

# TypeScript in Practice: Best Practices

Project-level Best Practices



**Dan Wellman**

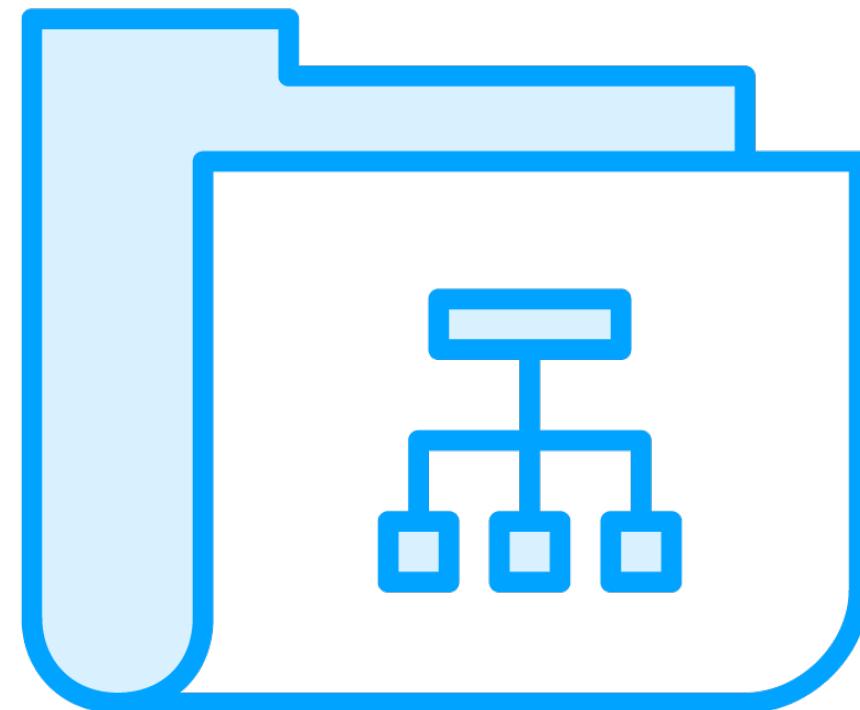
Senior Front-end Developer



# Structural Best Practices



# There Is No One True Folder Structure to Rule Them All!

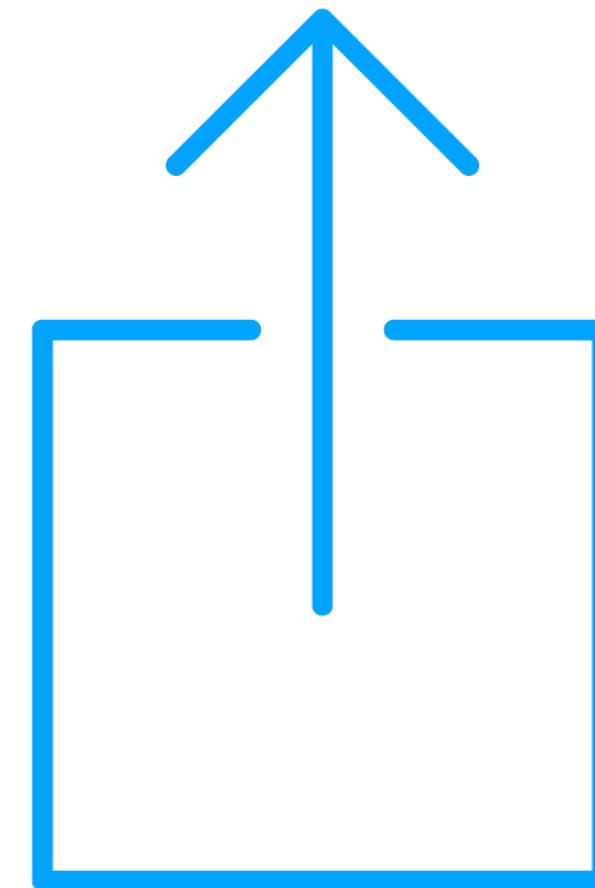


**File and folder structure will depend on:**

- The nature of the project
- Any frameworks in use
- Personal preference



# Remember the LIFT Principle



- Locatable
- Identifiable
- Flat
- Try to stay DRY



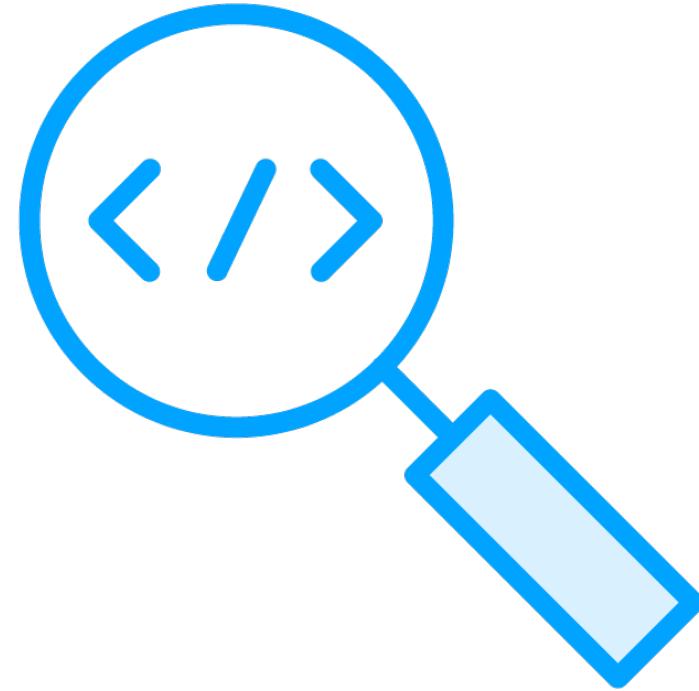
# Locatable



- Code Should be easy to find
- Name folders and files to promote findability



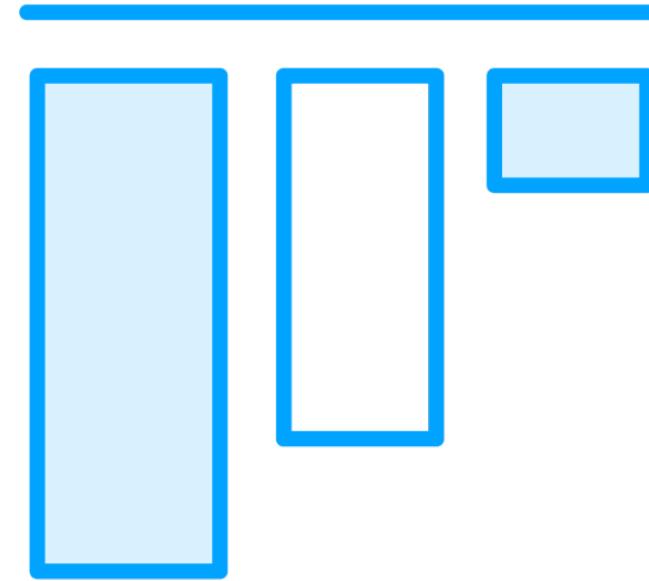
# Identifiable



- It should be easy to see which codes are in which files, and which files are in which folders



# Flat



- Keep the folder structure as flat as possible, with only as much nesting as required



# Try to Stay DRY



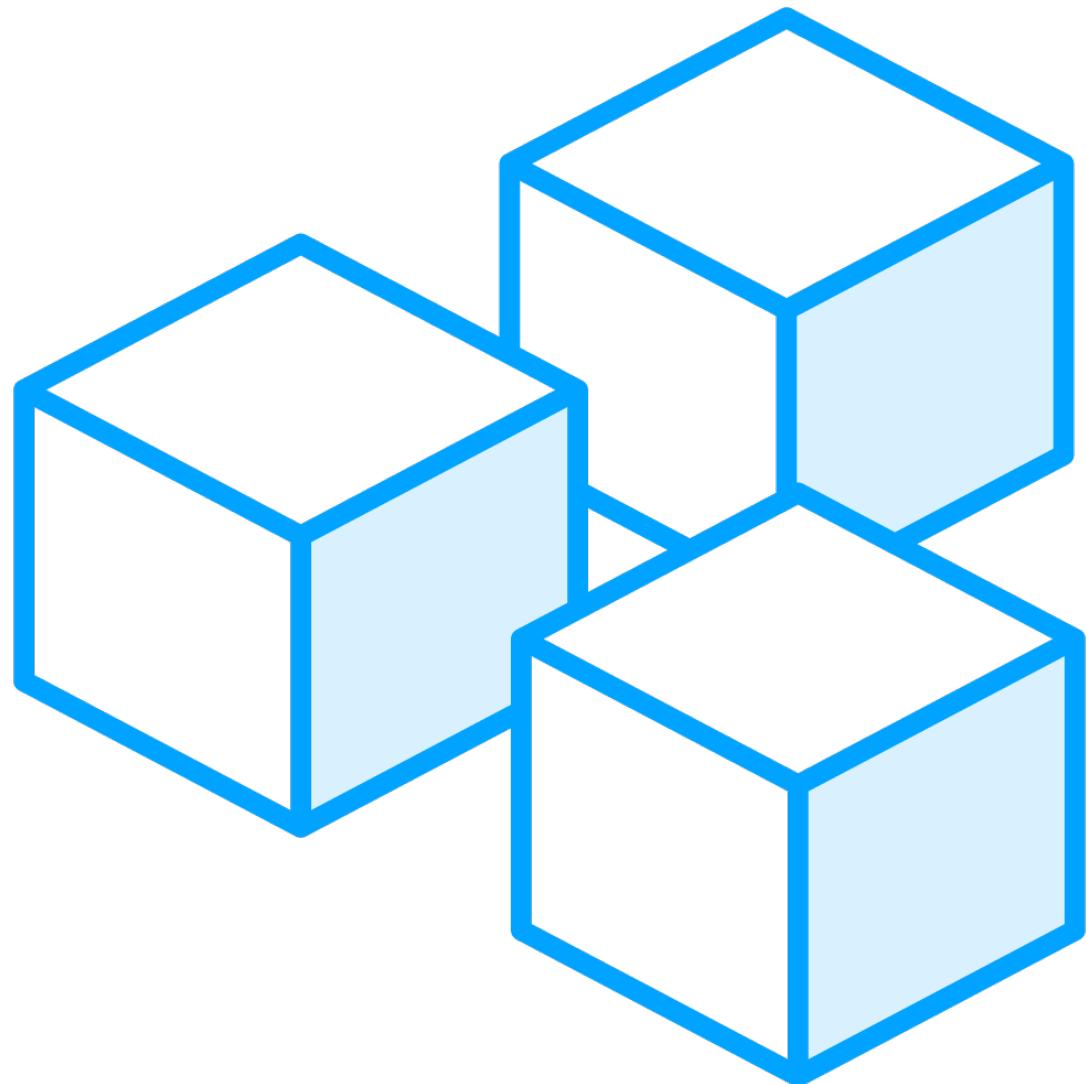
- Don't Repeat Yourself
- Avoid duplicating files of the same purpose and scope in multiple places
- Applies more to code but keep it in mind



# Architectural Best Practices



# Favor a Modular Design



- Code in discreet modules is easier to reuse throughout the application
- Modules should adhere to the single responsibility principle



# Functions should do one thing and do it well!

(Functions/classes/modules)



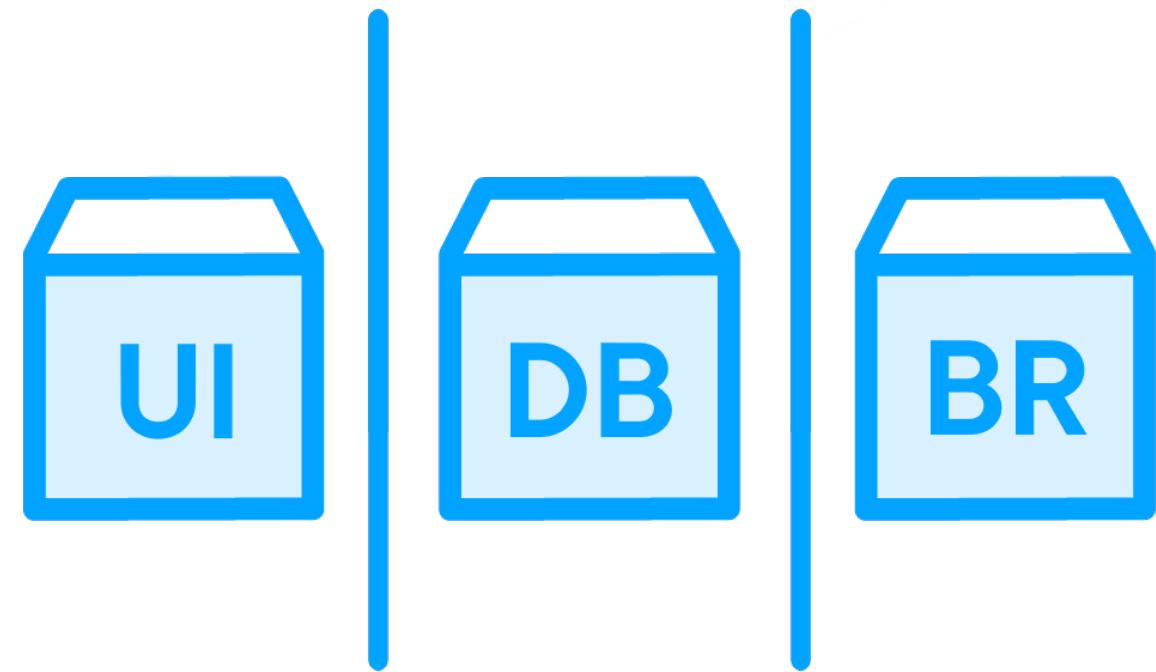
**The number one best  
practice for architecture is  
the separation of concerns.**





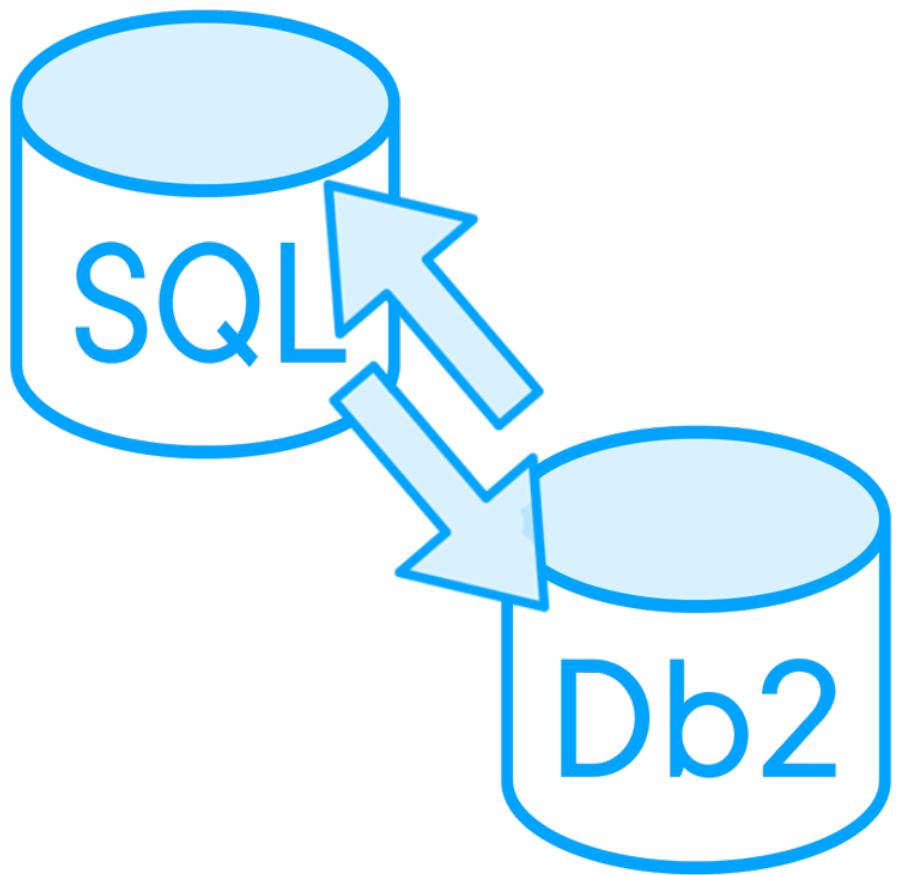
- 
- Different parts of our application





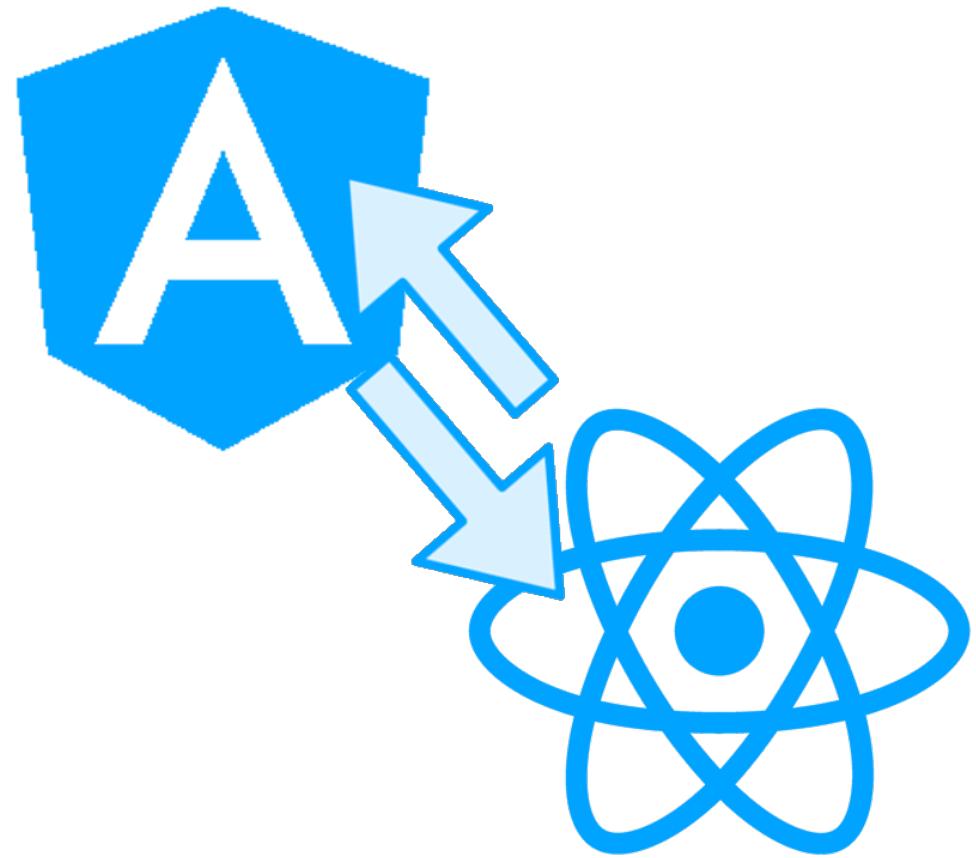
- Different parts of our application should be separated by clear and distinct boundaries





- You don't want to have to change your UI if you just want to change your database

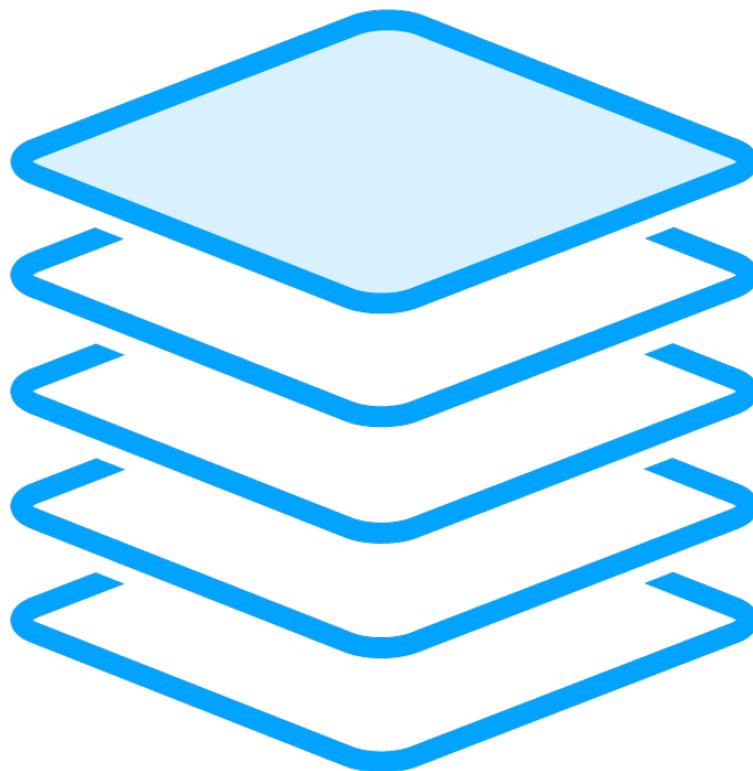




- You shouldn't have to rewrite your whole application just to change the UI framework



# Favor a Layered Design



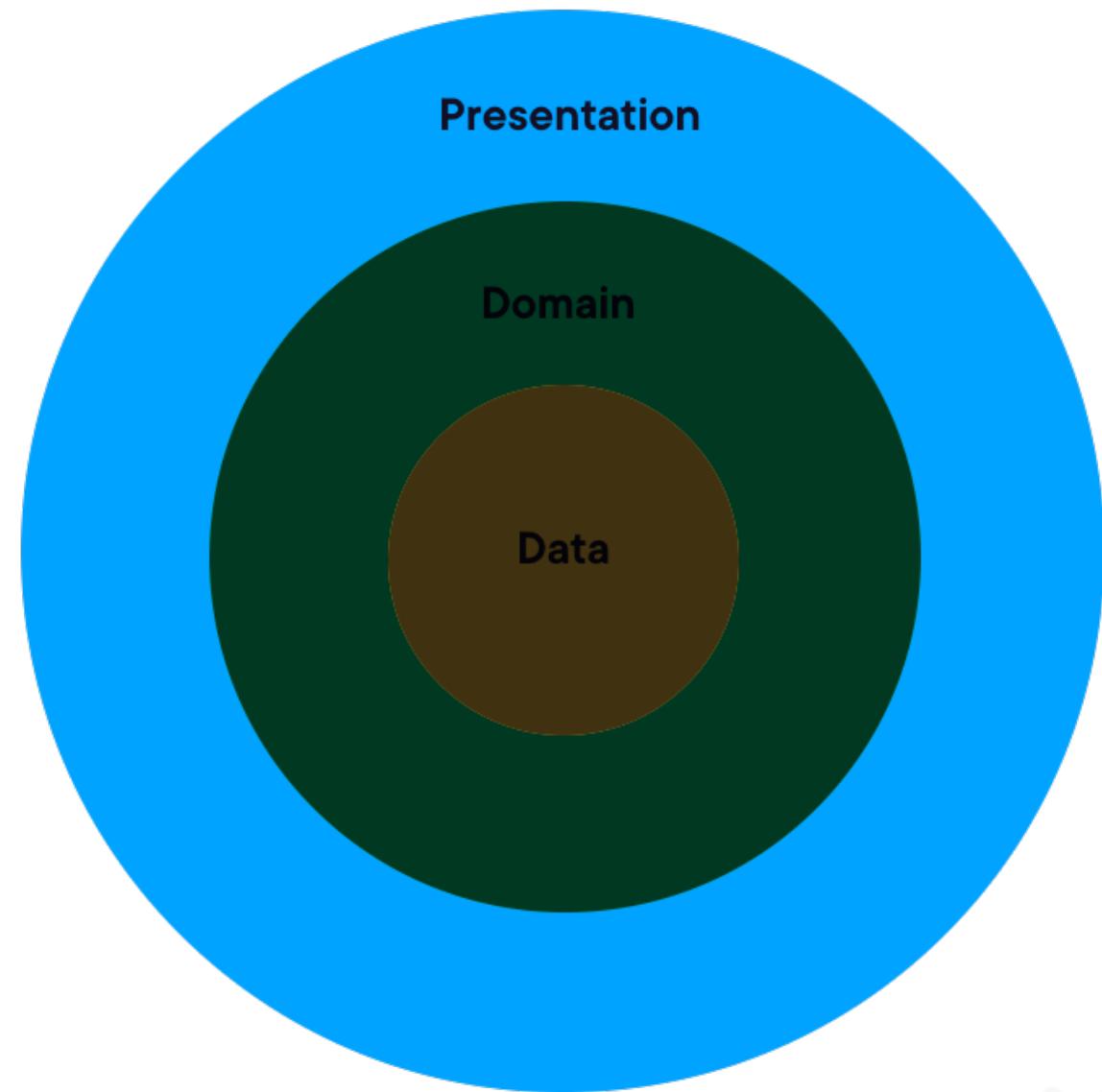
- Divide the application into distinct layers, each responsible for a different thing
- Minimize dependencies between layers



# Application Architecture



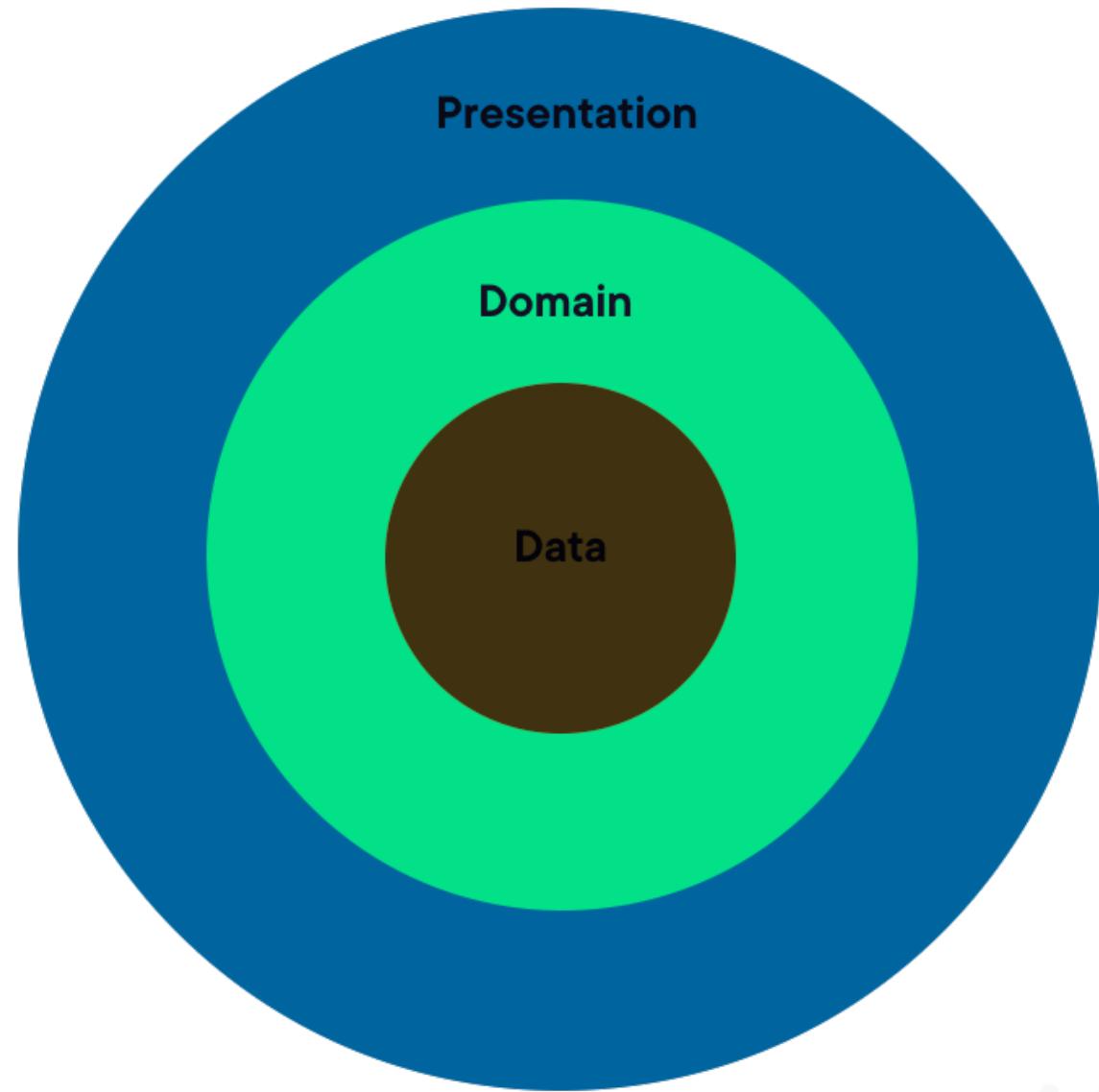
# Presentation Layer



- UI components, framework



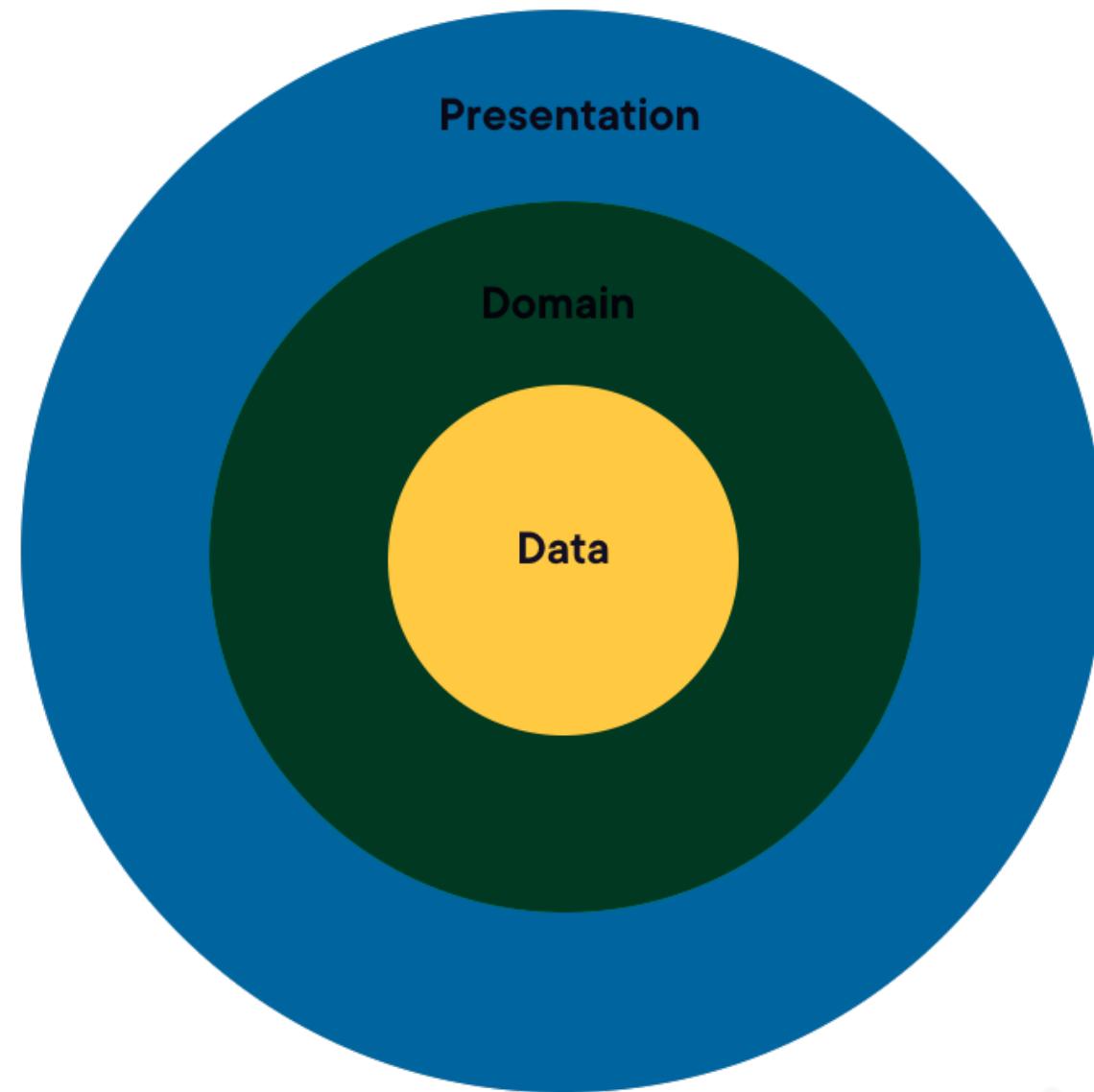
# Domain Layer



- UI components, framework
- Entities, business logic, use-cases



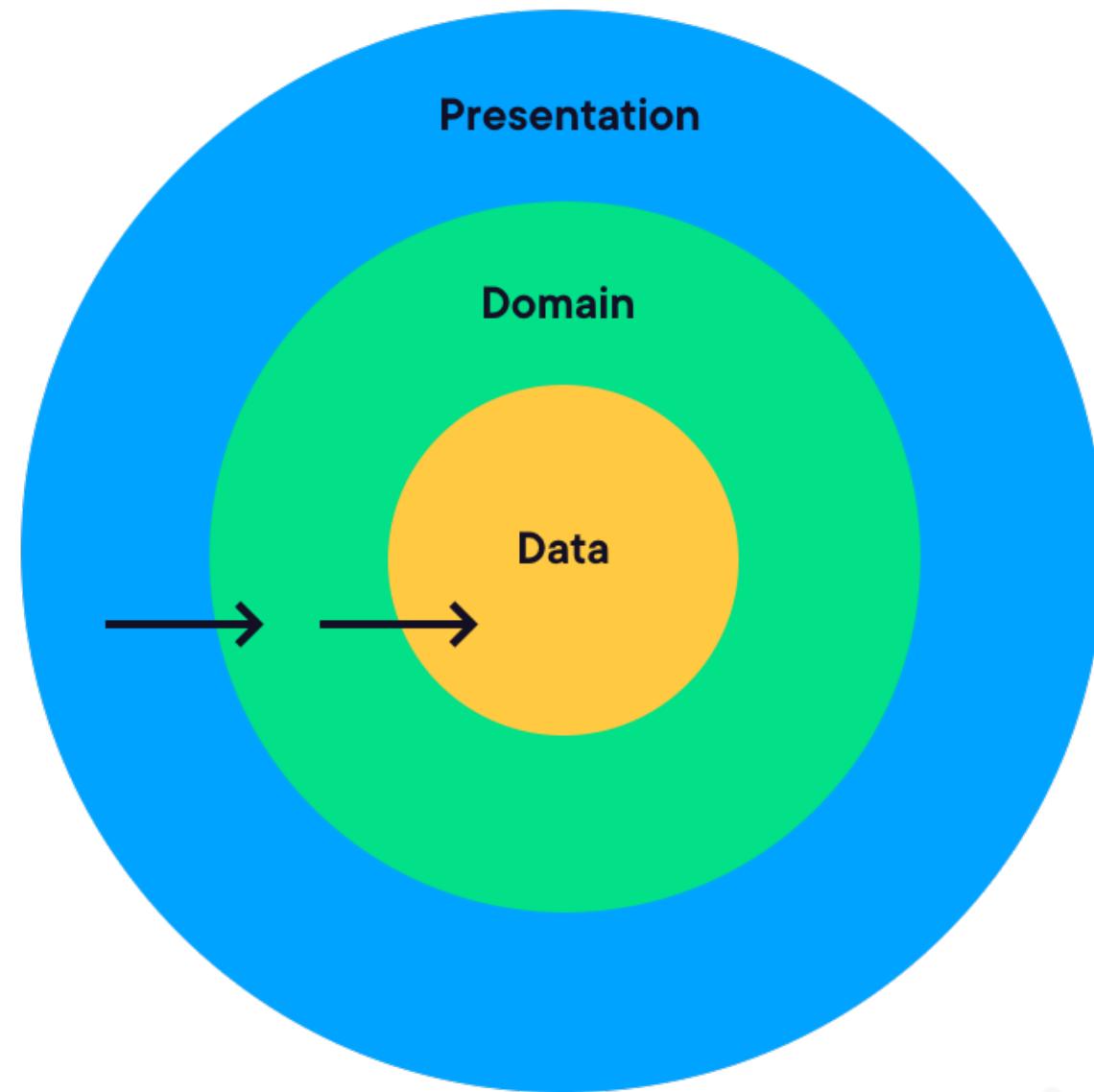
# Data Layer



- UI components, framework
- Entities, business logic, use-cases
- Storing/retrieving data



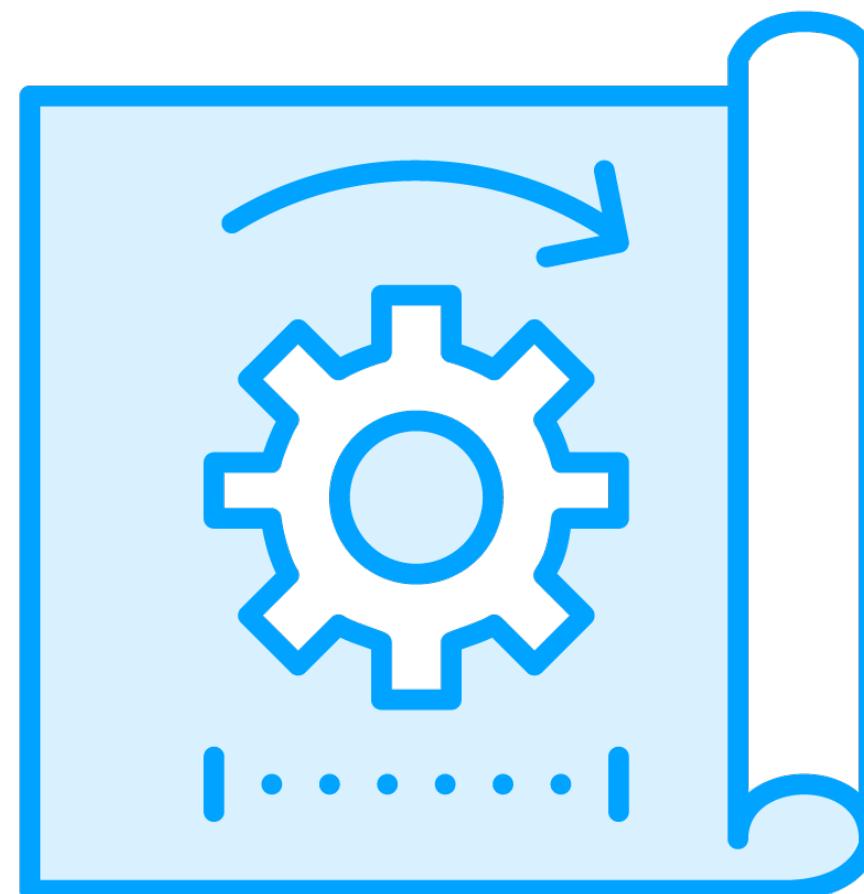
# Dependency Rule



- Dependencies should always point in
- Dependencies should be one level deep



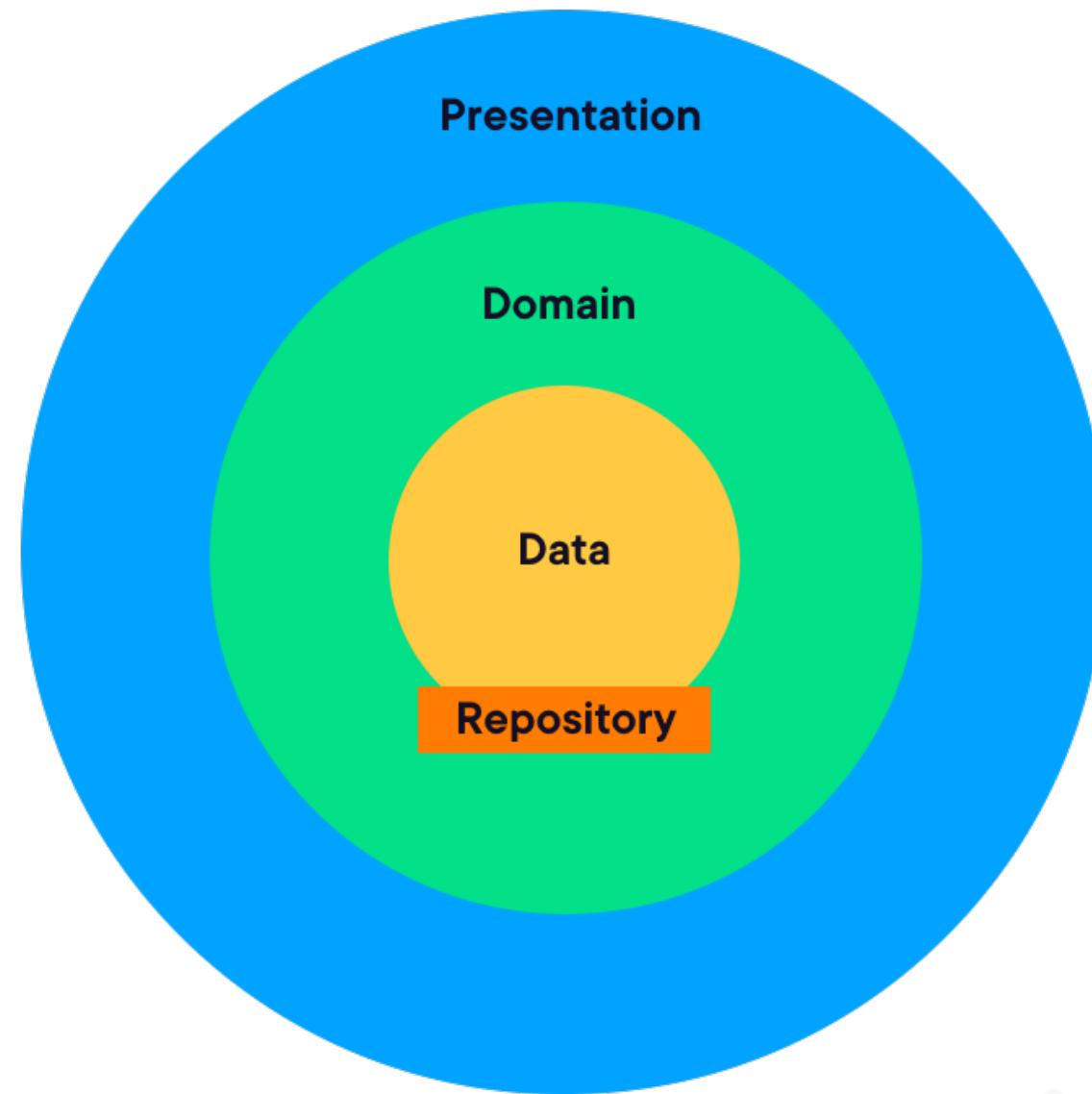
# Domain-driven Development



- First, design the core entities of the application and their behaviors and interactions
- Build the application around these



# Repository



- UI components, framework
- Entities, business logic, use-cases
- Storing/retrieving data
- A bridge between domain/data layers



**Repository – interact with a  
data source without  
knowing too much about it.**



**Best practices are just  
guidelines.**

**You can deviate from them  
in limited and specific ways.**





**More Information:**  
**Clean Architecture: Patterns,  
Practices, and Principles**  
**Matthew Renze**



# Configuration Best Practices



# TypeScript Configuration



- TypeScript apps are configured using one or more `tsconfig` files



# Always enable strict mode in new TypeScript applications!



# TypeScript Configuration



- TypeScript apps are configured using one or more `tsconfig` files
- New projects have strict mode enabled by default
- Purpose of strict mode is to allow older projects to be incrementally made strict



# Strict Mode Is an Umbrella For



- `alwaysStrict`
- `strictNullChecks`
- `strictBindCallApply`
- `strictBuiltInIteratorReturn`
- `strictFunctionTypes`
- `strictPropertyInitialization`
- `noImplicitAny`
- `noImplicitThis`
- `useUnknownInCatchVariables`



**Without strict mode, you lose  
much of TypeScript's power.**



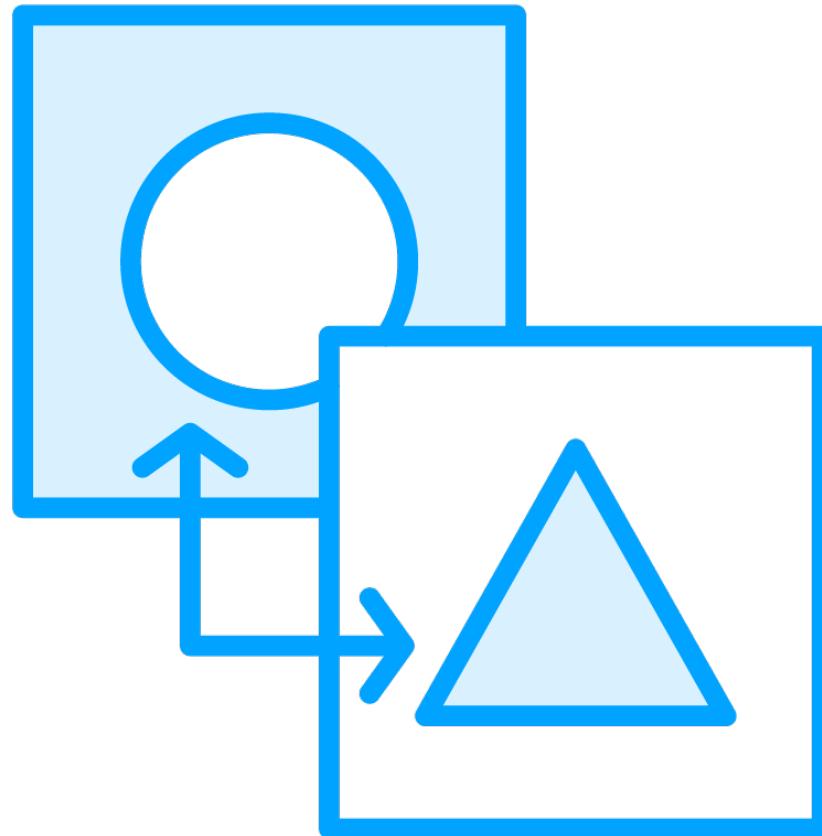
# Additional Type-checking Options



- noImplicitOverride
- noImplicitReturns
- noPropertyAccessFromIndexSignature
- noUncheckedIndexedAccess
- noFallThroughCasesInSwitch
- noUnusedLocals
- noUnusedParameters
- exactOptionalPropertyTypes



# TSC Compilation



- By default, TSC emits compiled files next to the original source file
- This can quickly clutter up the project with development/distributable files
- Separation makes tasks like source control and releases easier



# TypeScript Configuration



- TypeScript supports a wide range of configuration options to suit all projects
- Starting a project with maximum strictness/cleanliness options enabled gives a solid and safe foundation



# Tooling Best Practices



**Source control and a  
TypeScript-aware editor are  
a given in terms of tooling.**



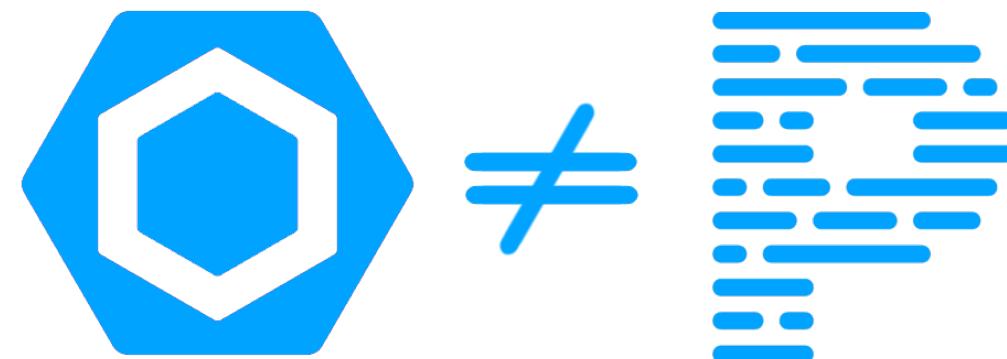
**Install a static code analysis  
tool to check for common  
errors.**



**Install a formatter for consistency in multi-developer projects.**



# Linters vs. Formatters



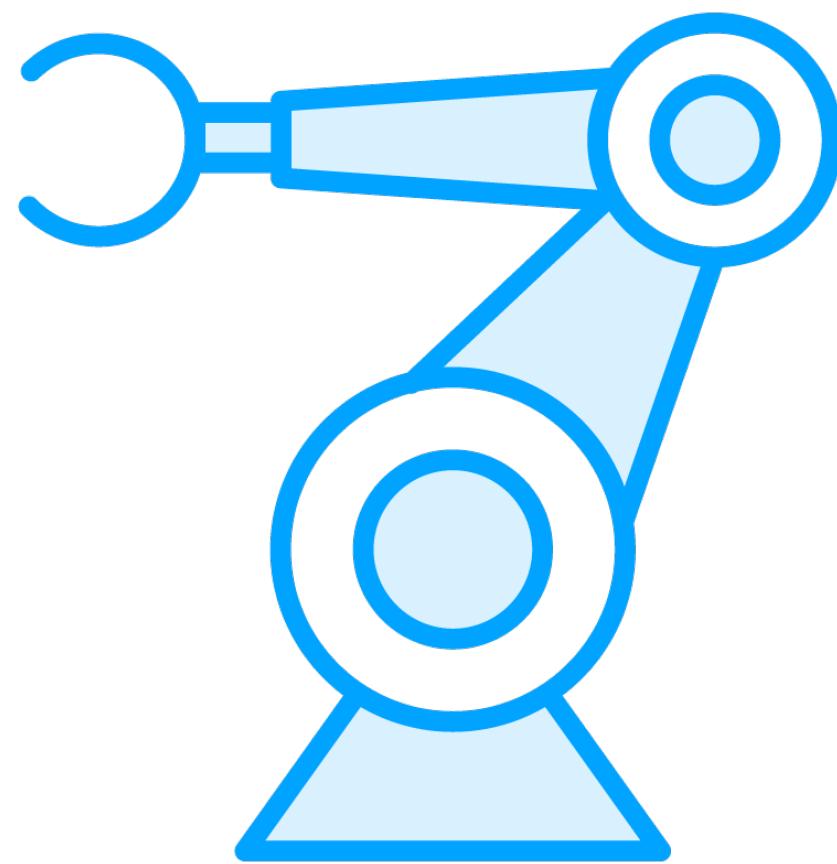
- Linters and formatters are not the same thing, although there can be overlap
- Formatters are concerned with the presentation of code in files
- We can configure whether semi-colons are added at line ends, tabs vs. spaces, etc.



# Automate linting and formatting tasks.



# Automation Options



## Editor integration

- Run Prettier on every file save
- Show lint warnings as you code

## Git hooks

- Pre-commit

