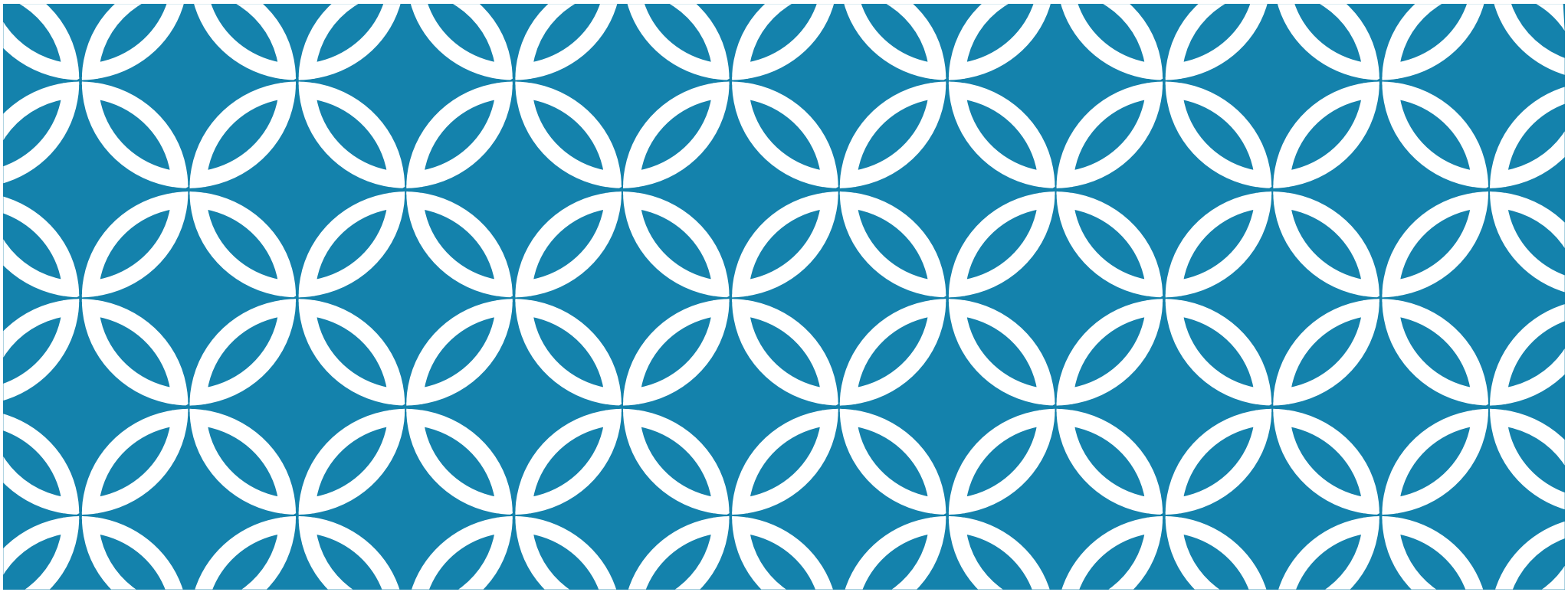
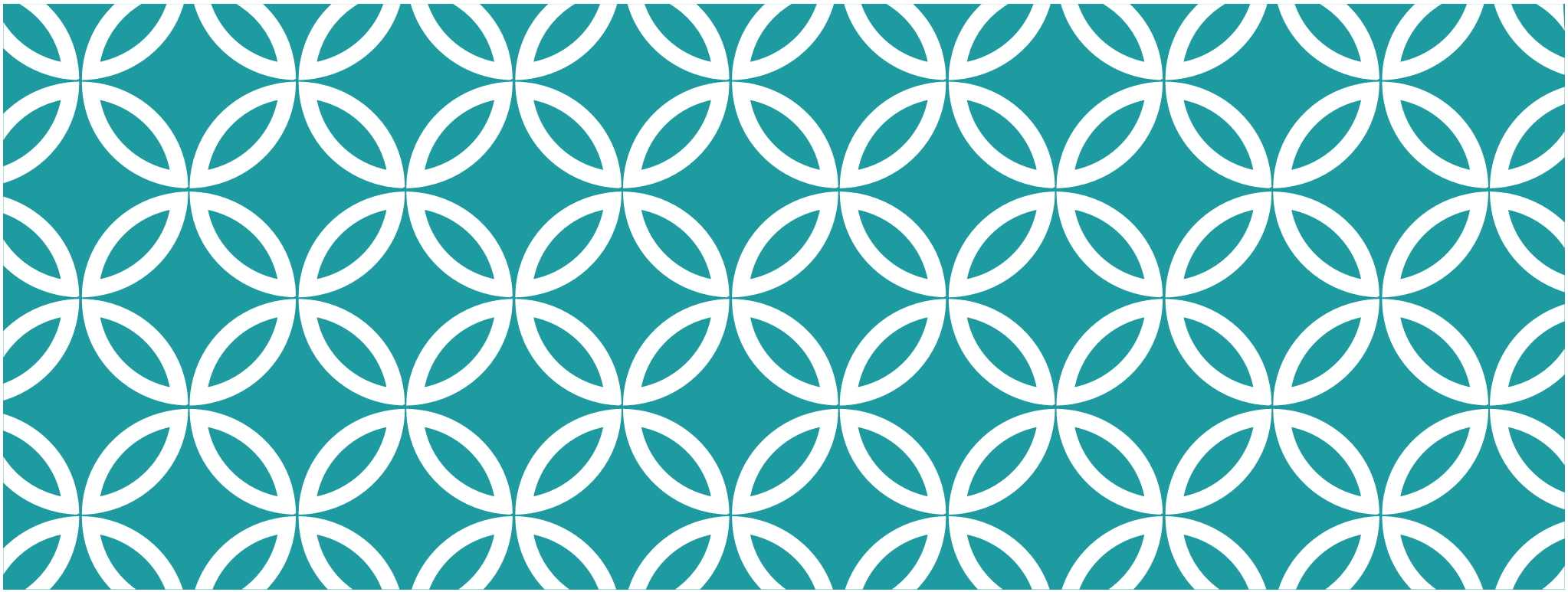


PRACTICES > GRADLE

CS 346: Application Development



INTRODUCTION TO GRADLE

CS 346: Application Development

DEPLOYING APPLICATIONS

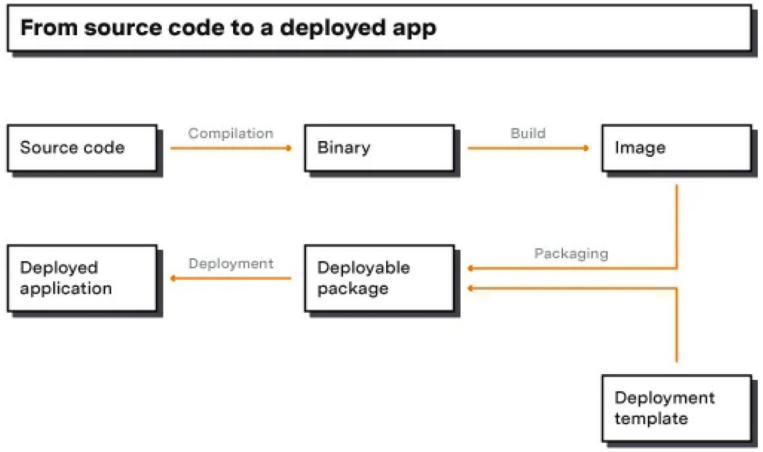
As a developer, your final goal isn't compiled code; it's an application that can be installed and executed by your users.

To achieve this, you will need to:

- Check versions of code, libraries.
- Setup and compile everything.
- Setup deployment and run tests.
- Build installers to distribute to users.

You don't want to do this manually!

- Error prone
- Complex



BUILD SYSTEMS

A **build system** is a system that manages the tasks associated with building software, including compilation, linking, automated testing, packaging.

- Examples: Maven for Java; Cargo for Rust; Cmake/Scons/Bazel for C++.

Characteristics of a *useful* build system:

- It provides **consistency** in builds so that you always get the same artifacts produced.
- The system is **expressive** so that you can define any necessary task.
- You can **automate** much of it to avoid user errors.
- It **integrates with other systems** so that you can report results, or delegate responsibility (e.g. to remote test under a different OS).

WHY WE DON'T USE `MAKE`

`make` is probably not be the best choice for large, complex projects.

- **Build dependencies must be explicitly defined.**
  - Libraries must be present on the build machine, maintained, and defined in your `makefile`.
- **Make is tied to the underlying environment** of the build machine.
  - It's difficult to completely isolate make's runtime behaviour from the underlying environment.
  - e.g., $LIB environment variable to track library location.
- **Performance is poor.** Make doesn't scale well to large projects.
- **The language itself isn't very expressive** and cannot easily be extended.
- **It's very difficult to fully automate** and integrate with other systems.

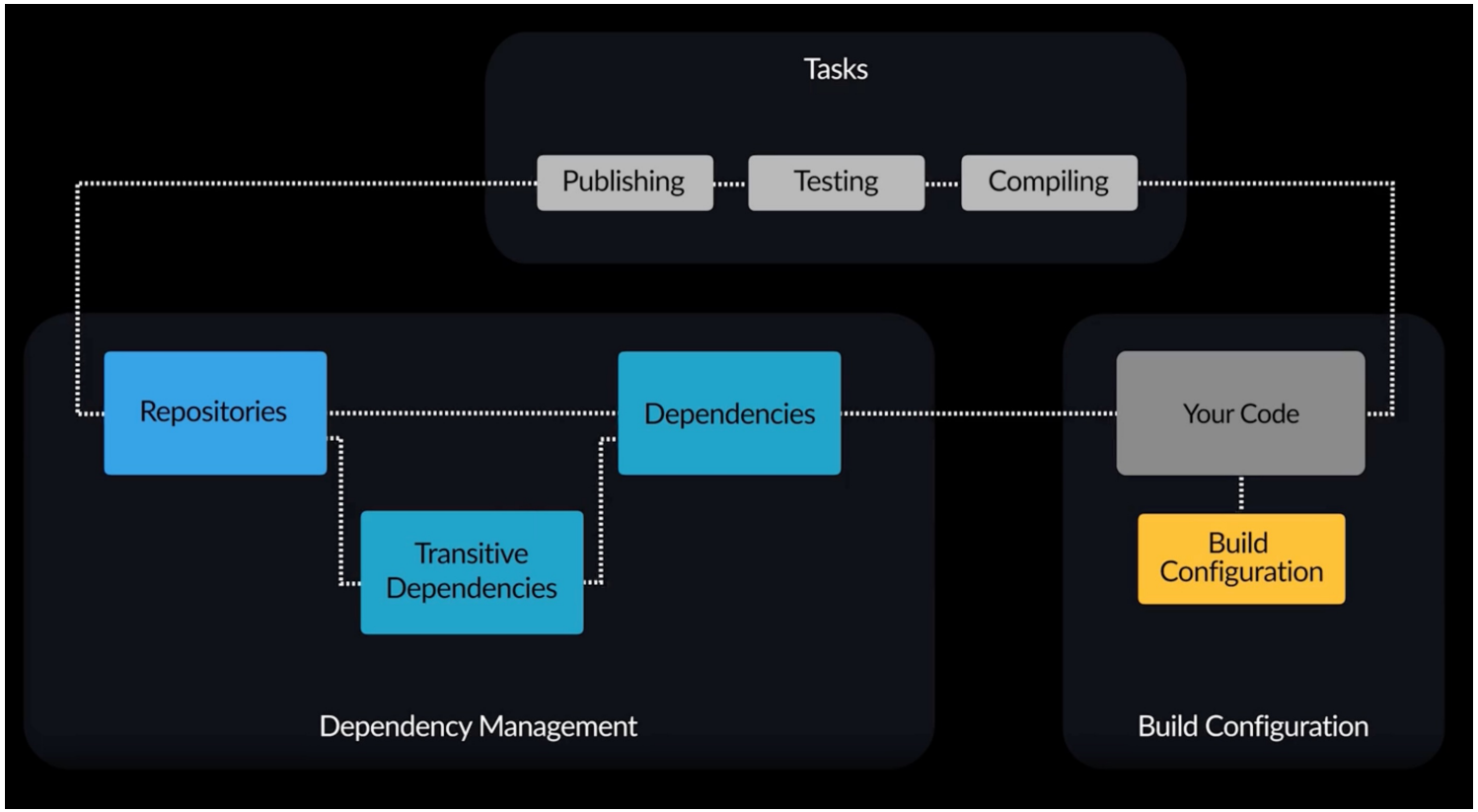
WHY GRADLE?

Gradle is a modern build system for Java/Kotlin.

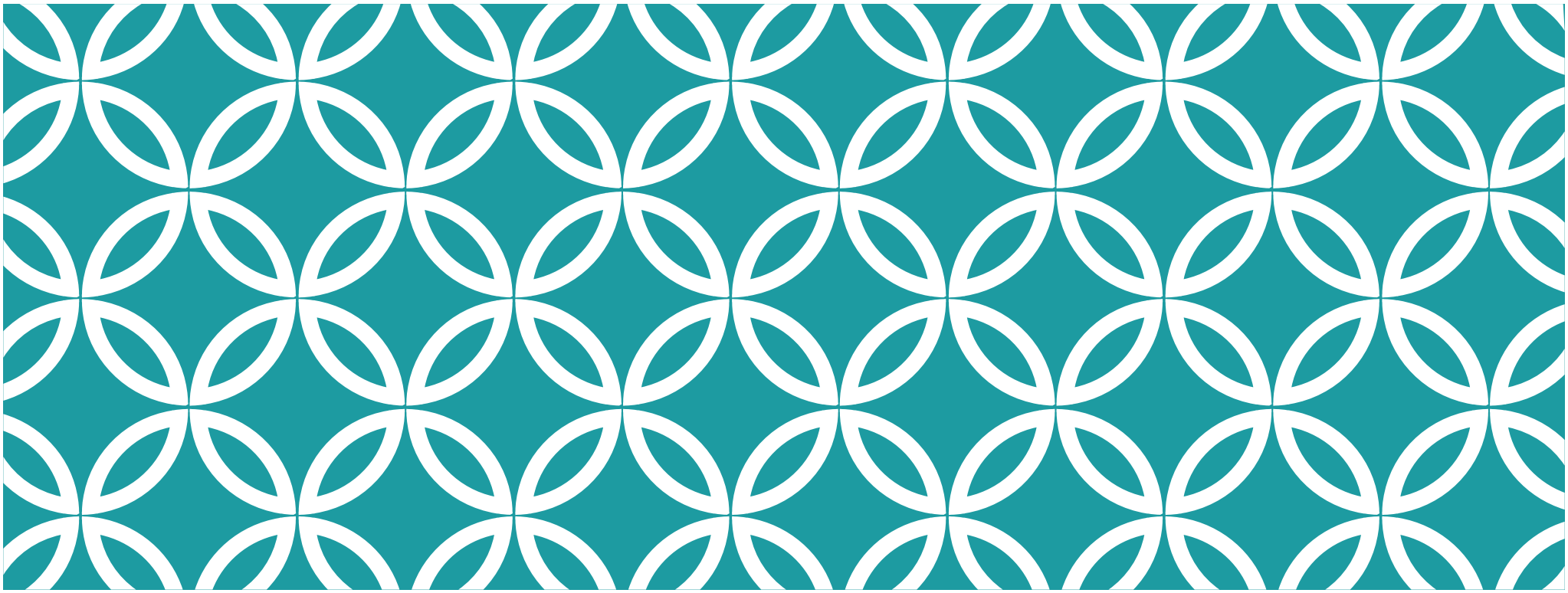
- It's popular in the Kotlin and Java ecosystems.
- It's the official Google-endorsed build tool for Android projects.
- It's cross-platform and programming language agnostic.
- It's open source and has a large community of users.

Three main areas of functionality that it provides:

1. **Managing build tasks**: Built-in support for discrete tasks that you will need to perform. e.g., downloading libraries; compiling code; running unit tests and so on.
2. **Build configuration**: A way to define how these tasks are executed.
3. **Dependency management**: A way to manage external libraries and dependencies.



The pillars of a build system: managing code and dependencies, tasks that define actions to take, and configuration scripts that determine how to run these tasks.



GRADLE > SETUP

CS 346: Application Development

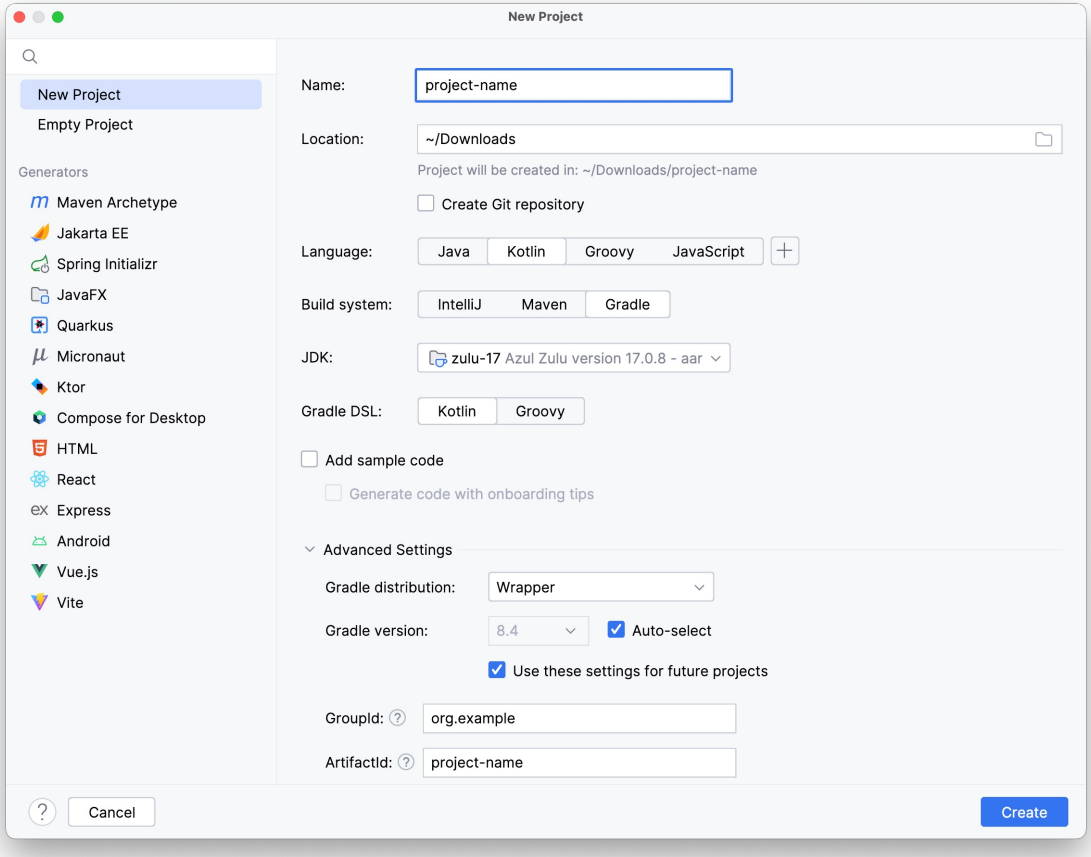
SETUP YOUR PROJECT

Gradle is a command-line application (like Git). It works on Gradle projects that you setup.

A Gradle project is a directory structure and configuration files that define how your source code will be built. You create the Gradle project, and then add your source code (and other assets) to the project.

Project creation can be done in IntelliJ IDEA or Android Studio, or by using the gradle command-line tool.

```
$ gradle init

Select type of build to generate:
  1: Application
  2: Library
  3: Gradle plugin
  4: Basic (build structure only)
Enter selection (default: Application) [1..4]
```



It doesn't matter if you create your project using Gradle or IntelliJ; they produce identical results.

Creating a project in IntelliJ IDEA. Choose Kotlin as your programming language, Gradle for your build system, and Kotlin for your DSL language.

PROJECT STRUCTURE

```
├── build.gradle.kts
├── empty.iml
├── gradle
│   └── wrapper
│       ├── gradle-wrapper.jar
│       └── gradle-wrapper.properties
├── gradle.properties
├── gradlew
├── gradlew.bat
├── settings.gradle.kts
└── src
    ├── main
    │   ├── kotlin
    │   │   └── Main.kt
    │   └── resources
    └── test
        ├── kotlin
        └── resources
```

build.gradle.kts is the main config file.

empty.iml is the IntelliJ config file.

gradle: contains the gradle wrapper config.

gradlew and **gradlew.bat** are wrapper scripts.

settings.gradle.kts is a top-level project config file.

src: contains source code

- `src/main/kotlin` is a source code module
- `src/test/kotlin` is a unit test module

GRADLE WRAPPER

At the top-level of your project's directory structure are two scripts:

`gradlew` for Unix users, and `gradlew.bat` for Windows users

These are *Gradle wrapper scripts.* You can use them to run Gradle tasks without having to install Gradle on your machine.

- Pass them command-line arguments.
- The scripts will download Gradle for you and install it, and then run the commands.

e.g.,

```
$ gradlew build
```

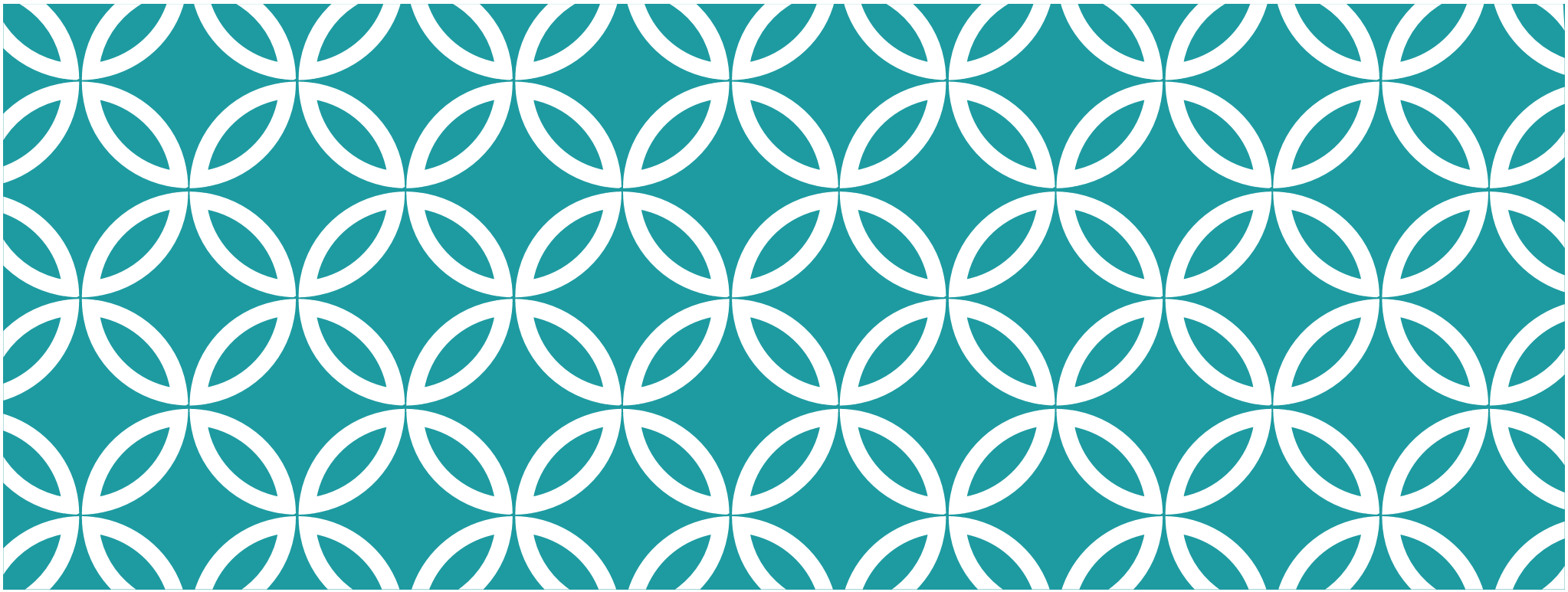
Is this a good idea? Why not just install Gradle?

GRADLE WRAPPER CONFIG

The Gradle project configuration (`gradle/gradle-wrapper.properties`) lists the version of Gradle to be used for your project. It's a text file, with contents (something like) this:

```
distributionBase=GRADLE_USER_HOME
distributionPath=wrapper/dists
distributionUrl=https\://services.gradle.org/distributions/gradle-8.0.2-bin.zip
zipStoreBase=GRADLE_USER_HOME
zipStorePath=wrapper/dists
```

To specify the version of Gradle being used in your project, change the distributionURL line to the correct version e.g., Gradle 8.0.2 in this example.



GRADLE >BUILD TASKS

CS 346: Application Development

WHAT ARE BUILD TASKS?

Projects often have complex build requirements that include a series of steps that need to be performed. For example, you might need to:

1. Compile your source code,
2. Run tests to make sure it built and works properly,
3. Build a distributable package.

You might have additional steps. e.g., generate documentation; deploy to a server.

Any build system needs to support a wide range of steps like this, and it should allow you to define how they will be performed. It should also run them in the correct order.

We call these `build tasks`. Your application probably has many these that need to be run in-order.

RUNNING TASKS

Gradle uses the term **task** to describe a set of related functions that can be applied to a *particular type of project*.

To run Gradle tasks from the command line, use the Gradle wrapper with the appropriate the task name.

Run `./gradlew tasks` to see a list of tasks that are supported in a project.

Execute any of them with the wrapper. e.g., `./gradlew clean`.

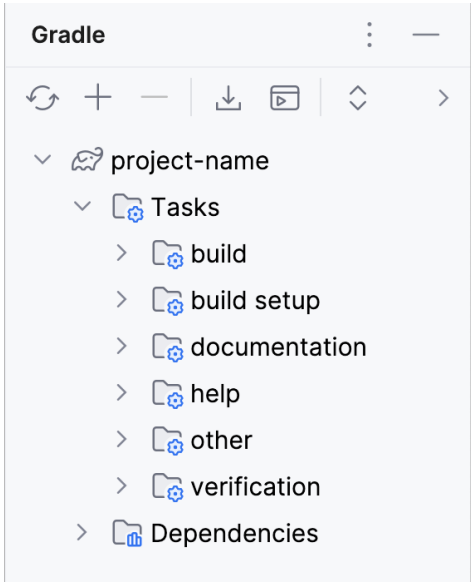
```
$ ./gradlew tasks

> Task :tasks

------------------------------------------------------------
Tasks runnable from root project 'gradle'
------------------------------------------------------------

Application tasks
-----------------
run - Runs this project as a JVM application

Build tasks
-----------
assemble - Assembles the outputs of this project.
build - Assembles and tests this project.
buildDependents - Assembles and tests this project and all projects that depend on it.
buildNeeded - Assembles and tests this project and all projects it depends on.
classes - Assembles main classes.
clean - Deletes the build directory.
jar - Assembles a jar archive containing the classes of the 'main' feature.
testClasses - Assembles test classes.
```

INTELLIJ SUPPORT

Before you panic…. IntelliJ IDEA has built-in support for Gradle. You do not need to memorize hundreds of tasks.

Commonly used tasks include:

- gradlew help
- gradlew tasks
- gradlew clean
- gradlew build
- gradlew run



View > Tool Windows > Gradle will open the Gradle window, listing the supported tasks for your project.

PLUGINS

Gradle comes with a small number of predefined tasks. You will usually need to add additional tasks that are specific to your type of project. We do this via `plugins`.

A plugin is a collection of tasks that have been bundled together to perform a specific function. For example, the java plugin adds tasks for compiling Java code.

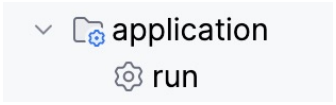
- **Core plugins**: These are the plugins that are included with Gradle by default. They provide functionality that is required by many projects. e.g.,
  - `java` plugin (adds language support) and
  - `application` plugin (adds support for running a console app).
- **Community Plugins**: These are plugins that are created by the community and are not included with Gradle by default. Community plugins can be found in the Gradle Plugin Portal.

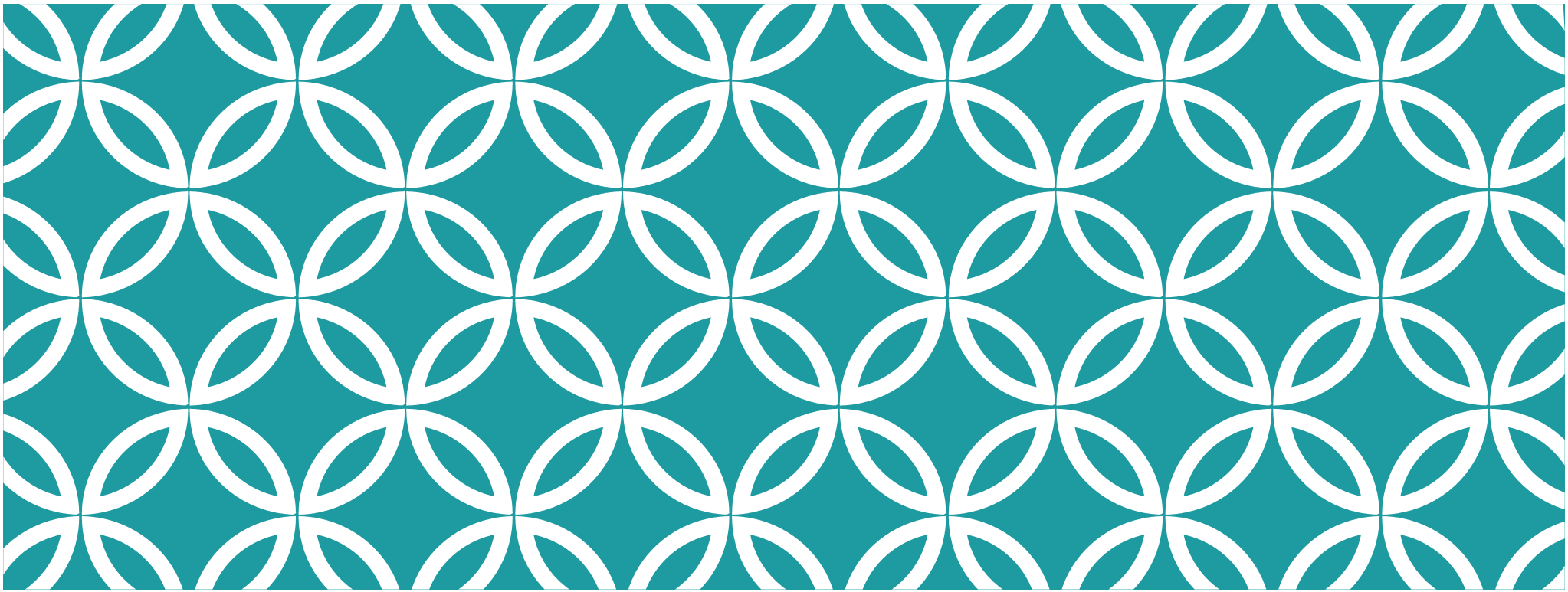
You specify which plugins to load in your `build.gradle.kts` file (which we discuss next).

build.gradle.kts

```
plugins {
   application
    kotlin("jvm") version "2.0.10"
}

application {
    mainClass = "ca.uwaterloo.cs346.MainKt"
}

group = "ca.uwaterloo.cs346"
version = "1.0.0"
```

Application plugin adds the run task.





GRADLE > BUILD CONFIGURATION

CS 346: Application Development

WHAT IS BUILD CONFIG?

Once you have tasks defined, you need some way to configure and control how they are executed.

It's certainly possible to write custom scripts e.g., bash shell scripts to execute these tasks, but they are challenging to maintain and need to be written for each project.

Instead, Gradle provides a way to define tasks in **build configuration files**, and then run them with a single command. This makes it easy to build complex projects and ensures that the build process is consistent across your projects.

- Unlike other configuration-based build systems, Gradle uses a Domain Specific Language (DSL) to define build scripts. You write your scripts in Groovy or Kotlin.
- We'll use Kotlin DSL to write our build scripts (so that both our code and our config files are Kotlin syntax – less to learn!)

CONFIG FILES

```
├── build.gradle.kts
├── empty.iml
├── gradle
│   └── wrapper
│       ├── gradle-wrapper.jar
│       └── gradle-wrapper.properties
├── gradle.properties
├── gradlew
├── gradlew.bat
├── settings.gradle.kts
└── src
    ├── main
    │   ├── kotlin
    │   │   └── Main.kt
    │   └── resources
    └── test
        ├── kotlin
        └── resources
```

There are two main config scripts for your project:

build.gradle.kts is a module specific build config script.

- It is possible to have multiple modules (e.g., app/, service/). Each of these would have its own `build.gradle.kts` file that was specific to that type of module.
- This example has a single module, at the *root*.

settings.gradle.kts is a project config file.

- It contains settings that apply to all modules.

SETTINGS.GRADLE.KTS

This is the top-level configuration file. You don't need to modify this for single-target projects.

```
// list any plugins that you want to use across all modules
plugins {
    id("org.gradle.toolchains.foojay-resolver-convention") version "0.5.0"
}

// top-level descriptive name
rootProject.name = "project-name"
```

settings.gradle.kts

BUILD.GRADLE.KTS

This is the detailed build configuration. You might need to modify this file to:

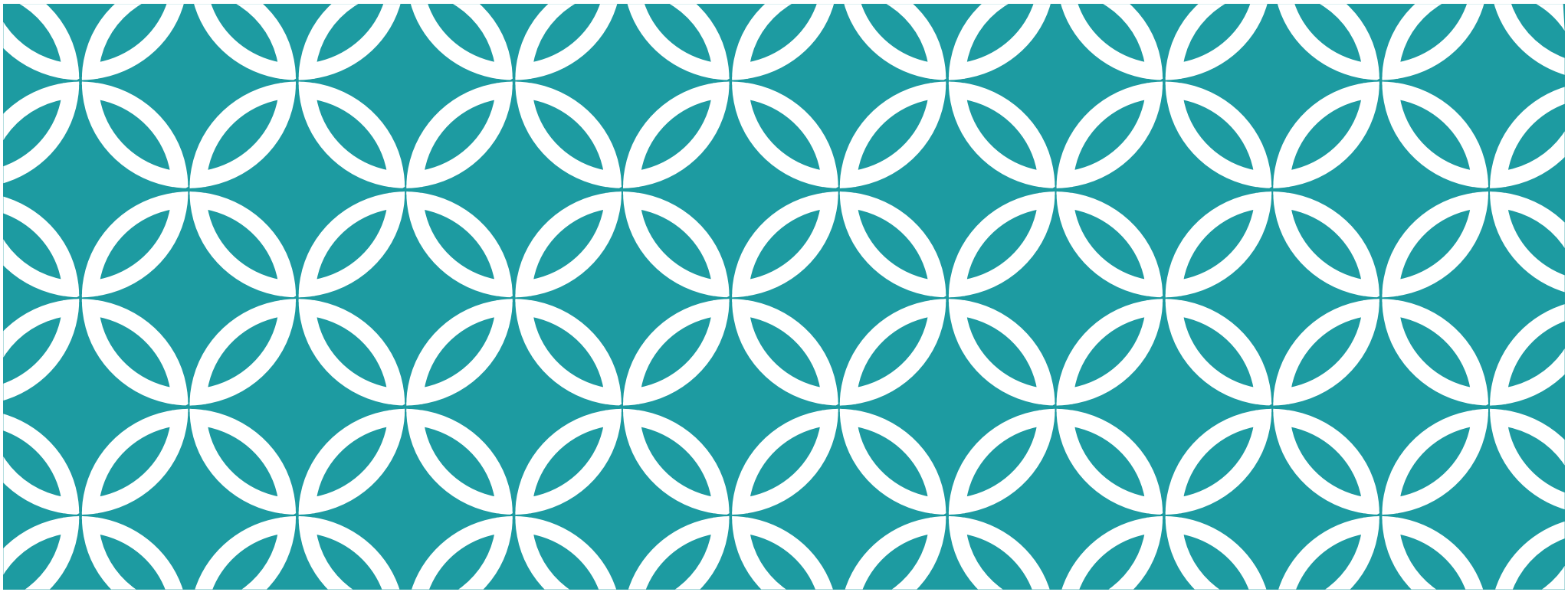
- Add a new dependency (i.e. library)
- Add a new plugin (i.e. set of custom tasks)
- Update the version number of a product release (version below).

Don't expect to create the perfect config file right-away.

- Start with the one generated by IntelliJ IDEA (or gradle init)
- Modify as you add dependencies or make changes.

```
// needed for desktop
plugins {
    kotlin("jvm") version "2.0.10"
}

// product release info
group = "org.example"
version = "1.0.0"

// location to find libraries
repositories {
    mavenCentral()
}

// add libraries here
dependencies {
    testImplementation(`org.jetbrains.kotlin:kotlin-test`)
}

tasks.test {
    useJUnitPlatform()
}

// java version
kotlin {
    jvmToolchain(21)
}
```

build.gradle.kts



GRADLE > DEPENDENCIES

CS 346: Application Development

WHAT ARE DEPENDENCIES?

When we write software, we often rely on external libraries to provide functionality that we don't want to write ourselves. e.g., networking, user interfaces. These libraries are dependencies of our application.

A large challenge of any build system is managing these dependencies properly. i.e.

- Make sure that you have the correct version of a library,
- Include dependencies that library might need (called *transitive dependencies*).
- Make sure that the library is compatible with the rest of your software, and that it doesn't introduce any security vulnerabilities.

In Gradle, you specify your dependencies in your build scripts.

- Gradle will download them from an online repository as part of your build process.

SIDEBAR: REPOSITORIES

A **repository** is a location where libraries are stored and made available; these can be private (e.g. hosted in your company) or public (e.g. hosted and made available to everyone).

Typically, a repository will offer a large collection of libraries across many years of releases, so that a package manager is able to request a specific version of a library and all its dependencies.

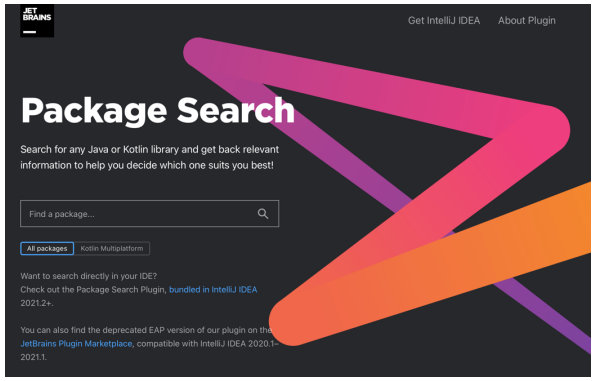
The most popular Java/Kotlin repository is mavenCentral, and we'll use it with Gradle to import any external dependencies that we might require.

FINDING DEPENDENCIES

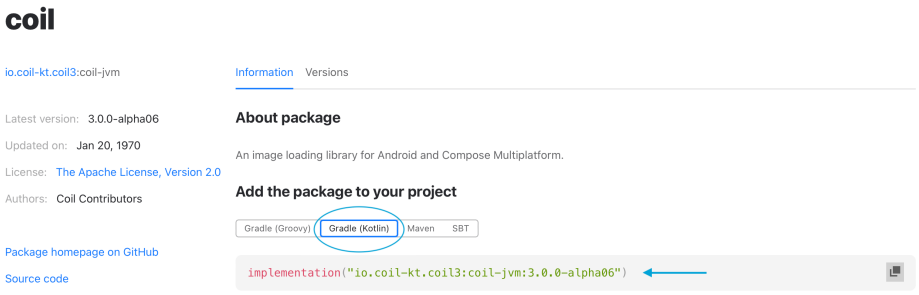
You can search Maven Central, or use a package manager like this one.

This is just a nicer front-end to the actual repositories e.g., Maven Central.

Each package information page will include the details of how to import the package into your Gradle project.



JetBrains Package Search lists all libraries.



Library details include information that you need to use it.

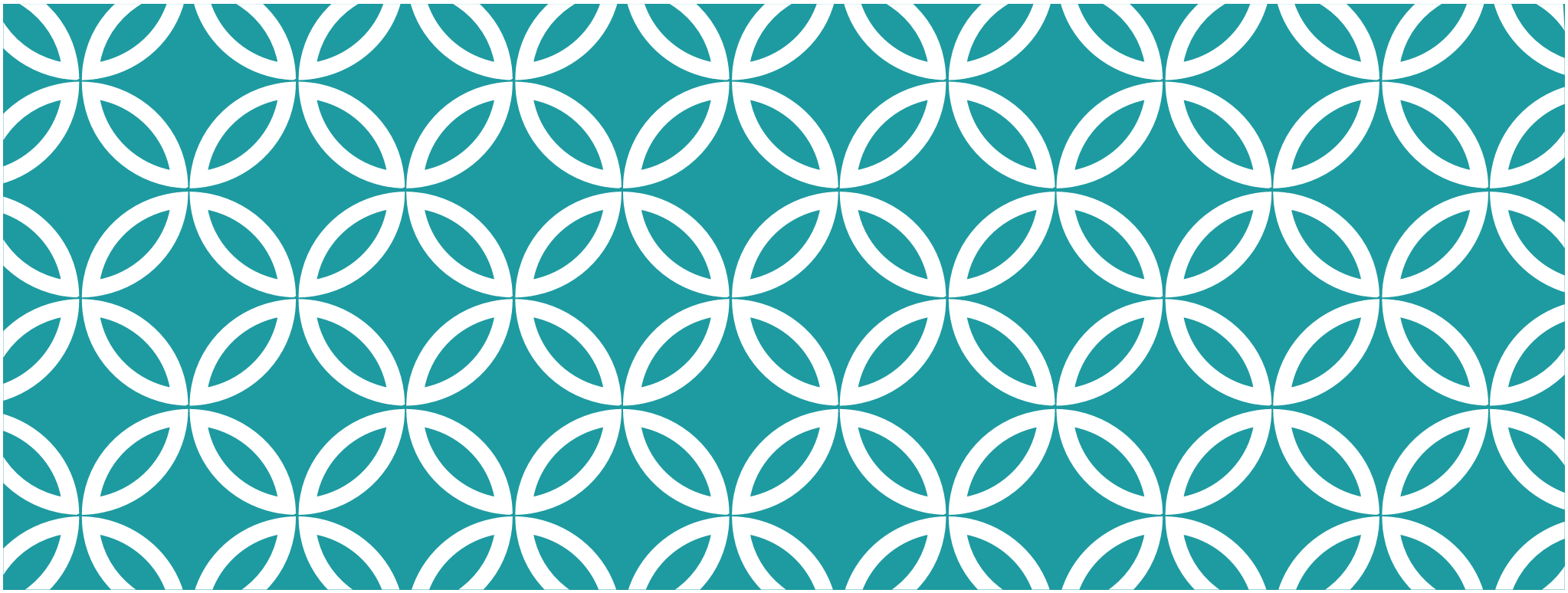
ADDING DEPENDENCIES

You add a specific module or dependency by adding it into the dependencies section of the `build.gradle.kts` file. Dependencies need to be specified using a "group name: module name: version number" (with a colon separating each one).

From the previous example, we can copy and paste the dependency line from the package information page directly into our `build.gradle.kts`

```
dependencies {
    implementation("io.coil-kt.coil3:coil-jvm:3.0.0-alpha06")
}
```

A project might have dozens of these.
It can get very messy to track.



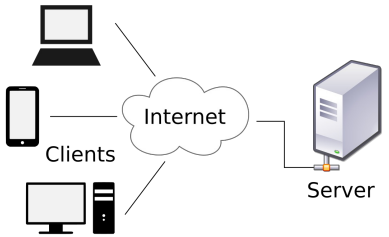
MULTI-PROJECT BUILDS

CS 346: Application Development

MULTI-PROJECT?

In a large application, you may design multiple components that are dependent on one another. For example, we could have client and backend that together provide functionality for our users.

- **Client**, using Kotlin and Compose. Your application could be a desktop or mobile application that uses the server.
- **Server**, using Kotlin and Ktor. Your server could be installed on a web server (as a JAR file) to provide services to applications.
- **Libraries** contain shared code used in both components.



You want these related modules to be stored in the same project, to avoid copy-pasting source code between projects (e.g. *monorepo*).

This avoids code duplication!

REMINDER: SINGLE PROJECT STRUCTURE

Let's see how Gradle handles this. Here's the standard single-project structure.

```
$ tree -L 5
.
├── build.gradle.kts
├── src
│   ├── main
│   │   ├── kotlin
│   │   └── resources
│   └── test
│       ├── kotlin
│       └── resources
├── gradle
│   └── wrapper
│       ├── gradle-wrapper.jar
│       └── gradle-wrapper.properties
├── gradlew
├── gradlew.bat
└── settings.gradle.kts
```

```
gradle/                 — gradle wrapper
gradlew                 - wrapper scripts
gradlew.bat


build.gradle.kts        - config for this module
settings.gradle.kts     - project settings


src/                    - source and unit tests
```

MULTI-PROJECT STRUCTURE

Each module/subproject is in a subfolder (e.g. android, application) with its own build.gradle.kts file

- e.g. `android, application, models, server`

The top-level, `settings.gradle.kts` file describes which modules to include.

Each module will have its own rules for building!

- An application builds and runs, and you can build an installer (and executable) from it.
- A library just exports to a jar file which can be used in other projects.
- A server exports a jar file which needs to be hosted on a web server, or some container.

```
$ tree -L 2
├── android
│   ├── build.gradle.kts
│   └── src
├── application
│   ├── build.gradle.kts
│   └── src
├── gradle
│   └── wrapper
├── gradle.properties
├── gradlew
├── gradlew.bat
├── local.properties
├── models
│   ├── build.gradle.kts
│   └── src
├── server
│   ├── build.gradle.kts
│   └── src
└── settings.gradle.kts
```

CREATING THIS STRUCTURE?

Just make the changes manually!

1. Create a regular Gradle project in IntelliJ IDEA.
2. Create your first subdirectory and move the `src/` folder and `build.gradle.kts` into the subdirectory.
3. Modify the `settings.gradle.kts` to point to the subdirectory.
4. Repeat steps 1-2 for every project type that you want to add.

For complex project structures (e.g., Android) I often "cheat"; I'll create a separate empty project, and then just copy the folder structure into my multi-project folder.

When creating simpler projects (e.g., Desktop, models) the above method works fine.

VERSION CATALOGS

One challenge to using a lot of dependencies is keeping track of the versions of libraries that you are using.

Gradle has a feature called version catalogs, which is a centralized file that contains a list of libraries and their versions.

- Gradle will automatically keep versions up-to-date using this file.
- In Gradle 7.x or later, the version catalog is contained in a file `libs.versions.toml` in your `gradle/` project directory.

You use the dependencies defined in the version catalog in your build config files.

gradle/libs.versions.toml

```
[versions]
guava = "32.1.3-jre"
junit-jupiter = "5.10.1"

[libraries]
guava = { module = "com.google.guava:guava", version.ref = "guava" }
junit-jupiter = { module = "org.junit.jupiter:junit-jupiter", version.ref = "junit-jupiter" }
```

build.gradle.kts

```
dependencies {
    // Use JUnit Jupiter for testing.
    testImplementation(libs.junit.jupiter)
    testRuntimeOnly("org.junit.platform:junit-platform-launcher")

    // This dependency is used by the application.
    implementation(libs.guava)
}
```