Starlane: Reduce the Drudgery of Infrastructure Code with WebAssembly

A Unique Vision for WebAssembly

WebAssembly (https://webassembly.org) (Wasm) will disrupt the tech industry in the next few years. Many amazing projects are innovating at a furious pace to make Wasm ubiquitous. I am a huge fan of these efforts! But when I first encountered WebAssembly I was filled with excitement because it was exactly what I needed to create a particular type of framework that I had been mulling and yearning for… a framework that I believe will: ***reduce the drudgery of creating infrastructure code and shift developer focus to code that adds value to their users***.

I hope you will read on and let me know what you think of this proposal: good or ill!

The Problem: Value vs. Infrastructure

I have been a full stack developer for 25 years. As my career progressed I began to lament that the workload for every project was split so that 40% of my efforts was in crafting the "***unique value offering***" components of the project and remaining 60% of my work was creating the necessary *"plumbing"* or infrastructure to connect those components which entailed configuring, deploying, upgrading, transforming, sanitizing and filtering data, routing, network calls, message queues, persistent storage and and infinite list of other odds and ends.

To illustrate this dichotomy I want to expand on the meaning of these two distinct categorizations:

1. ***UNIQUE VALUE OFFERING CODE***
   Every software solution must provide a ***unique value offering*** to its users — if it lacks a unique value offering it shouldn't exist. The unique value offering is the reason people choose software in the first place. For instance: every smartphone comes bundled with a standard calculator app that has basic operations. Would you download another calculator app that has the exact same features? Probably not. Might you try another calculator app if it did something your present calculator doesn't do? Maybe you find another calculator app integrates with your online banking software eliminating a lot of copy and paste effort. The new calculator app has a unique value offering that removes the pain of looking up and copying figures from a spreadsheet People will install the new app because with it their phone will become more useful than before! That is the whole reason software exist on computers so it can constantly do new stuff that wasn't imagined at the time the hardware and operating system shipped!

2. ***INFRASTRUCTURE CODE***
   Infrastructure can mean many things. I categorize infrastructure code as anything that isn't unique value offering code. Infrastructure is of course vital. For example: a web application is not of much use if it doesn't make rest calls to the backend… of course no user in history uses a website because it claims as a feature: *"Now it can make rest calls to our backend!"* Infrastructure is vital to the success of any application yet it behooves the developer to seek frameworks and solutions that automagically provide as much framework as possible so he or she can focus as much as possible on the unique value offering code that his users are paying him for.

NOT A NEW IDEA

A Framework that eases the burden of developer efforts towards infrastructure is not a a revolutionary idea that came to me. Throughout my career I have embraced many brilliant innovations from other developers.

I was a Java developer for 25 years and I think the most impactful framework that eased my infrastructure woes has been the Spring Framework. The Spring Framework provides infrastructure via ***"Inversion of Control (https://en.wikipedia.org/wiki/Inversion_of_control#:~:text=In%20software%20engineering%2C%20inversion%***Which means basically that the Spring Framework can be configured via convention, configuration and annotation to inject infrastructure into the unique value offering components.

Starlane's offerings build on many great frameworks rather than aim to replace them. With that explained I will now sally into the unique innovations that Starlane proposes.

The Starlane Framework

Starlane is written in Rust although developers can use any language that compiles to WebAssembly to extend it.

Starlane composes an application from a developer configuration which may be a single process or span across multiple heterogeneous devices (client, server, browser, IoT, Edge… anywhere WebAssembly can be run.)

Starlane extends itself via the unique value offering code which is contained in WebAssembly guest components packaged and distributed by an artifact repository.

Starlane facilitates WebAssembly components with a variety of host implementations. The simplest components merely need to accept input and produce output and error streams which means the most basic components require no system or infrastructure code whatsoever.

Developers can configure components to run in more powerful guest environments which require the incorporation of some rudimentary guest apis in order to facilitate the proper execution.

Back Up… Why WebAssembly?

> (Please skip to ***"That was 'Interesting' but Back to Starlane!"*** if you aren't interested in my historical accounting and comparison and contrasting of Containers, The Java Virtual Machine and WebAssembly)

WebAssembly is a standard for a deterministic virtual machine which is completely isolated from the host environment (it cannot make any dangerous system calls directly.) Guest method invocation on a host must be explicitly whitelisted making it reasonably secure by design.

WebAssembly was introduced as a high performance alternative to JavaScript on the browser but in recent years its compact and secure nature has made it an obvious choice for backend solutions and in particular will probably replace or augment containers. In fact Docker has experimental support for WebAssembly execution. Many popular languages are presently undergoing accelerated efforts to update their compilers to support WebAssembly targets.

There are many recent offerings to proliferate WebAssembly and make it ubiquitous everywhere (embedded, edge, backend, desktop apps, mobile apps, the browser…). And many of those projects are awesome!

WebAssembly vs. Containers

I am excited about WebAssembly because it can be delivered far more compactly and with greater security than a container…. Let's explore the many ways WebAssembly distinguishes itself from containers:

Docker images are typically heavy multi-megabyte artifacts that executes a single and complex process usually bundled with a minimal linux operating system. Docker has a fantastic security model for executing trusted code. Organizations either run docker containers they themselves created or container images supplied by trusted third parties like Nginx or Wordpress. Containers have been and will continue to be a great solution for server applications.

WebAssembly is meant to execute in a completely isolated virtual machine host. This architectural choice means that a WebAssembly guest cannot by default even query the computers clock time. No system calls are permitted at all. The host is able to whitelist an enumeration of host method calls which a guest WebAssembly can call. And then the Host can in turn make any system calls that are required and of course a given exposed host method could be poorly conceived resulting in a security concern, however, its important to appreciate that WebAssembly's architecture makes guest execution inherently more secure by design.

And I brought the receipts for artifact size! I have never been able to create a Docker image smaller than a handful of megabytes *(this doesn't mean it can't be done — there are developers of greater docker prowess than I — but hyper small images are not common). *I have created a WebAssembly component image that prints "Hello World" and is only 500 bytes(not megabytes, not kilobytes just plain old bytes!)

I believe that size to power ratio of WebAssembly images are an essential prerequisite for expanding the micro service pattern even to an even more fine grained solution and also expanding the micro service pattern to the client side. I intend to create a future article explaining my rational for these beliefs… please save any hate mail until AFTER I publish the article…

WebAssembly vs. Java

And why is WebAssembly's approach exciting over Java? Well, *it's story time*: in my previous life I was a Java developer and Java was introduced to the world as a major innovation over c and c++ because among other things it executed inside a Virtual Machine (https://en.wikipedia.org/wiki/Virtual_machine). The advantages to any virtual machine are portability and security… right now we are only discussing security…

When a native C or C++ app executes it is not required to have an internal security model. Of course the operating system does provide some security guardrails but consider the case of a personal computer running Windows or MacOs: the security philosophy of the PC is that YOU are the sole owner of the computer and its content and YOU are intensionally running a program to accomplish something and the operating system cannot discriminate if an app is deleting some files because thats what you want or the app was written by a stinker of a naughty developer that filled his code vessel with malice and the will to destroy.

The Java Virtual Machine by default *permits* every system call imaginable that could lead to trouble, but it also shipped with a security model. The java security model worked by blacklisting system calls… meaning the platform would have to intensionally configure which system calls to reject.

The blacklist approach was somewhat successful in the form of a lost art of Java Applets which basically ran Java code on the browser with a highly restrictive security model. Java Applets had a lot of problems and often frustrated users and eventually fell out of fashion — but when first introduced it was a revolutionary to be able safely execute machine code on a client.

Unfortunately since the blacklisting approach requires intentionality on the part of the platform engineer the task is not frequently bothered with at all. This very oversight culminated in one of the biggest security scandals of the last decade when a little known java logging utility called log4j for unknown reasons exposed

a system call that allowed the java program to gain administrator access to its host operating system. And that's the scary thing about any language that incorporates useful libraries… even security scans cannot be relied upon to weed out every permutation of vulnerably in the windy depths of third party libraries.

That was "Interesting" but Back to Starlane!

Starlane Resources

What is a resource? In Starlane resources can be a file system, database, an external proxy a file an S3 Bucket… it is really an abstraction for *anything* in the application *anywhere* in the application.

The abstract resource pattern was inspired by the brilliant unix architectural philosophy that "everything is a file (https://en.wikipedia.org/wiki/Everything_is_a_file#:~:text=%22Everything%20is%20a%20file%22%20is,throu innovation. The file abstraction in unix simplified infrastructure tooling and alleviated developer headaches because the same tools that developers created to handle a data file could also be used to work with devices, sockets, processes and more.

Starlane facilitates Resource interoperability via message passing.

The WebAssembly components created by the application developer are actually just another kind of Starlane Resource.

Resource Component Decoration

Resources can also be decorated with Pre and Post WebAssembly components to filter, transform or validate messages… for Instance it may be desirable on a particular Image repository FileSystem to verify that the data payload is actually a recognized image format a WebAssembly decorator component can accept the payload of a create message and validate that it is actually a recognized image format and return an error on stderr if it is not.

The Starlane Registry

Starlane provides discovery services to its WebAssembly components via its Resource registry.

Starlane can provisions resources via configuration or via special custom WebAssembly initiator components.

Components are able to lookup, create and delete resources via the Starlane Registry. Importantly the components do not need to know anything about the infrastructure topology to interact with the Resources. It is Starlane's job to manage these resources in the way they are configured.

Starlane Message Actions

Messages have a small enumerated list of familiar actions including: *Create, Read, Update and Delete.*

The Resource abstraction eliminates many infrastructure hassles faced by the app developer. For example if a developer wants his WebAssembly component to create a file he simply issues *Create* message to a FileSystem resource . The message payload is taken as the content data of the File. A few code saving phenomenons are happening in this example:

1. The developer WebAssembly component includes a guest library for messaging the host provided and maintained by the Starlane framework. This library is simple yet because of the Resource abstraction it is quite powerful. In the given example the component sends a create message to a

filesystem resource, but could have just as easily sent it to a MessageQueue resource. Like unix the Starlane framework scales developer productivity faster because it strives to unify solutions to common patterns.

2. The developer's WebAssembly component does not need to know anything about the application topology in order to route the create message. It is Starlane's job route the message to a resource that is local to the process or transform the message into a rest call and send it a thousand miles away on a remote server. The guest component developer supplies the end address and Starlane handles the drudgery of making sure the message get to the desired destination.

3. Starlane resource drivers handle abstracted requests to generate a consistent result. I mentioned creating a File on a FileSystem resource… but what is the FileSystem resource? It could be a local directory on the client machine where the WebAssembly component is running OR it could be an S3 Bucket somewhere on the backwaters of the cloud. Starlane provides some common Resource Drivers and also allows developers to create 3rd party Resource Drivers to expand Starlane's capabilities.

Starlane Security

Starlane has a security auditor which checks if the sending resource is allowed to message the receiving resource with a particular action.

Security is a very broad topic that I intend to go into greater detail in a future article about Starlane security. Of course security is critical to the success of the Starlane Framework and be assured a lot of work continues to go into it.

More…

That is the initial introduction to Starlane. I intend to revise this article and add a summary based on the initial feedback I get. I also will be writing several articles about different aspects of Starlane not addressed here including one called *"The 3 Goals of the Starlane Framework"* which has been in development for quite a while…

The State of Starlane

Starlane is NOT ready for primetime and is probably not much of a ready state to even experiment with since I active halted active development.

The good news is I am sufficiently recharged and feel like this project has enough merit that it deserves some of my time

Where to get Starlane

As I said it isn't ready for primetime but the GitHub repository is publicly available so if you are up for the pain please knock yourself out!
GitHub - cosmic-initiative/cosmic-initiative: Starlane makes it easy to deploy and interoperate… (https://github.com/cosmic-initiative/cosmic-initiative)