> For an overview of the general TypeScript compiler architecture and layering, see [[Architectural Overview]]

Overview

A simple analogy for a language service object is a long-lived program, or the compilation context.

Design Goals

There are two main goals that the language service design aims to achieve:

1. **On demand processing**

The language service is designed to achieve quick responses that scale with the size of the program. The only way this can be achieved is by only doing the absolute minimum work required. All language service interfaces only compute the necessary level of information needed to answer a query.

For instance, a call to `getSyntacticDiagnostics` will only need the file in question to be parsed, but neither binding nor type checking will be performed in the process. A call `getCompletionsAtPosition` will only attempt to resolve declarations contributing to the type in question, but not others.

2. **Decoupling compiler pipeline phases**

The language service design decouples different phases of the compiler pipeline that would normally happen in order in one shot during command-line compilation; and it allows the language service host flexibility in ordering these different phases.

For instance, the language service reports diagnostics on a file per file basis, all while making a distinction between syntactic and semantic errors of each file. This ensures that the host can supply an optimal experience by retrieving syntax errors for a given file without having to pay the cost of querying other files, or performing a full semantic check. It also allows the host to skip querying for syntax errors for files that have not changed. Similarly, the language service allows for emitting a single file (`getEmitOutput`) without having to emit or even type check the whole program.

Language Service Host

The host is described by the LanguageServiceHost API, and it abstracts all interactions between the language service and the external world. The language service host defers managing, monitoring and maintaining input files to the host.

The language service will only ask the host for information as part of host calls. No asynchronous events or background processing are expected. The host is expected to manage threading if needed.

The host is expected to supply the full set of files comprising the context. Refer to reference resolution in the language service for more details.

ScriptSnapshot

A `ScriptSnapshot` represents the state of the text of an input file to the language service at a given point of time. The ScriptSnapshot is mainly used to allow for an efficient incremental parsing. A `ScriptSnapshot` is meant to answer two questions:

1. What is the current text?
2. Given a previous snapshot, what are the change ranges?

Incremental parsing asks the second question to ensure it only re-parses changed regions.

> For users who do not want to opt into incremental parsing, use `ts.ScriptSnapshot.fromString()`.

Reference resolution in the language service

There are two means of declaring dependencies in TypeScript: import statements, and triple-slash reference comments (`/// <reference path="..." />`). Reference resolution for a program is the process of walking the dependency graph between files, and generating a sorted list of files comprising the program.

In the command-line compiler (`tsc`) this happens as part of building the program. A `createProgram` call starts with a set of root files, parses them in order, and walks their dependency declaration (both imports and triple-slash references) resolving references to actual files on disk and then pulling them into the compilation process.

This work flow is decoupled in the language service into two phases, allowing the host to interject at any point and change the resolution logic if needed. It also allows the host to fully manage program structure and optimize file state change.

> To resolve references originating from a file, use `ts.preProcessFile`. This method will resolve both imports and triple-slash references from a given file. Also worth noting is that this relies solely on the scanner, and does not require a full parse, so as to allow for fast context resolution which is suited to editor interactions.

Document Registry

A language service object corresponds to a single project. So if the host is handling multiple projects it will need to maintain multiple instances of the LanguageService objects; each instance of the language service holds state about the files in the context. Most of the state that a language service object holds is

syntactic (text + AST). The projects can share files (at minimum, the library file `lib.d.ts`).

The document registry is simply a store of SourceFile objects. If multiple language service instances share the same `DocumentRegistry` instance they will be able to share `SourceFile` objects, allowing for more efficient memory utilization.

A more advanced use of the document registry is to serialize `SourceFile` objects to disk and re-hydrate them when needed.

The Language service comes with a default `DocumentRegistry` implementation allowing for sharing SourceFiles between different `LanguageService` instances. Use `createDocumentRegistry` to create one, and pass it to all your `createLanguageService` calls.