Hecate

About modular design and data driven tooling

Introduction

Hecate or Hekate[a] is a goddess in ancient Greek religion and mythology, most often shown holding a pair of torches, a key, snakes or accompanied by dogs and in later periods depicted in triple form. She is variously associated with crossroads, entrance-ways, night, light, magic, witchcraft, the Moon, knowledge of herbs and poisonous plants, graves, ghosts, necromancy, and sorcery.

en.wikipedia.org/wiki/Hecate

Any sufficiently advanced technology is indistinguishable from magic.

Arthur C. Clarke

Build systems are often a messy set of scripts and configuration files that handle build, test, package, deliver, and install the code. Developers either love or loathe build systems. Software development is a complex activity that involves tasks such as code management, code generation, automated source code checks, documentation generation, compilation, linking, packaging, creating binary releases, source-code releases and deployment. Every change a developer, graphic artist, technical writer, or any other person that creates source content makes, must trigger at least a partial rebuild of the entire game

Convention over Configuration

Is a principle that has successfully governed in domains like web frameworks, reducing the learning curve and increasing developer productivity. It demands that a project is organized in a consistent way. Regular directory structure is the key principle on which such tools rest. Even in the most complicated systems, there is usually a relatively small high-level directory structure that contains special purpose directories. Usually the build system should be aware of the location and names of the top-level directories and their meaning to the project.

Each directory usually contains a small number of file types that are automatically discovered, based on extension. The build system usually knows what files to expect, how to handle each file type and performs many tasks automatically. In particular, it doesn't even need a build file in each directory that tells it what files are in it, how to build them or even a monolithic “solution” that points to every single directory. The regular directory structure, combined with knowledge of files types (e.g., .cpp or .h files), allows the build system to figure out what files it needs to take into account, so developers just need to make sure the right files are in the right directory.

Managing dependencies can be simple or complicated depending on the project. In any case, missing a dependency leads to linking errors that are often hard to resolve. This build system analyzes the language specific statements in the source files and recursively creates a complete dependencies tree. The dependencies tree determines what static libraries a dynamic library or executable needs to link against. However, good tools are designed in a global fashion so that they can resolve dependencies to modules that are installed into a generic directory as well as those defined per project.

Different IDEs, as well as command-line based tools like Make, use different build files to represent the meta information needed to build the software. The conventional build system maintains the same information via its inherent knowledge combined with the regular directory structure and can generate build files for any other build system by populating the appropriate templates. This approach lets developers build the software via their favorite IDE (like Visual Studio) without the hassle involved in adding files, setting dependencies, and specifying compiler and linker flags. It is important to understand that the build process executes independently of any project files for the development environment, such as .sln or .vcproj files for Visual Studio. These files are useful to have for editing purposes though, so there is a tool being provided to generate them dynamically, based on the contents of a project directory tree

Build Actions

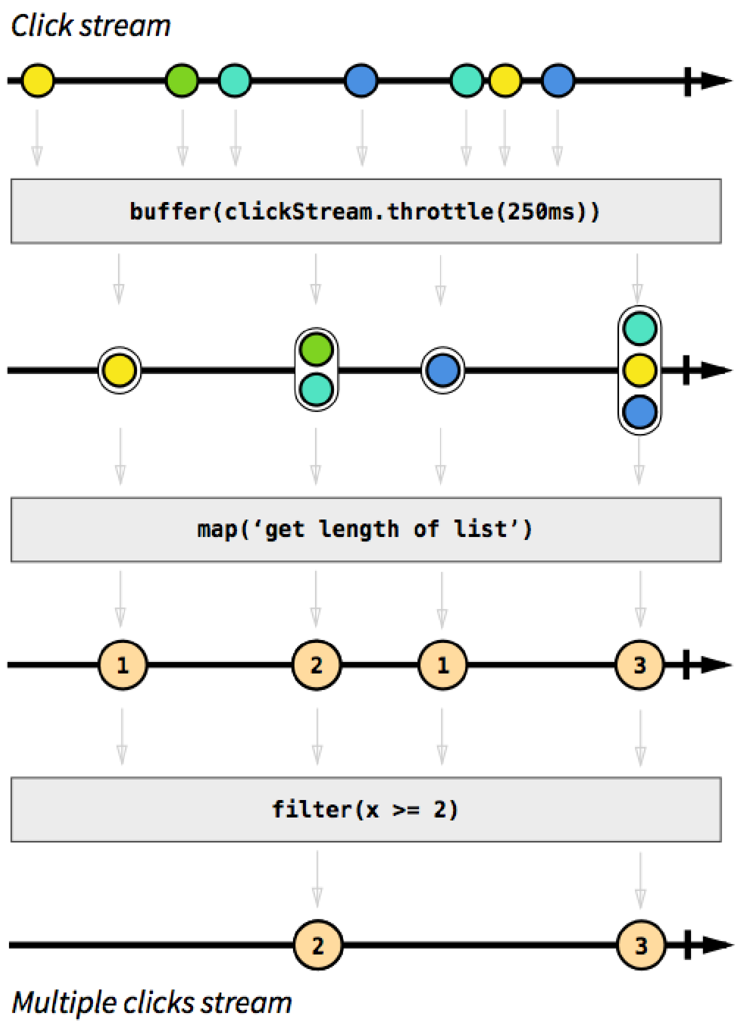
Regardless of how a build system is set up for a project, all of them have something in common. They perform a set of actions or nodes that may relate to the output of a previously finished node up to the final compilation, building and deployment steps. Each node consists of tasks executed in sequence to produce some sort of output

Reactive Programming

Is a program design paradigm that relies on asynchronous data streams of time-ordered sequences of related event messages. Those streams are coherent, cohesive collections of digital signals created on a continual or near-continual basis. Streams are cheap and ubiquitous, anything can be a stream; variables, user inputs, properties, caches, data structures and many more. A given stream will generally start with an observer and so relates to the observer pattern.

The observer pattern lets any object that implements the subscriber interface subscribe for event notifications in an observable object, letting observers hook up their custom code via custom subscriber classes. In reactive programming, observables are coupled to observers through operations. When an observable sends data, the data is operated on by the operations and then received by the observer.

Operations are the functions that do transformations on the data that is sent from an observable to an observer or on the observable itself. A stream can be used as an input to another one or even multiple streams can be input for one or more other streams

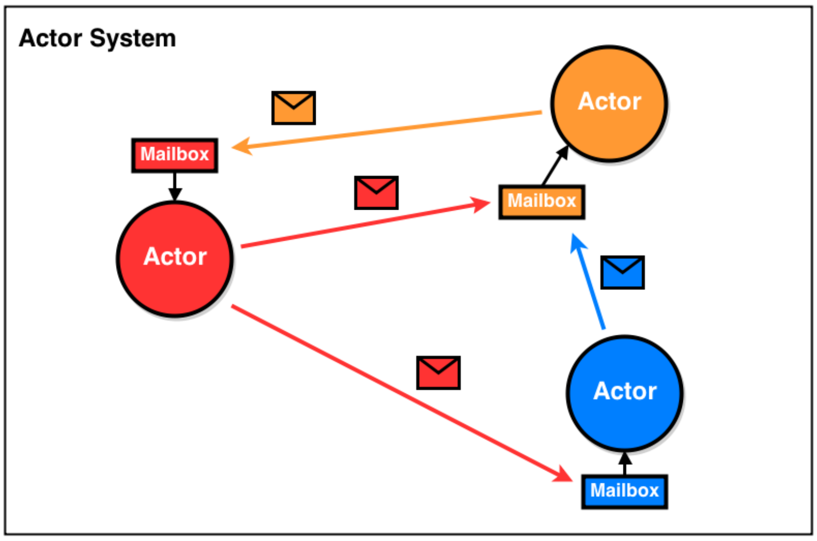

Click stream
buffer(clickStream.throttle(250ms))
map('get length of list')
1
2
1
3
filter(x >= 2)
2
3
Multiple clicks stream

Actor Model

Is a conceptual model to deal with concurrent computation. It defines some general rules for how the system's components should behave and interact with each other. Actors are the primitive unit of computation, receiving messages and doing some kind of computation based on those messages. They are completely isolated from each other and will never share any memory.

Actors can also be used to create self-healing or fault tolerant systems. The first actor created becomes the parent of other actors it creates. This actor is the supervisor of any other actor. If an actor gets into a faulting state, its supervisor can handle that state in order to heal the system and take it back into a running state.

Messages are sent asynchronously to an actor. An actor processing a message can itself send messages to other actors and also await response

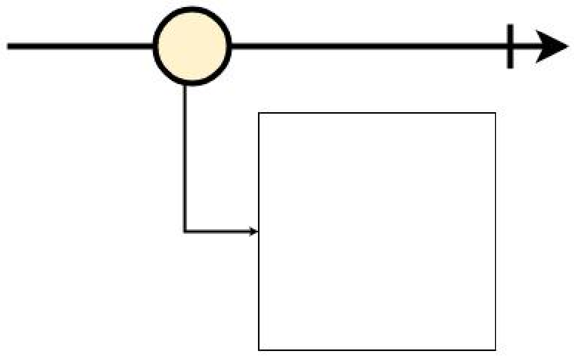


Processor Units

Are specialized stacks of actors that subscribe to a multi-wired message stream. The built system envelopes nodes to an actor stream that eliminates many manual steps from the process and enables a smooth, automated flow of data from one node to the next. The data stream is related to the reactive programming paradigm, which allows actors to subscribe for certain kind of data, enables annotations for the type of machine that nodes are supposed to be executed on, providing a list of recipients for failure notifications if a step fails, and groups nodes that should only be executed after an explicit action finished. It automates the processes involved in extracting, transforming, combining, validating, and loading data for further execution.

Those nodes are called Processor Units. They are different from a regular actor in such a way that it is assigned to a single kind of processor family that manages exactly one type of data. They can be defined and used by the built system from different sources like user code or a plugin which determines their logical location in the hierarchy, regardless of where the code is originated in.

Processor Units of the same family can co-exist in one pipeline, ordered into a hierarchy that determines which instance is currently addressed to the data. Built-in instances are usually defined by a tool or module in the and form the lowest level of a cluster. These nodes become the default behavior for certain type of data. Nodes that are originated in the global installation directory are prioritized above built-in ones but after those originated in a project directory. Finally, nodes related to a certain path anywhere in the file system have the highest priority and are chosen if data from the pipeline is covered by the path that node is originated in



The build system’s inherent knowledge combined with the regular directory structure can fulfill a lot of purposes based on the built-in default behavior but can also make use of clustered actors to define local overrides of the default behavior on a per-path scope and so a fine grained conventional customization of actions performed

Project Entities

Describes a component based design approach which promotes code reusability by separating data from behavior. The built system does not has systems that usually operate on entities and only provides methods for querying components.

In line with the motto *there are no projects, everything can be a project*, the build system doesn’t know anything about the concept of a source code project. It relies on the user to provide one or more paths to the root directories of what is considered to be handled as a code module instead. Those paths are treated as the starting point of any action it performs and become fortified with more information while being processed. To make this possible, paths are treated as entities and every piece of additional information added to them is handled in the sense of adding additional data components. It is arguable that the build system’s data driven environment is pillared on the back of an entity-component system

```
public class CodeModule : FlexObject
{
    /// <summary>
    /// The directory path of this module
    /// </summary>
    public PathDescriptor Location
    {
        get;
    }

    /// <summary>
    /// The name of this module
    /// </summary>
    public string Name
    {
        get;
    }
}
```

Components are usually created and passed to the underlying reactive data pipeline in order to have them processed by the corresponding processor units.

Linting

Usually is the automated checking of source code for programmatic and stylistic errors. This is done by using a lint tool, otherwise known as linter. A lint tool is a basic static code analyzer.

Due to the data driven facility of the build system, it also relies on a linter for every supported language to statically analyze code files in order to determine dependencies between code modules and projects.

It can however deliver great benefits in development, especially when the compliance of strict rules is important to be fulfilled. Therefore the build system offers the opportunity to declare additional rules. They will be applied together with the built-in rules to every code file

C# Code

Projects in .NET enable managing source code files as cohesive groups and also support the building of each cohesive group into one deployable artifact (.dll, .exe, etc.). Each project is represented by one project file.

Project files are essentially XML files that act as containers of files. Every Visual Studio project includes an MSBuild project file, with a file extension that reflects the type of project. A project file usually includes dependencies such as Other projects from the same solution, system assemblies, custom assemblies and NuGet packages.

The convention over configuration approach eliminates the use of such detailed project files. This means that dependencies have to be resolved by static code analysis only and so it is necessary to make use of peculiarities of the programming language itself. A fast and reliable way to get to a proper solution is to have the build system to maintain the C# related project component with namespace definitions found in the corresponding files and match using directives to them. This will also match precompiled system assemblies and custom assemblies once they have been loaded via. NET Reflection

C++ Compilation Units

In the C/C++ compilation model, individual .c/.cpp source files are preprocessed into translation units, which are then compiled separately by the compiler into object files. Those files can then be linked together to create a single executable file or library. #include is a way of including a standard or user-defined file in such a program. This directive is read by the preprocessor and orders it to insert the content of a user-defined or system header file into the current translation unit. These files are mainly imported from an outside source into the current program.

Like a preprocessor is handling dependencies, the build system has to resolve them by static code analysis too and so it is necessary to make use of the same peculiarities of the programming language. Whenever an include directive occurs, the build system performs lookups into the code modules currently loaded and tries to match the desired file location with those files covered by the code module location in the file system. The C++ related project component then maintains the matches found

Modules

Modular programming is a general programming concept that involves separating functionality into independent pieces or building blocks. Each module contains all the parts needed to execute a single aspect of the functionality. Building blocks need however to be distinguished between atomic features and those that are assembled from other building blocks.

Modules usually make code easier to read and understand because it is clearly separated into functions that each are categorized into dedicated aspects of the overall application. It also makes code files a lot smaller and easier to understand compared to monolithic code projects. Modularity however involves grouping similar types of functions into their own files and libraries.

Projects split into distinct modules are also easier to test and debug. Because of the separation of features into standalone functionality, tests can be much stricter and more detailed and also the process of debugging a single module is a lot more specific. Bugs can be stemmed more easily to a single module or the communication between building blocks that make use of other building blocks.

Modular programming is essential where multiple teams need to work on different components of a program. Code that is split between more functions, files and packages, can be easier split between multiple developers and even teams assigned ownership to specific modules of code, ensuring they are responsible for their part of the software, and enabling them to break the work down into smaller tasks

Packaging

A code package is usually a module that can be added to any project to add additional features or functionality. They are an essential tool for any modern development platform providing a mechanism through which developers can create, share, and consume useful code. A package-management system is a collection of software tools that automates the process of installing, upgrading, configuring and removing modules and also handles dependencies between code modules.

The built system contains a customization of NPM, a package manager for source packages. A package contains all the files needed for a module which are language independent code libraries included in a project. An NPM package is a file or directory that is described by a package.json file. Open-source developers from every continent use NPM to share and borrow packages, and many organizations use NPM to manage private development as well.

Each module has a base directory that controls how it is built, which files belong to the module, a package.json defining module dependencies and a well set unique name as same as the version used by NPM lookup. The built system makes heavy use of code module packages, defined by a contained package.json file. When it starts to process a code module selected for build action, it automatically detects any known packages the module depends on and also those that peer-depend on this particular code module and adds them into the processing as well.

Plugins

Are software additions that allow for the customization of the built system, adding a feature to the software without changing the software at all. NPM also allows for peer dependencies, a pretty simple to use solution writing a plugin. When it starts to process a code module selected for an action, it also detects packages that peer-depend on this particular code module.

Defining the version of the host package, a plugin peer-depend on, and adding it to the plugins package.json is anything needed to allow the build system to detect peer relations and properly handle adding plugins into processing. This way a developer can add additional behavior to any software project without modifying existing code modules and their package description

Integration

The built system is written in C# .NET and delivered as a set of necessary open source code modules. It is a console application and needs to be compiled into an executable. NET assembly once before further packages can be installed. A set of batch and shell scripts covering different platforms, like Windows and Linux, and .NET versions handle the steps to locate the appropriate C# compiler and perform the initial compilation of the necessary code files and modules into a single binary executable. The build system is independent from usual .NET development environments. Additional software tools like Visual Studio or MSBuild aren't needed.

After the initial setup is completed, the build system takes over and can even rebuild itself, taking further code modules and plugins into the updated assembly, to modify the behavior and extend provided features like supported programming languages

Editor

Apart from being a standalone tool, the build system can also be integrated into various IDEs engines and their editors, in order to be easily accessible for everyone to become part of the development process. There are multiple ways of accomplishing a well working integration into an editor, the most valuable is however to create a custom code module to acts as bridge between the command line driven build system and the usually UI driven editor application. Turning this code module into a peer-depending package enables the modification to be built as a true alteration to the build system