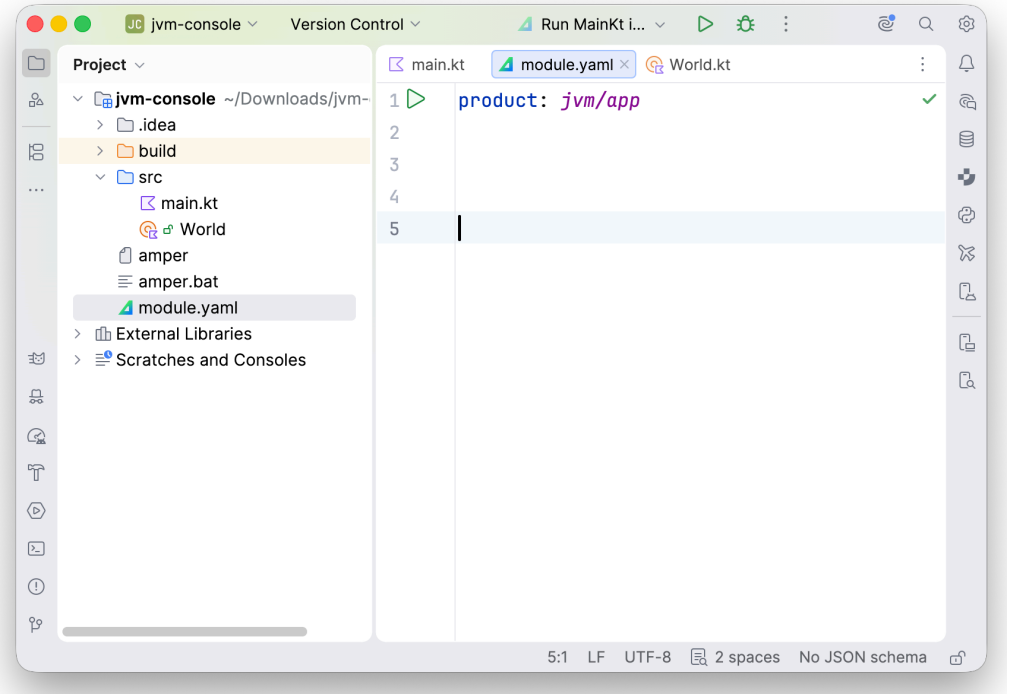



Figure 7: Console applications are the simplest configuration, only needing the `stdlib` to run.

# JVM GUI Application

This is a standalone desktop application, which uses `Compose Multiplatform` for the user interface.

To build this type of project, select `JVM GUI Application (Compose Multiplatform)` from the project template dropdown.

- You can target Windows, Linux or macOS with this project.
- The starter project is similar to the console project, but has imports for the Compose framework.
- Note the dependencies in the `module.yaml` file.

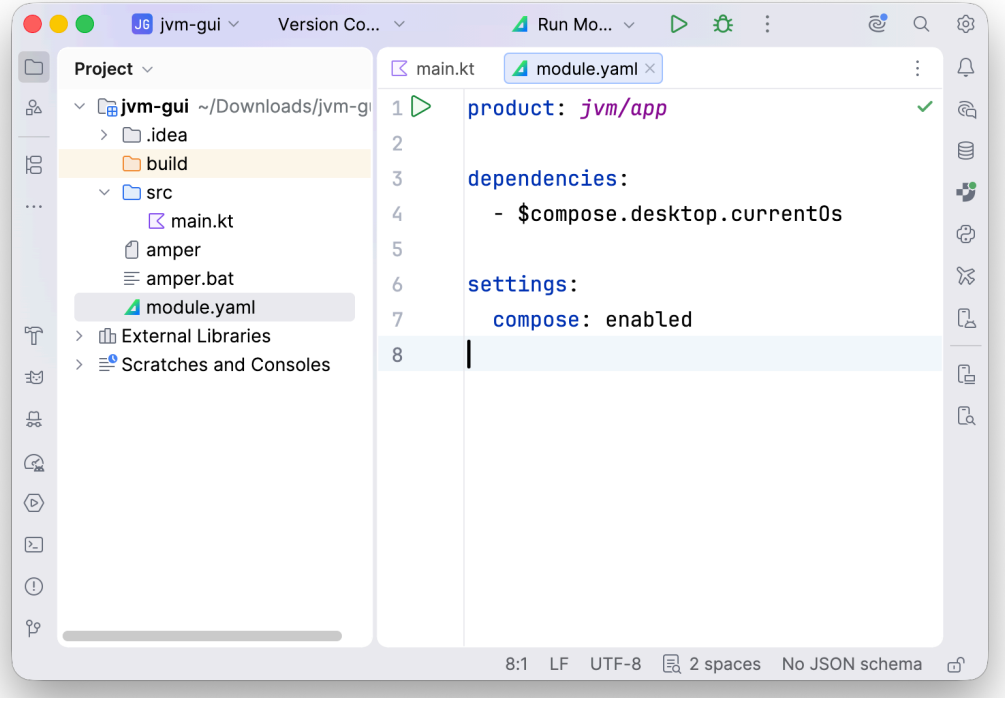



Figure 8: Console applications are the simplest configuration, only needing the `stdlib` to run.

# Android Application

This is a standalone Android application, which builds a native graphical application using Jetpack Compose.

To build this type of project, choose `Android application (Jetpack Compose)` from the project template dropdown.

- This builds a standalone Android executable.
- The dependencies include the Android framework and classes.
- There are extra configuration files which we will discuss later (in the Android lecture).

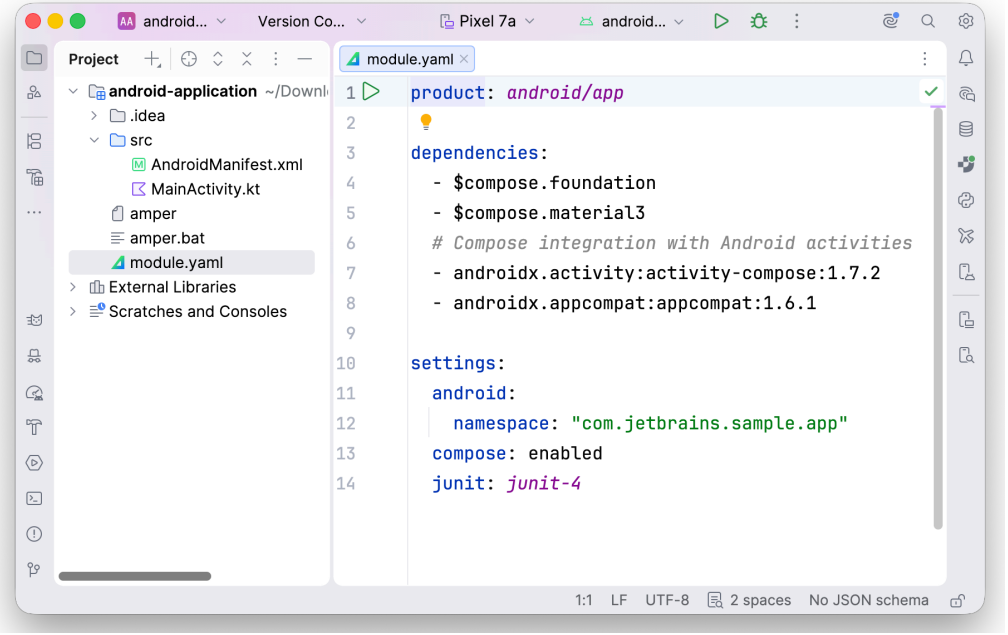



Figure 9: Android applications require more dependencies, and some Android-specific configuration files.

# iOS Application

This is a standalone iOS project, that uses KMP and Compose Multiplatform for a user interface.

To build this type, choose `iOS application (Compose Multiplatform)` from the project template dropdown.

- This builds a standalone iOS application from Kotlin code, using XCode and other Apple build tools.
- You need to have XCode and the Command-Line tools installed.
- If you run into XCode build errors, you *may* need to open the `module.xcodeproj` file in XCode and update settings there.

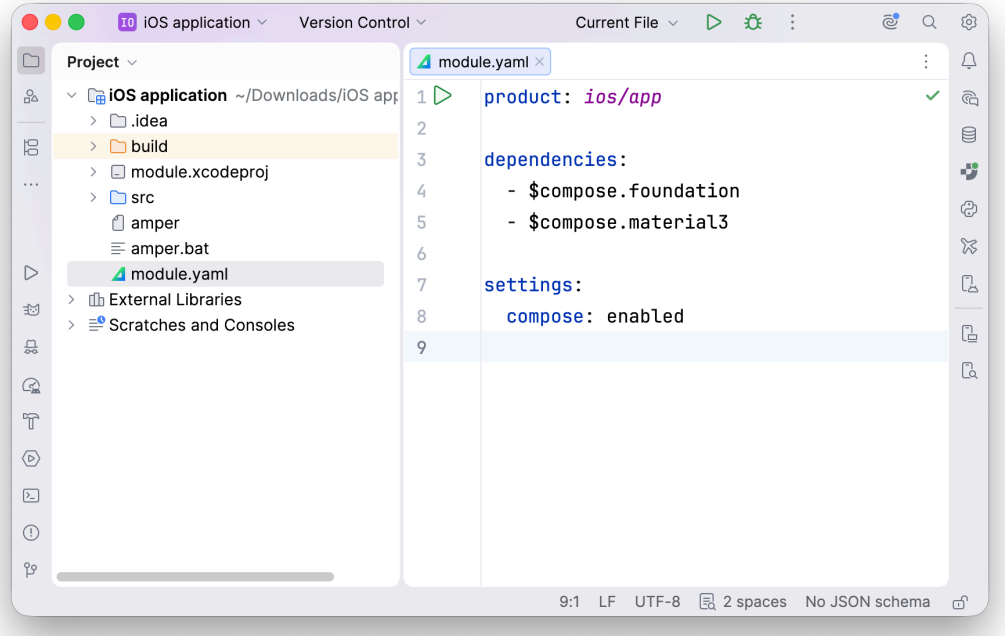



Figure 10: An iOS project uses the XCode build tools to build an iOS executable.

# Compose Multiplatform

Our final project is the most flexible and potentially the most complex. This is a multiplatform project, that uses KMP and Compose Multiplatform to target desktop, Android and iOS in a single project.

To build this type, choose `Compose Multiplatform application` from the project template dropdown.

- A top-level folder for each target that contains the entry-point (i.e., distinct for each platform).
- A single `shared` folder where you put code that you want to reuse.
  - `src` is common shared code.
  - `src@platform` is any platform specific implementation code.
- We'll discuss in the KMP lecture.

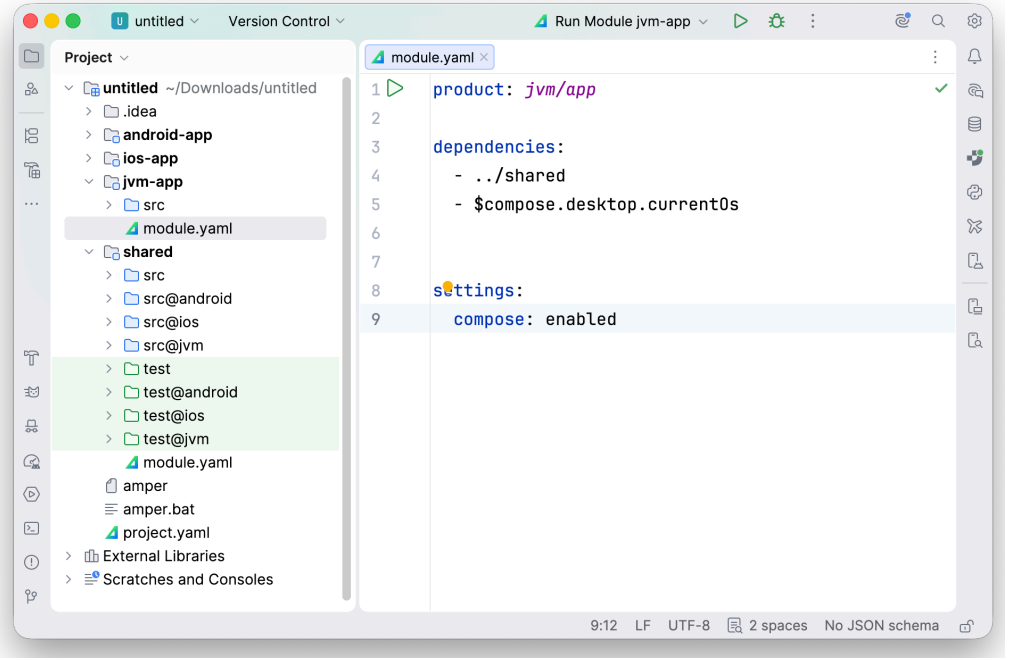



Figure 11: Everything, everywhere, all at once.

# Bibliography

[1] Gradle Inc., “Gradle User Manual.” [Online]. Available: https://docs.gradle.org/current/userguide/userguide.html

[2] JetBrains Inc., “Kotlin Toolchain Homepage.” [Online]. Available: https://kotlin-toolchain.org/