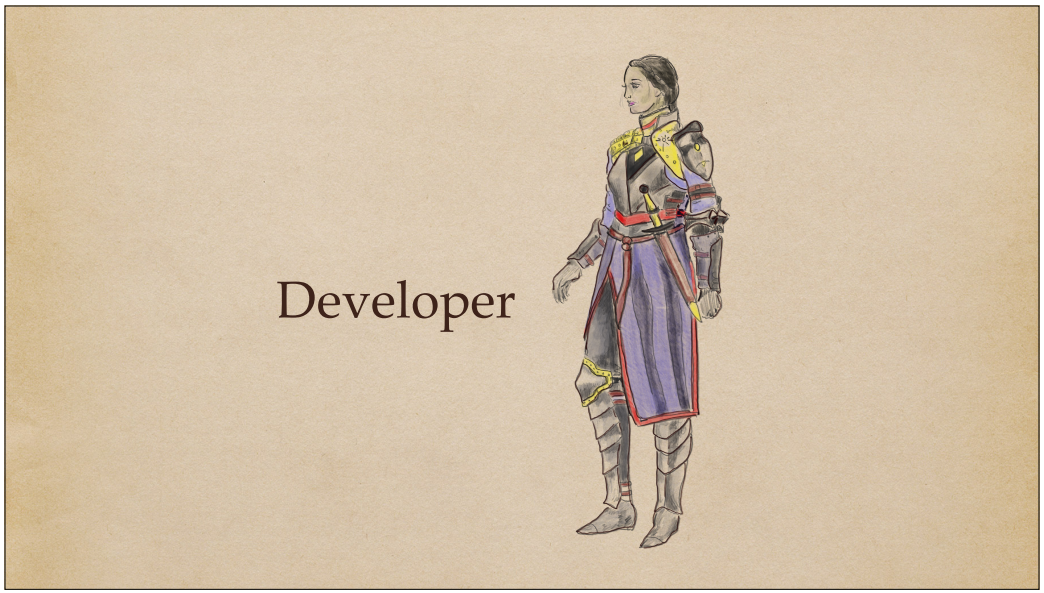
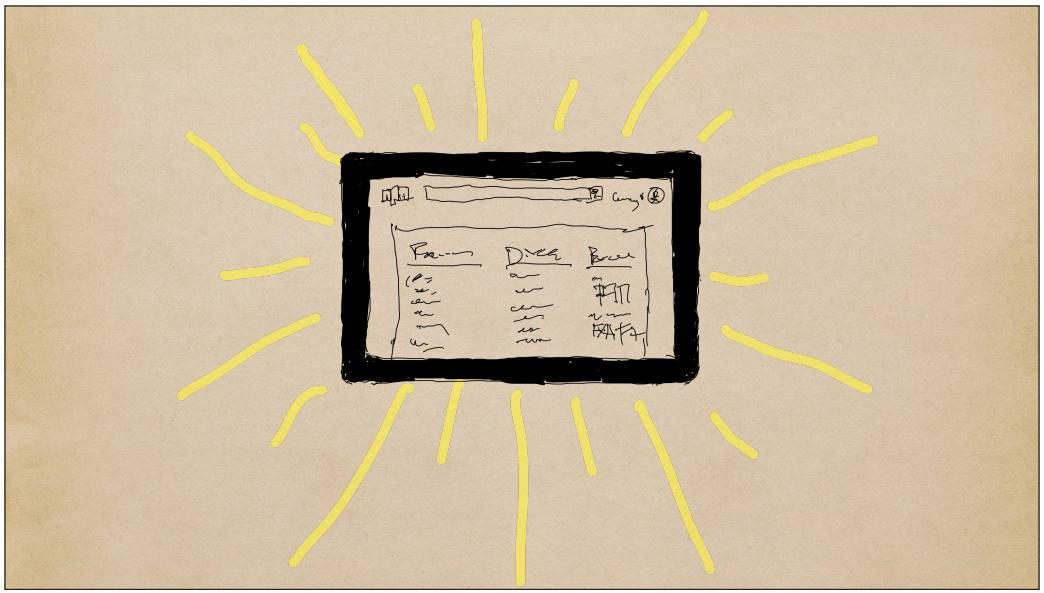


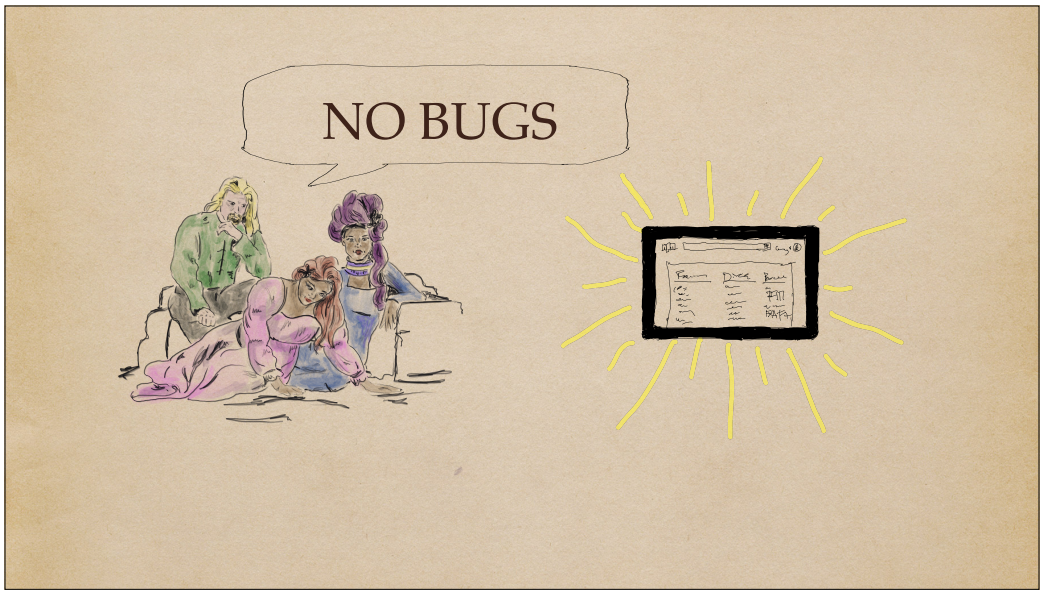
The world is veiled in darkness. Dependencies are inconsistent, builds are slow, version numbers matter not. The people wait, their only hope, a prophecy… When dependency hell has plagued your JavaScript project, a tool will come… After a long journey, some programmers arrive, each holding a pretty loaded metaphor for a technical talk.



Our story begins not so long ago, when you - our hero developer - are given a mission



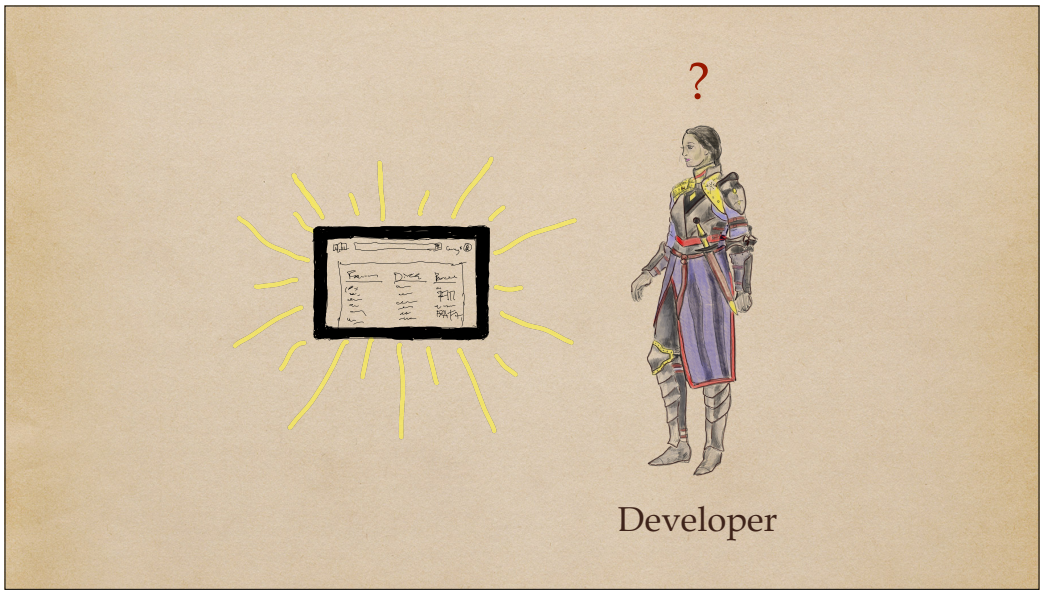
build a great website to honor your Gods of user.



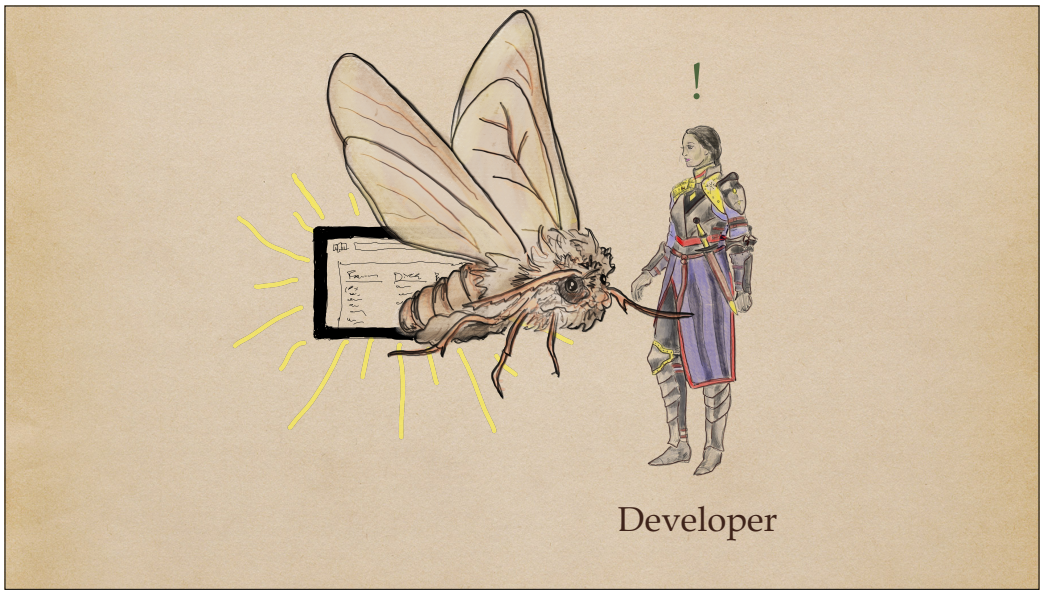
Your Gods are numerous and vengeful and will not tolerate an experience less than pristine.



Your bosses, The Masters of C-Team, are demanding that you appease the Gods quickly, [click] so as they can can be granted riches in this life and the next.



This task sounds grueling. Building each piece of a website from nothing involves strong magic, patience, and time.



In a world in which each of these items is scarce - we need something to help us, something to guide us, something to give us the pieces that leave us less exposed to the *click* cruel monstrous insects that anger our User Gods.



There is a place that is spoken of that was used long ago.

After the great programming war, when the dark ages came into place, [click]

all of our help and hopes were destroyed, and speaking of this location and the tool to get there was forbidden. But word of mouth has passed on from generation to generation and you, our hero, will search out and find the land of our desires, the Registry.

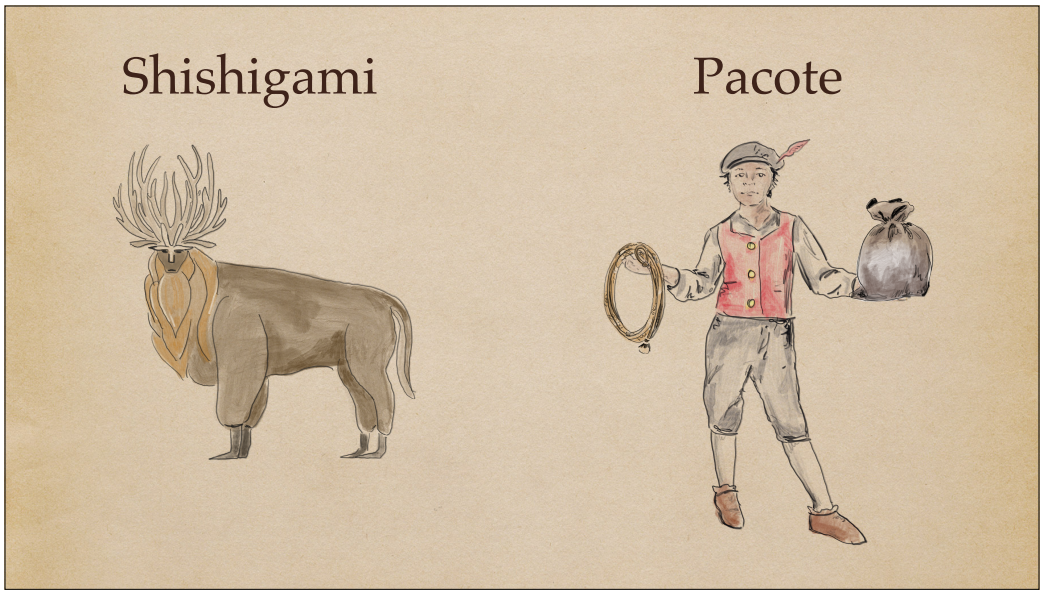


The only thing we know is that to enter the registry, there is talk of a tool — a magical weapon to destroy dependency hell that was forged by sorceresses of colorful hair. This tool survived multiple iterations and was faster and more powerful than anybody could have imagined. This tool, is called “n” “p” “m”. npm was cleanly broken apart by the sorceresses, in case the day ever came where npm needed to be built again. [click twice, book one chapter two follows]

Book One
Chapter One



According to old documentation found in the runes of Github, our magical tool requires the help of three companions:
Shishigami, the Forest Spirit



Pacote, the master of packages

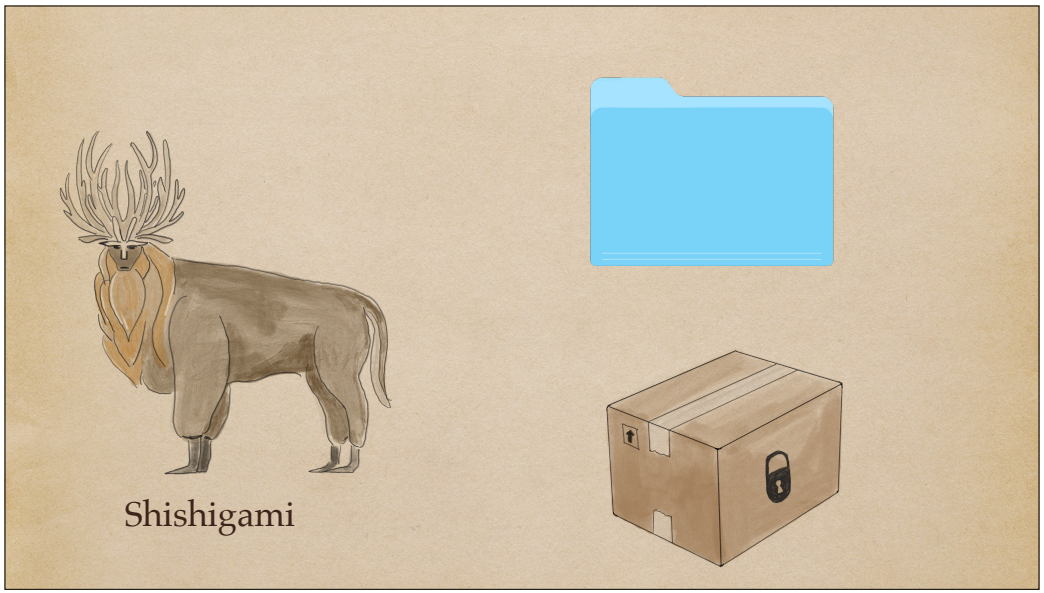


and Registry Client, a smuggler that can give you a special passcode to enter the Registry. Each of these, when brought to the sorceresses of colorful hair, can be forged together by the light of JavaScript to create the npm tool.

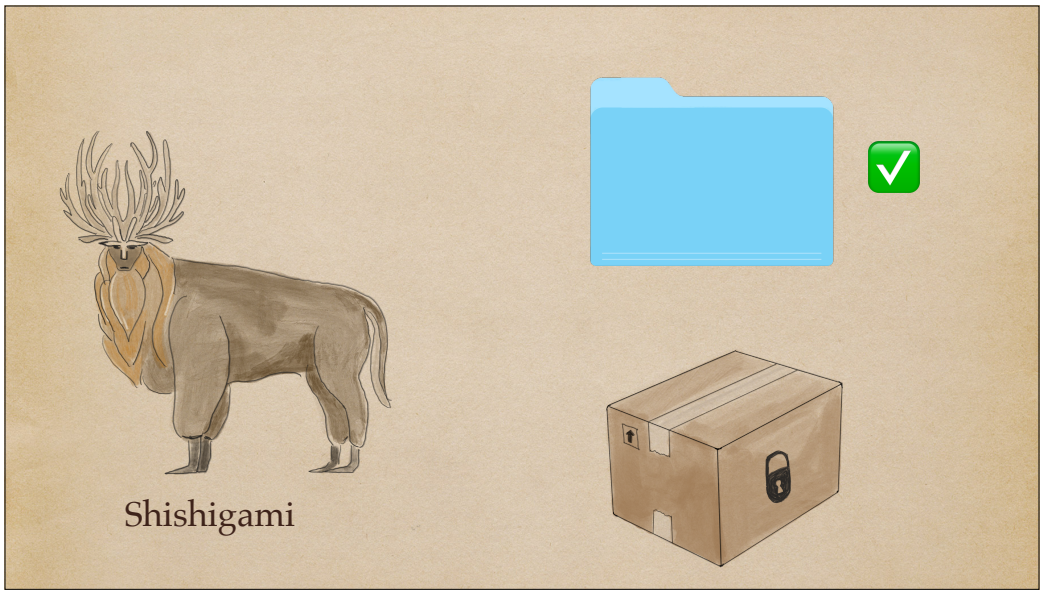


And so, hero, to begin, we visit Shishigami, the creator of the dependency tree.

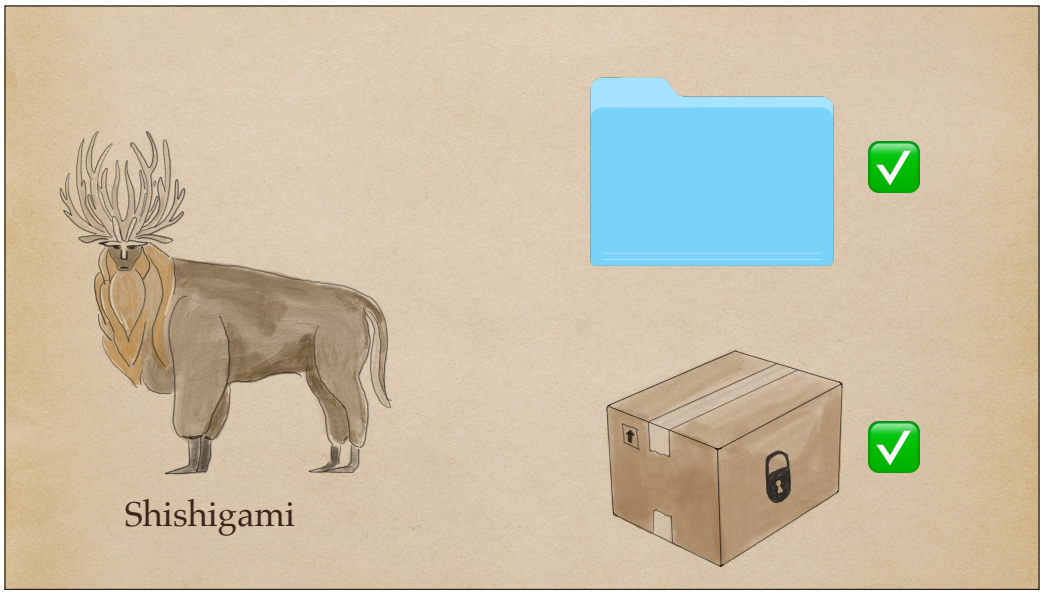
The forest spirit's job is to piece together exactly what you need from your install.



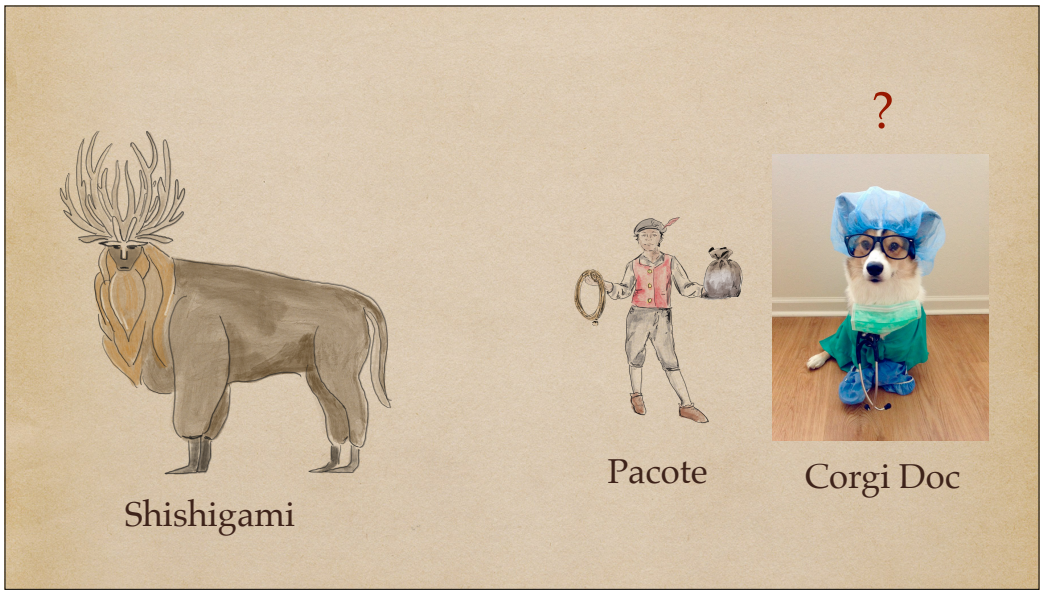
It scours through your list of package demands and also through what you already have installed and puts together an ideal tree - a perfect structure of your dependencies, so all your needs will be served.



It achieves this by first reading your local package tree. It does this by combing through your `node_modules` folder.

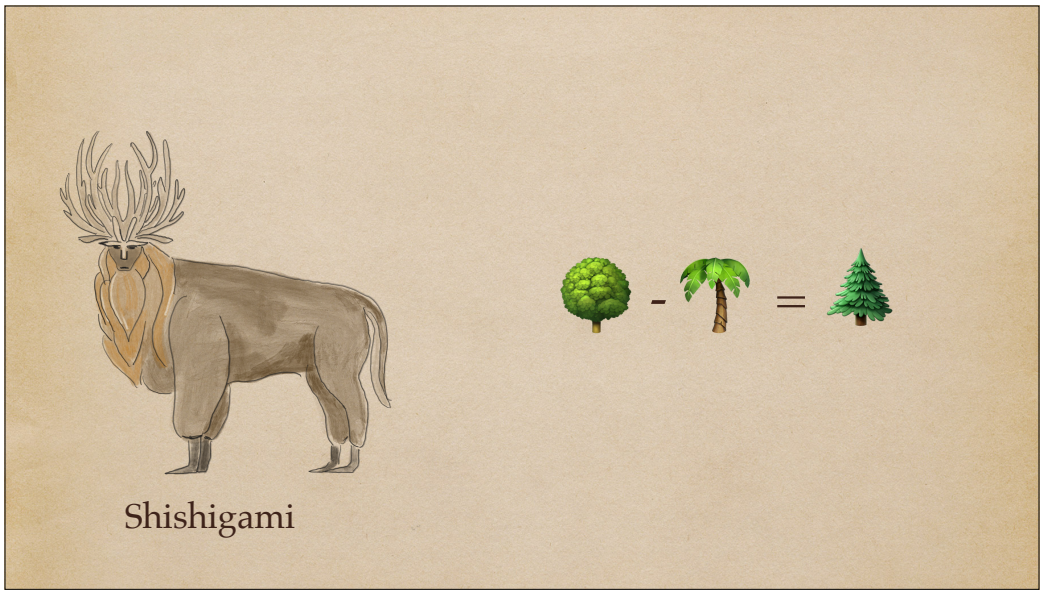


Then, it reads your package-lock.json to make sure that the tree built from your node_modules directory matches the tree from your package-lock. After that, it is time to get the metadata for each missing package.



At this point, Shishigami will call upon Pacote. Pacote's role within building a dependency tree is to reach out to the Registry to get information about each package needed to be installed.

According to the documented scriptures of Pacote, this information will be delivered via Corgi doc, which is not a small and cute dog (as represented here), but a very small piece of metadata that lets the tree worker know which versions of the package are available in the registry. Once this metadata is in place, the tree is normalized.



Shishigami then puts together the ideal tree and calculates the difference between that and the tree that has been put together from the package-lock and the metadata. This difference is where packages are moved, removed, or fetched and installed.

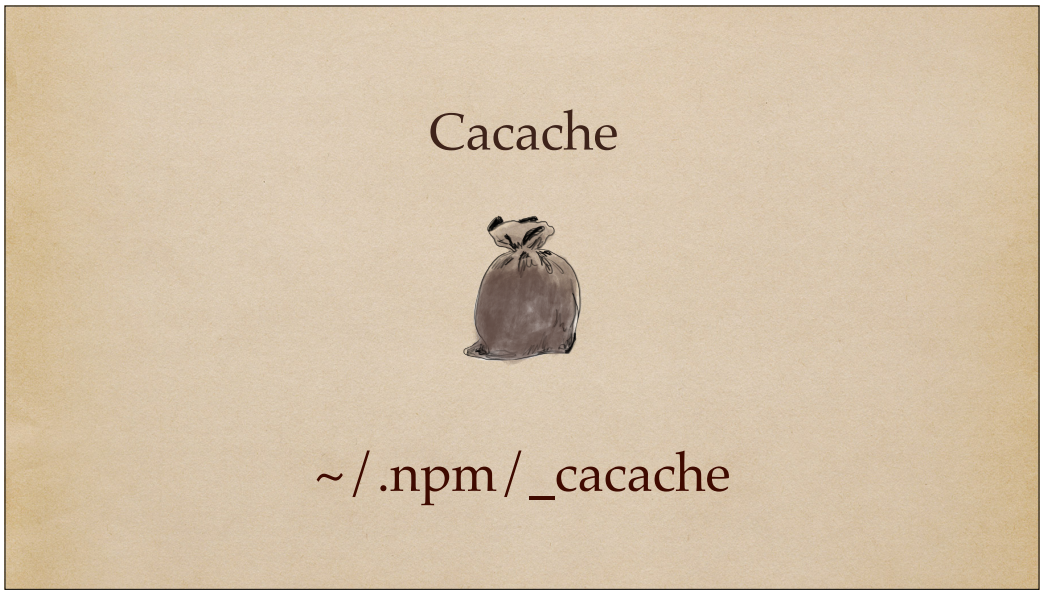


Shishigami is located within the npm repository. You go there and speak its name, it joins your party.

Book One
Chapter Two



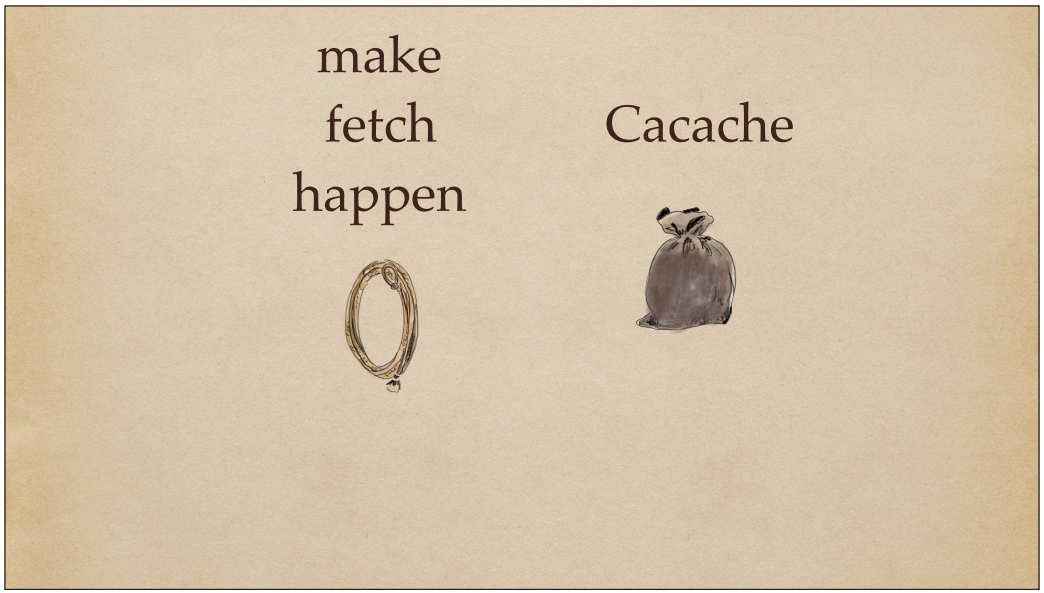
Now that you have Shishigami, you will need Pacote. Pacote is the master of packages and comes equipped with her own caching system, cacache



Cacache is Pacote's nearly endless pack of fun. It is a key value storage system written in Node.js that is immensely fast. What it does best is get you the correct data, quickly. It does this with great concurrency, incredible fault tolerance, support for sha1 and 512, and data consistency that notices if its data is corrupted.

During installation with the npm tool, Cacache stores the package data [click] on disk

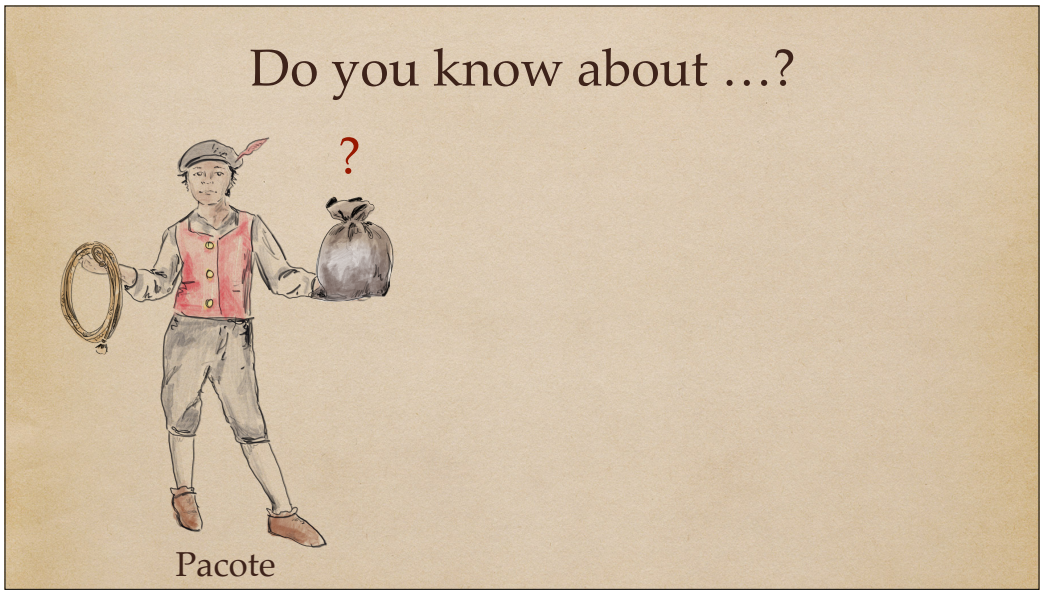
If the content is unavailable in the cache, Pacote uses make-fetch-happen to fetch the package



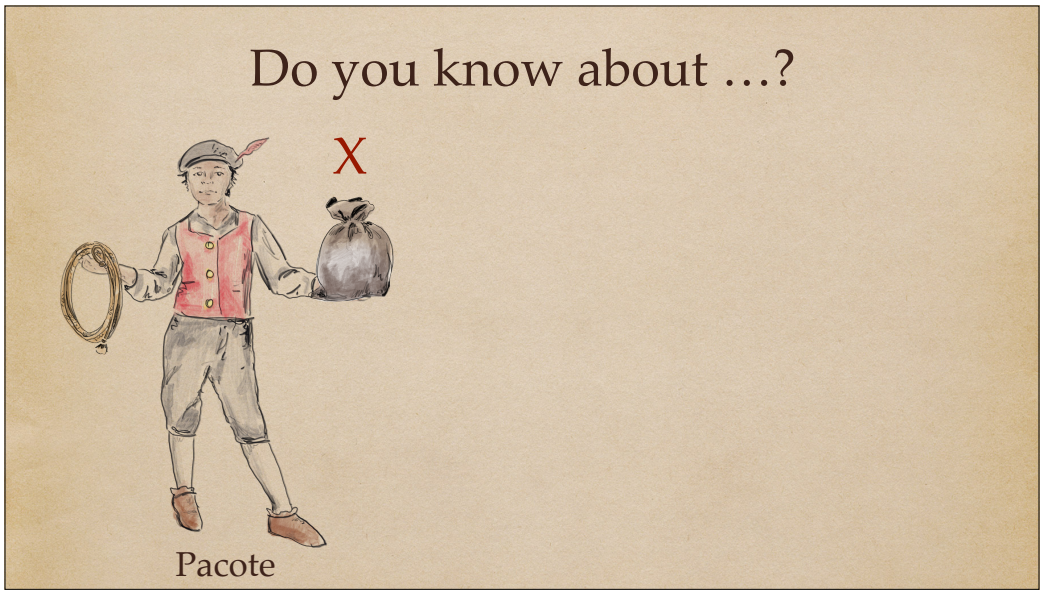
Make-fetch-happen is a wrapper for node-fetch (a node.js implementation of the browser's window.fetch). make-fetch-happen tacks on HTTP Cache support, request pooling and several more features to make it very easy to use and very fast. Its caching system relies upon cacache.



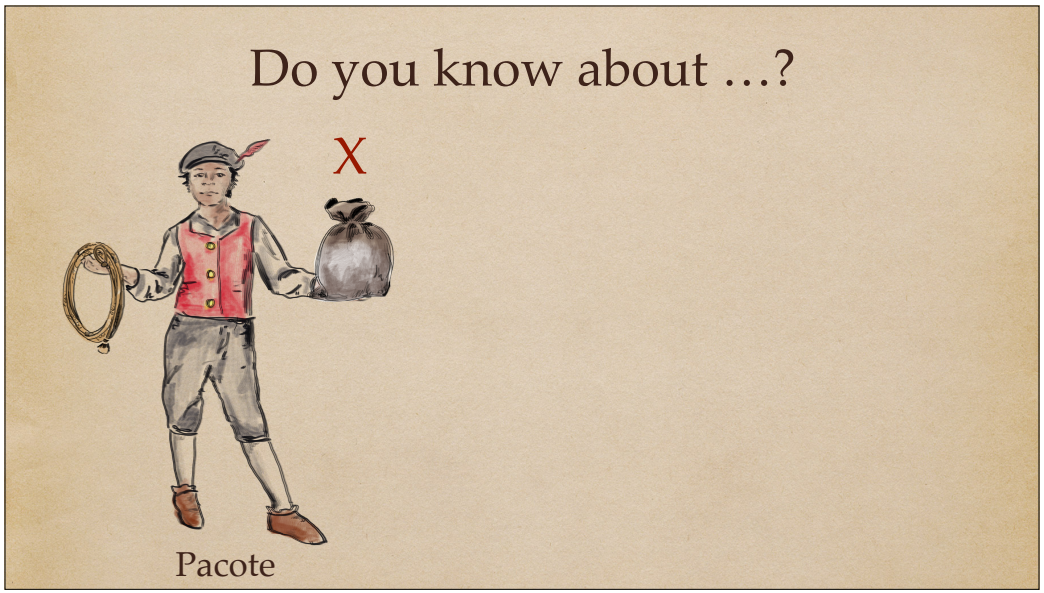
Because Pacote knows that make-fetch-happen uses cacache as its caching system, when Pacote is asked to fetch a package, it can use cacache directly to get any data that it would use make-fetch-happen to request before it uses make-fetch-happen to get the package.



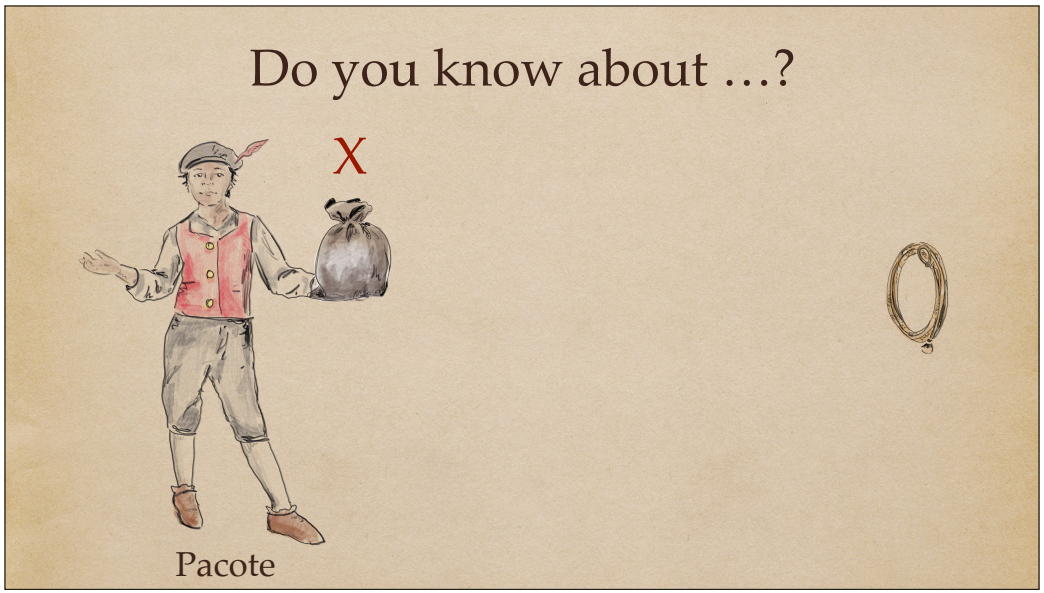
So, first it asks Cacache, do you know about the package I need?



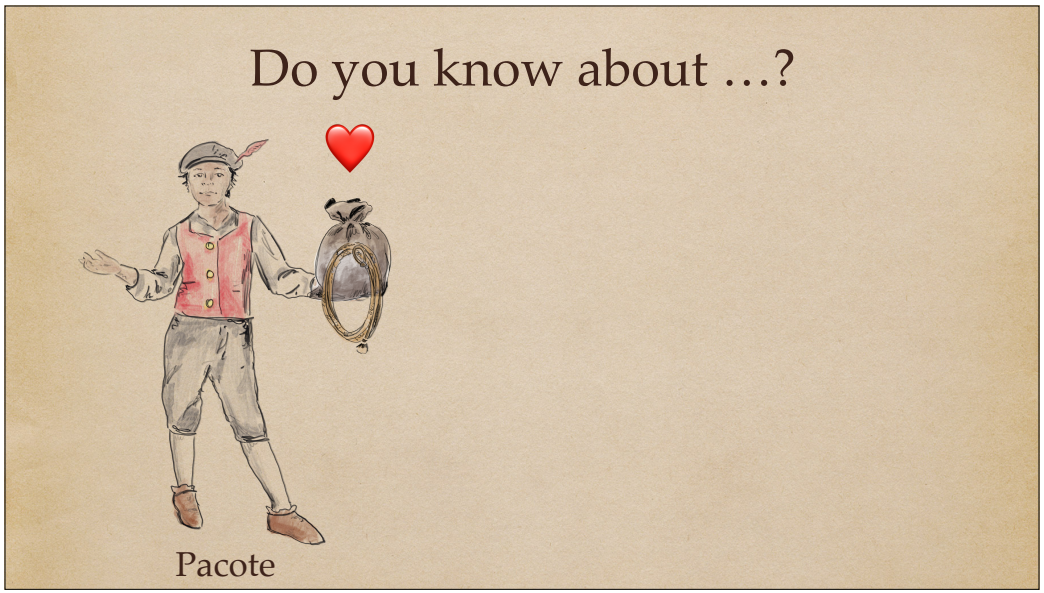
Many times, Cacache says yes, but when it says no…



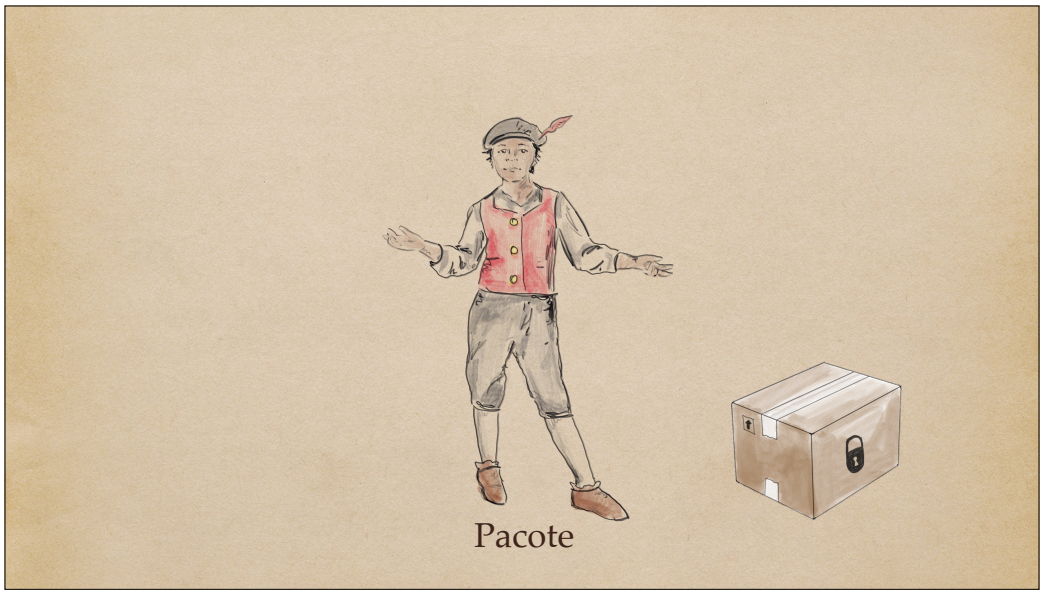
[CLICK AGAIN]



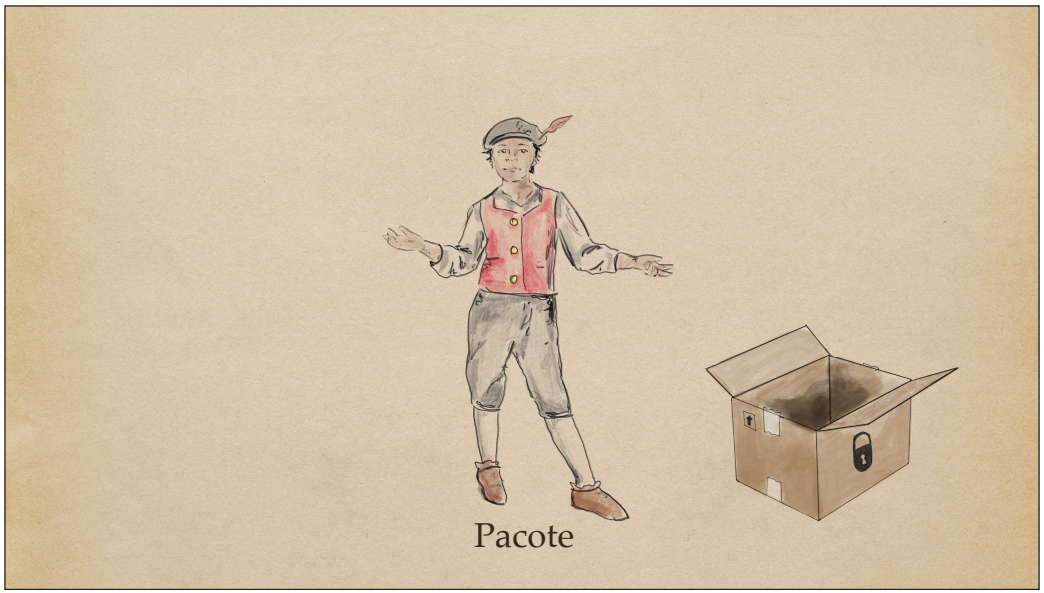
Pacote can use make-fetch-happen to go out and grab the package



Then, when make-fetch-happen returns the package, it can be stored back inside of Cacache for Pacote to get next time



Pacote also provides an extractor, in order to unwrap the packages that it is installing.



Packages come from the registry in the form of tarball, and Pacote understands how to make this into executable JavaScript for your consumption by unpacking the contents and placing that code accordingly.



And as mentioned before, when dealing with Shishigami, Pacote is capable of reaching out to the registry to grab the necessary metadata to understand just what packages you need. That metadata helps Shishigami by letting it know which versions the Registry has of the packages you need, so it can build your dependency tree perfectly.



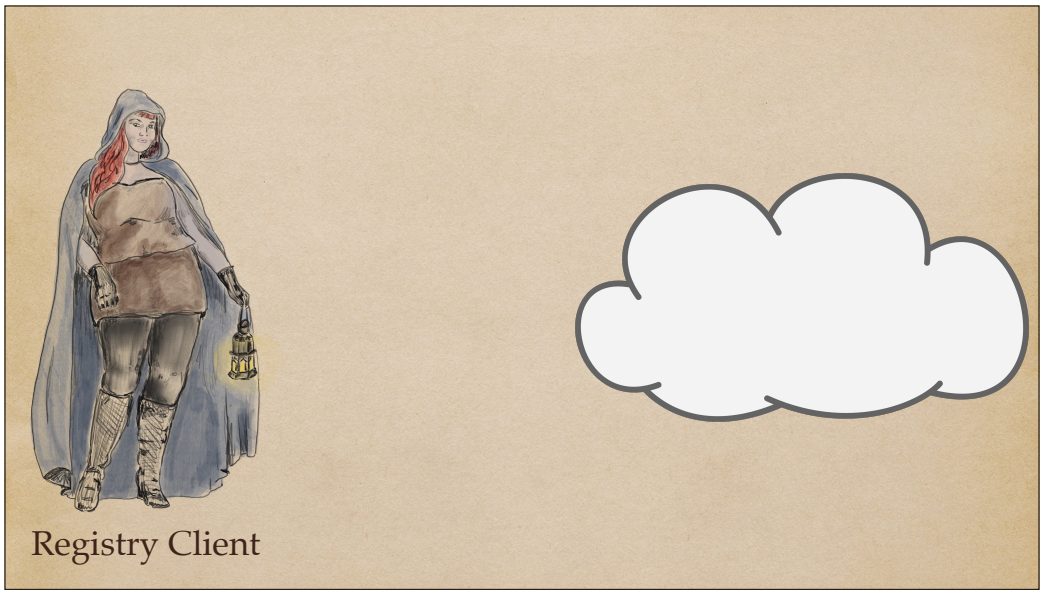
Pacote is largely the workhorse of the npm tool and lives in a repository on Github, where you find her. You ask. She eagerly joins your party.

Book One
Chapter Three

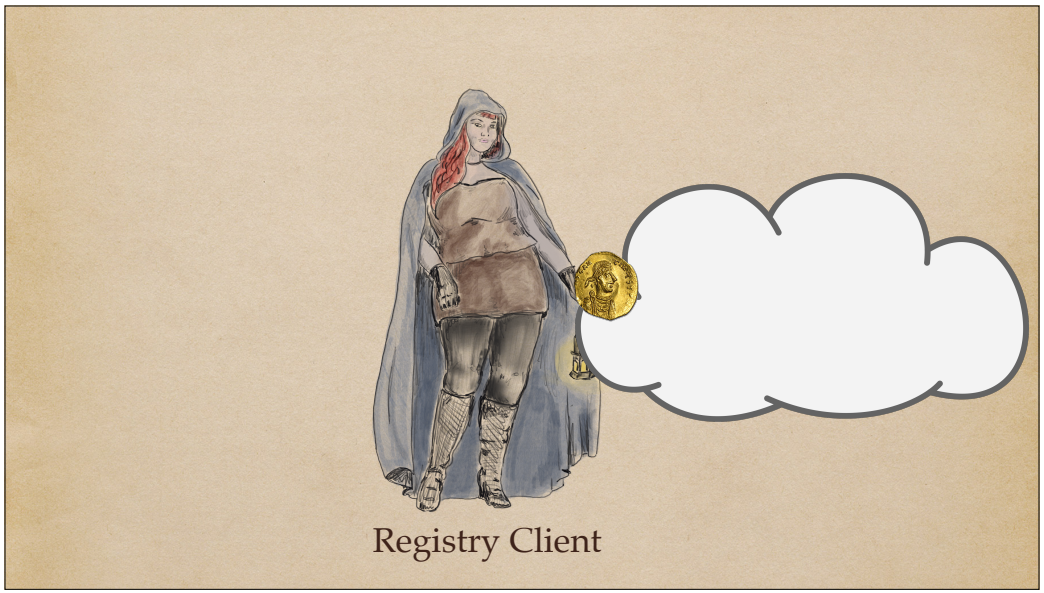


The last and final spirit you need to build the tool is Registry Client.

Much like many other API clients, the Registry Client is a wrapper around an external API. Registry client allows you to talk to the Registry for most anything revolving around your account and management of things you own. This includes changing ownership of packages, publishing and unpublishing packages, adding and removing members to and from your orgs and teams and, in this case, authentication.



When you use the registry client for authentication, it asks you for your username, password, and email address.



Then it takes that to the registry and returns with an authentication token. When you authenticate, the Registry remembers your token and you store it in your .npmrc file. [click] That token will be useful when you and Pacote use the tool and go to get your package.



You find Registry client lurking behind a corner on Github, you pass it a note saying “Will you join my party?” It nods its head and follows behind you.

Book One
Chapter Four



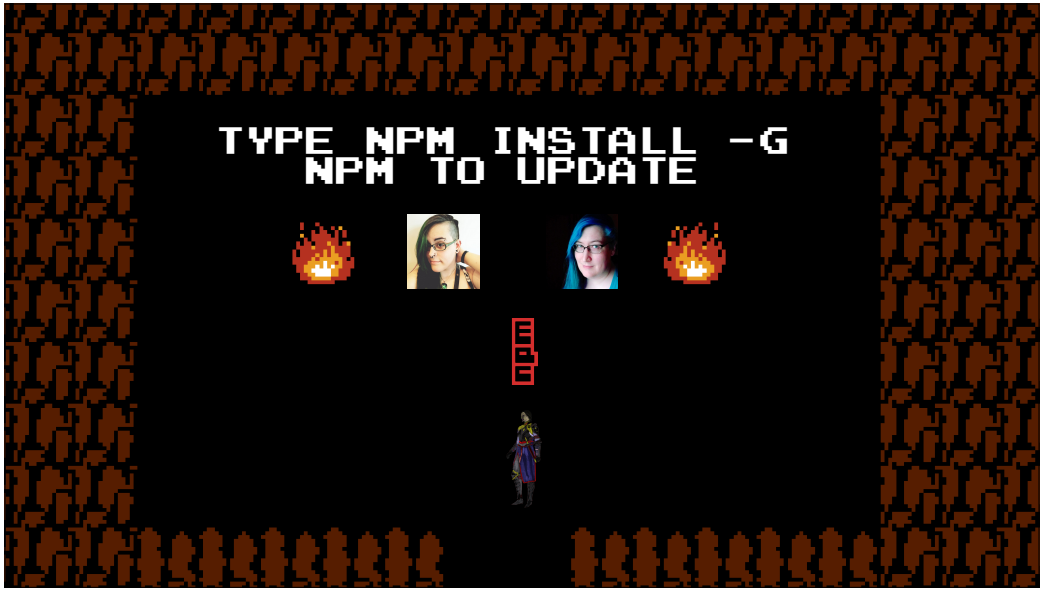
Pacote, Shishigami, and the Registry Client have guided you to npm, inc. Home of the sorceresses of brightly colored hair. They introduce themselves and walk you and your new party members to the Installer.



Within the realms of The Installer, they forge together the spirits of your party members to be called upon any time you desire.
[click]



Their magic is strong and they have added to the tool the ability to increase its level to learn new skills and talents along the way, while simultaneously making the quest to obtain your packages smoother.

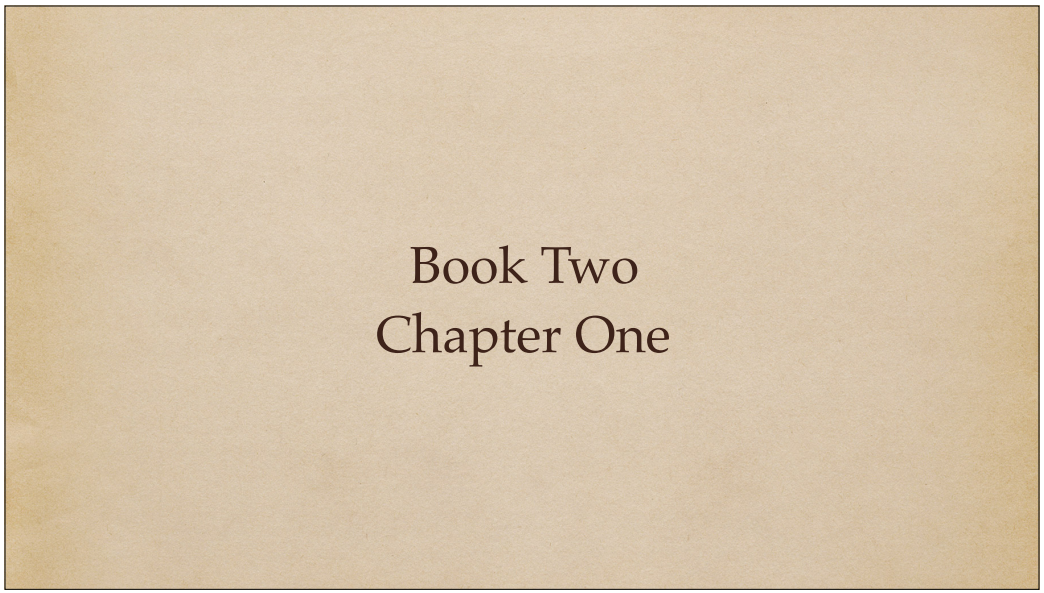


You can do this by telling the tool to install itself. And as of today, you will get version 6. With new functionality like an improved update command, a new npm view, a new init (which can take an initializer like create-react-app), and much more!

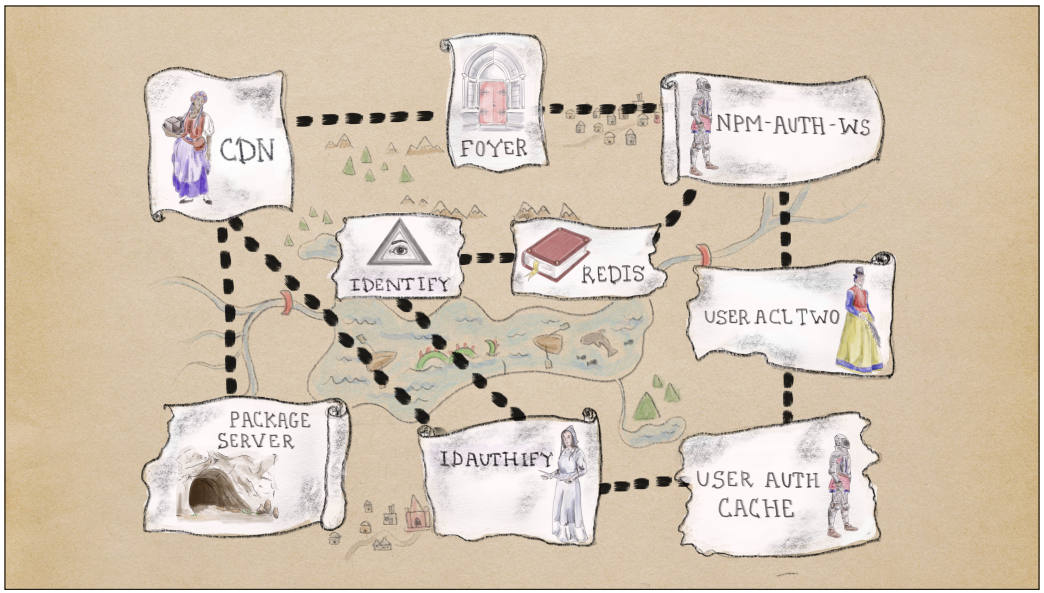


CLI. CLI. CLI. They chant.

The tool is yours, go, login, then install.



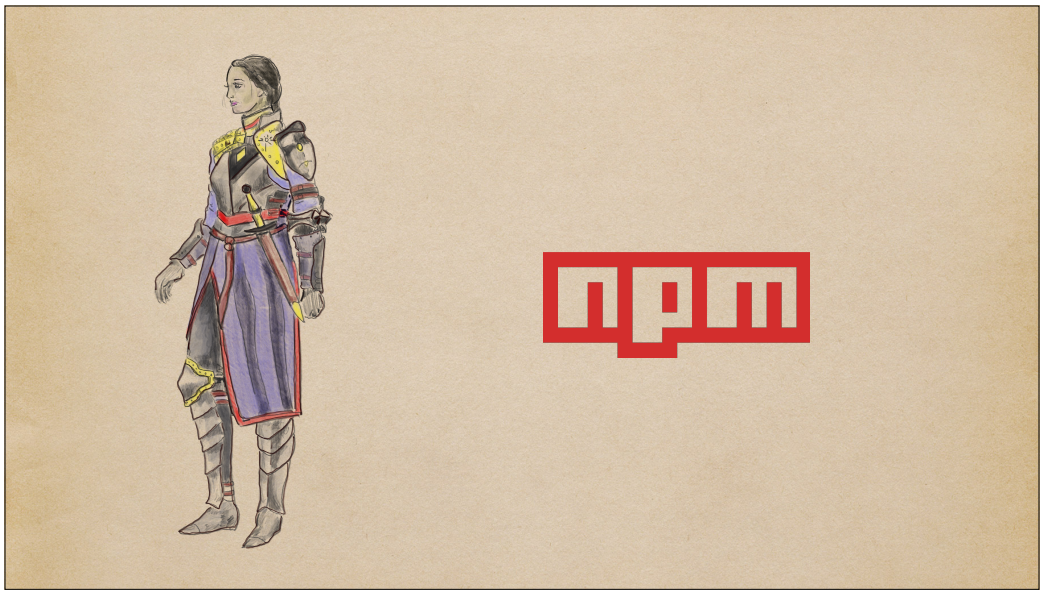
10 min



With the mission to forge the CLI complete, we turn to our map to see where the path forward leads. All signs point to the great registry: registry.npmjs.org. Let us make the journey.

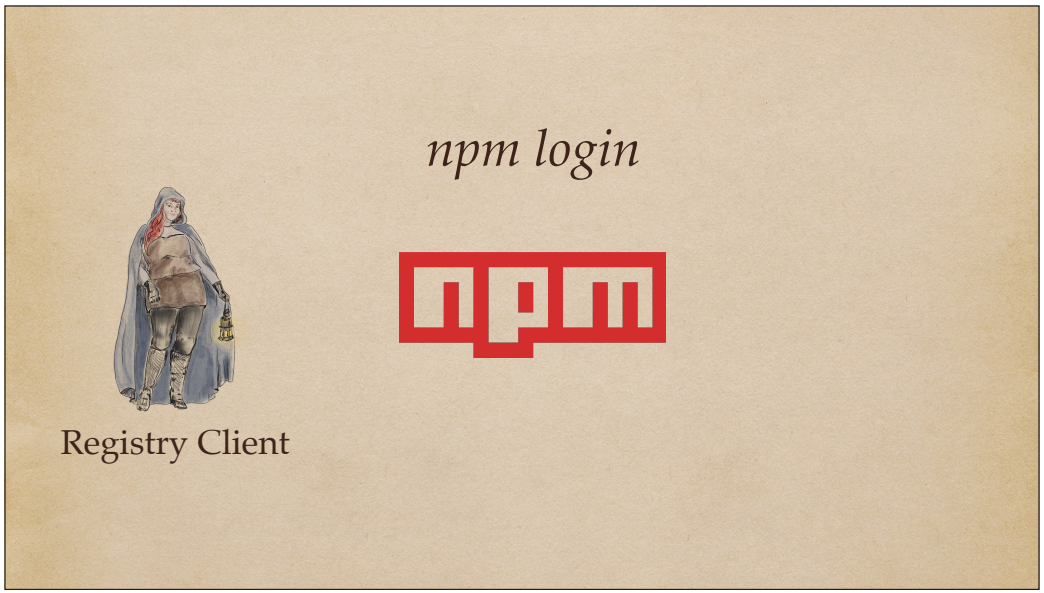


You start your quest at the land of Terminal Pointe



First, we login.

You pull out your CLI tool.

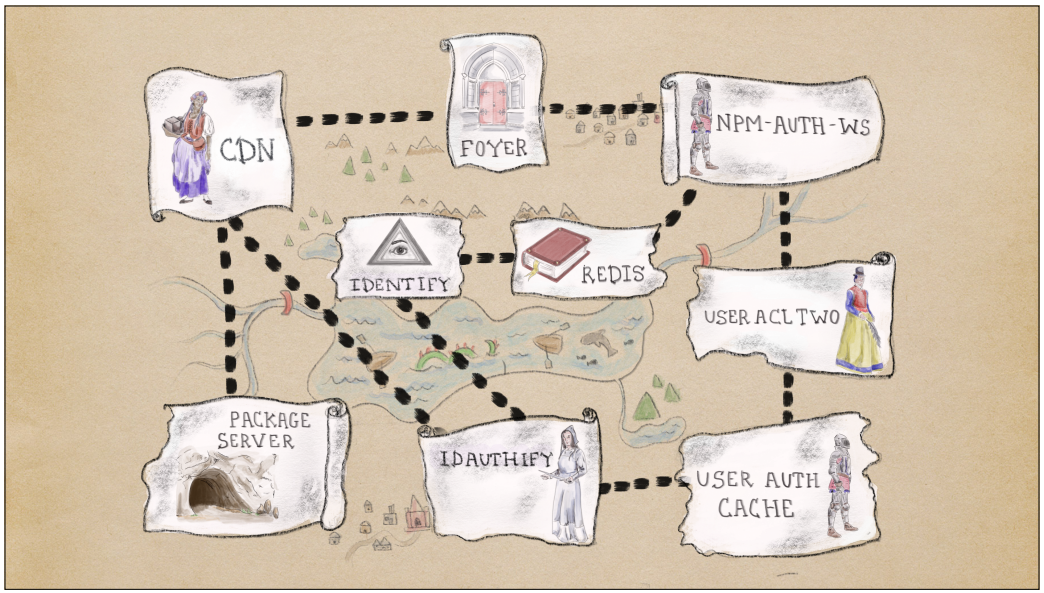


You speak onto it, `npm login`. [click] The spirit of the Registry Client emerges.



> “Username” > “jlembeck” > “Password” > “A very good password probably” > “Email” > “jeff@npmjs.com"

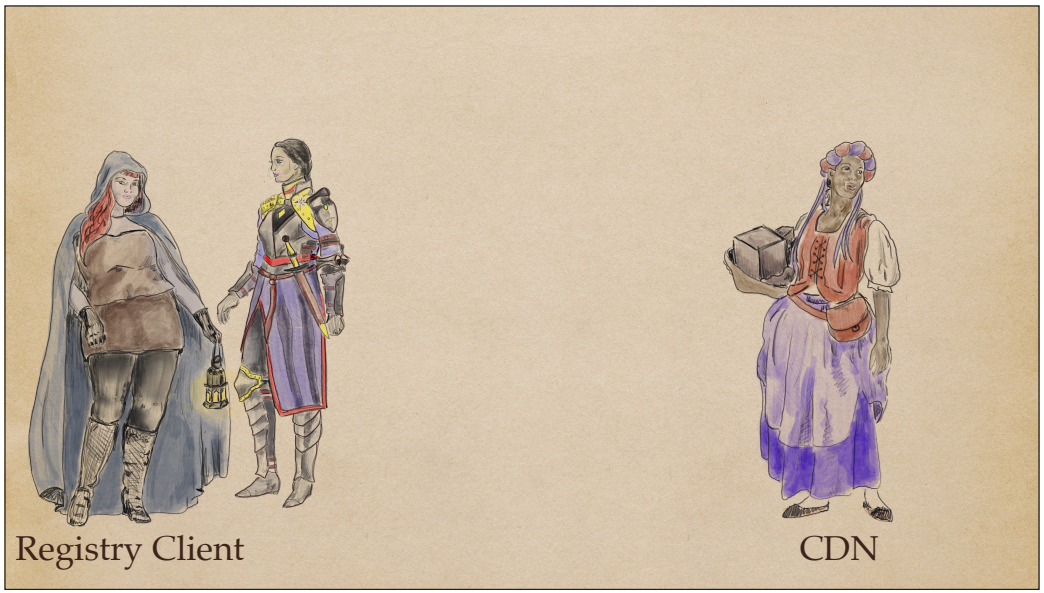
And with that, you are whisked away.



We ride the road of the great network to the first destination on this request, The Shopkeep.



She goes by the name of CDN. CDN is a class who specializes in speed. She and her sisters (all with the same name and same personality… just different locations, think Officer Jenny or Nurse Joy) spread the globe and position themselves in ways so you'll get to them far more quickly than if you were to reach out to just one part of the world. She handles the cache for packages and if you approach her with the name of the package you are looking for, she will do her best to get it for you.



Unbeknownst to many, CDN also provides traffic routing. She handles all requests to the Registry and will forward them along to where they need to go.

How she does her work is largely based on two things: her cache and a light routing layer written in VCL.



CDN looks at you.

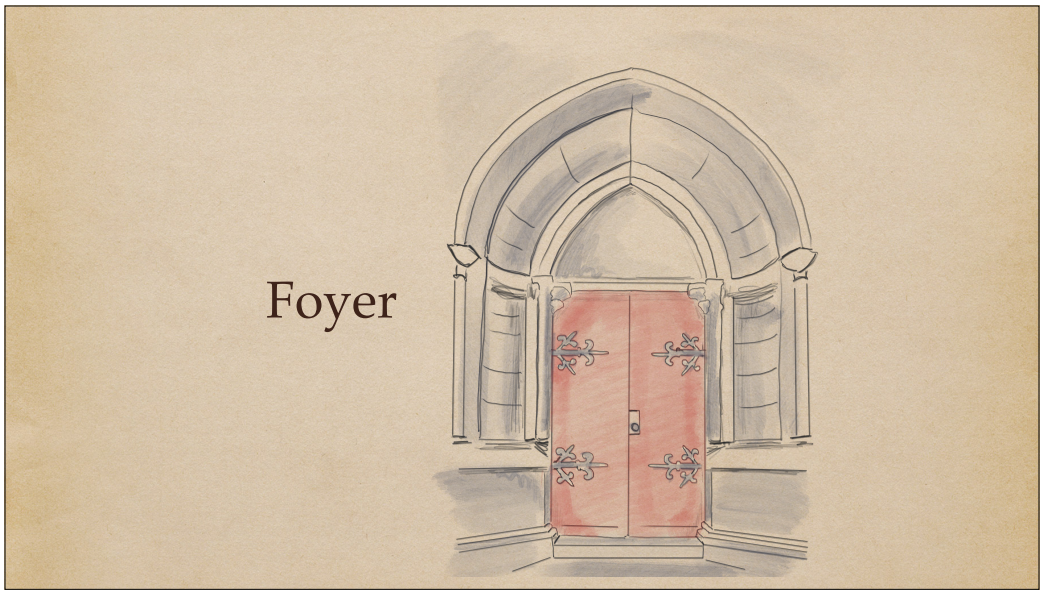
> What brings you here? > Authentication > Ah, let me see: Right this way. (DO NOT CLICK)

CDN guides you to a path

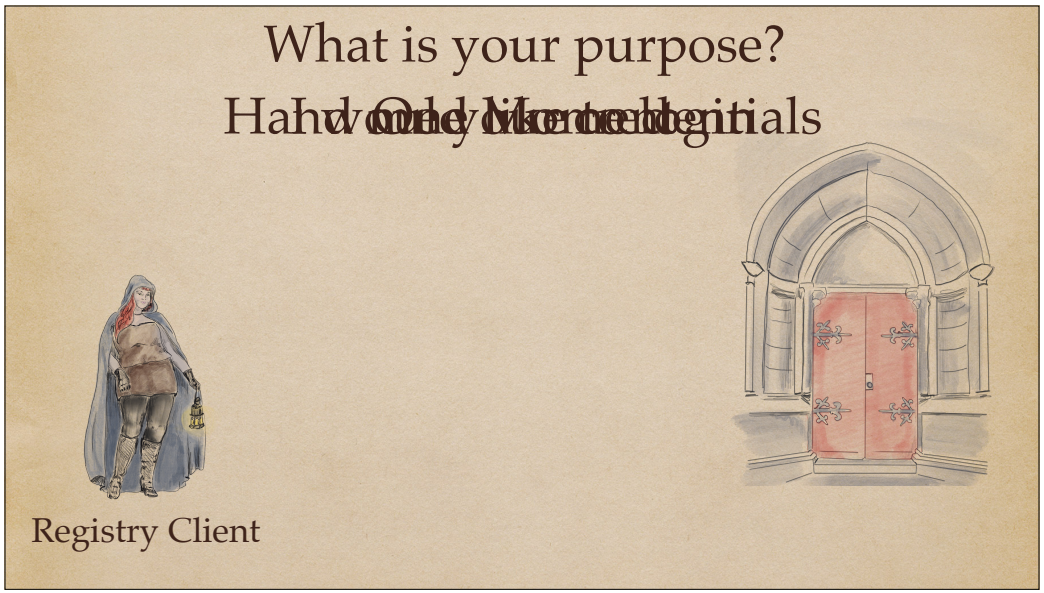
> Ahead lies Foyer, go.

You are whisked away to the next destination. (NOW YOU CAN CLICK, IDIOT)

Book Two
Chapter Two



Foyer operates as the entry point to the npm registry. Information about packages, users, orgs, teams, and any modification to those things must first go through Foyer. Foyer also handles the path to authentication.



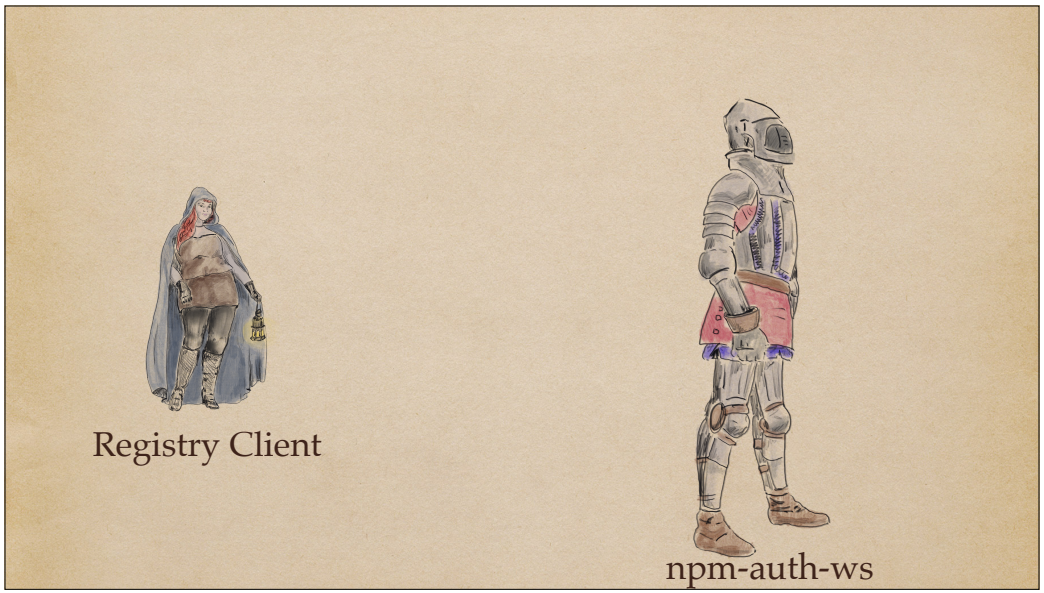
You and Registry Client knock on the doors.
From behind the walls you hear: > What is your purpose? > I would like to login > Hand me your credentials
[click]
> One moment

The guard behind the door calls for a messenger, the npm-auth-ws client and tells them to take this information and be on their way.

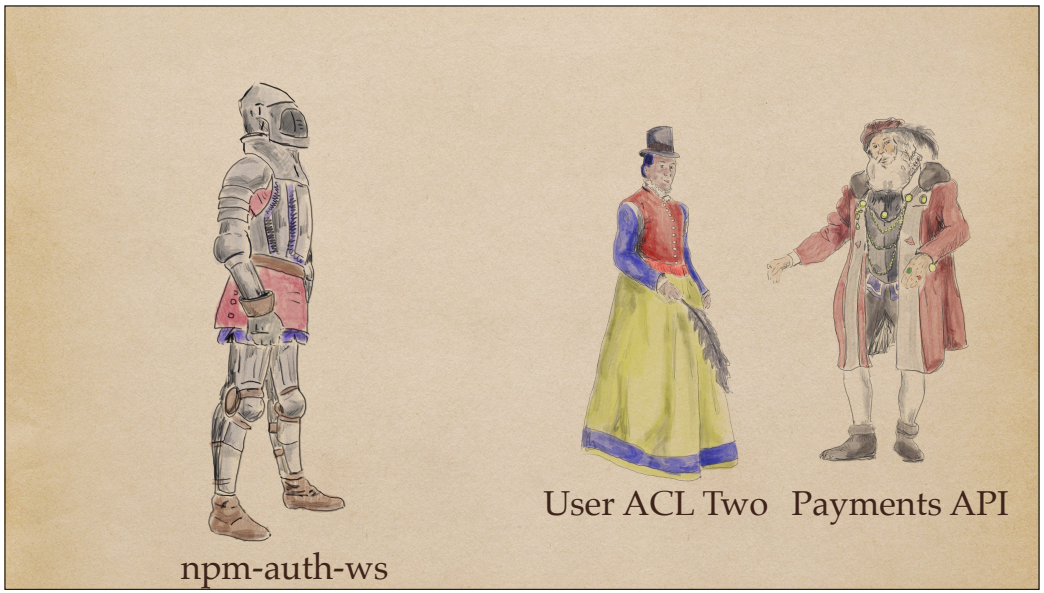
Book Two
Chapter Three



npm-auth-ws is a gatekeeper for proving who you are and where you can go. She is quiet and only speaks with messengers from behind the doors of the Foyer.



npm-auth-ws will be handling your authentication today. Authentication being when you prove who you are. This is versus Authorization being when you are checked to see if you have access to something. Note: _Saying just the word “auth” is confusing. Stop it._



To check whether you’re authenticated, npm-auth-ws will reach out to the Synecdoche API



The Synecdoche API is the glue that binds together the two Beholders of Truth: user-acl-two and payments api. Its API acts as a proxy for user-acl-two, and when requests come through that gather or change information about payment, it also makes a call to the payments-api.



As the npm-auth-ws reaches out to login, the request is proxied through Synecdoche API to user-acl-two.



User ACL Two is the grand beholder of truth. If any changes are made to any accounts, it is written by user-acl-two, and then that information is sent elsewhere via NSQ messages. Any of the calls to Foyer from the registry client that might involve changing teams, organizations, package permissions, or users? User-ACL-Two



And, most relevant for this story, logging-in happens in user-acl-two.



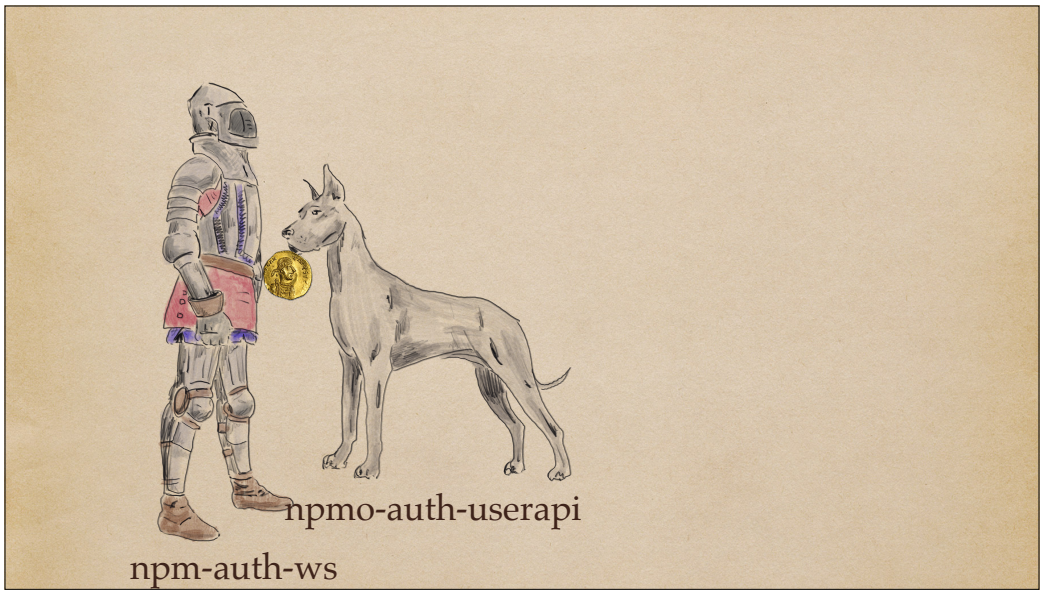
npm-auth-ws's trusty client friend npmo-auth-userapi



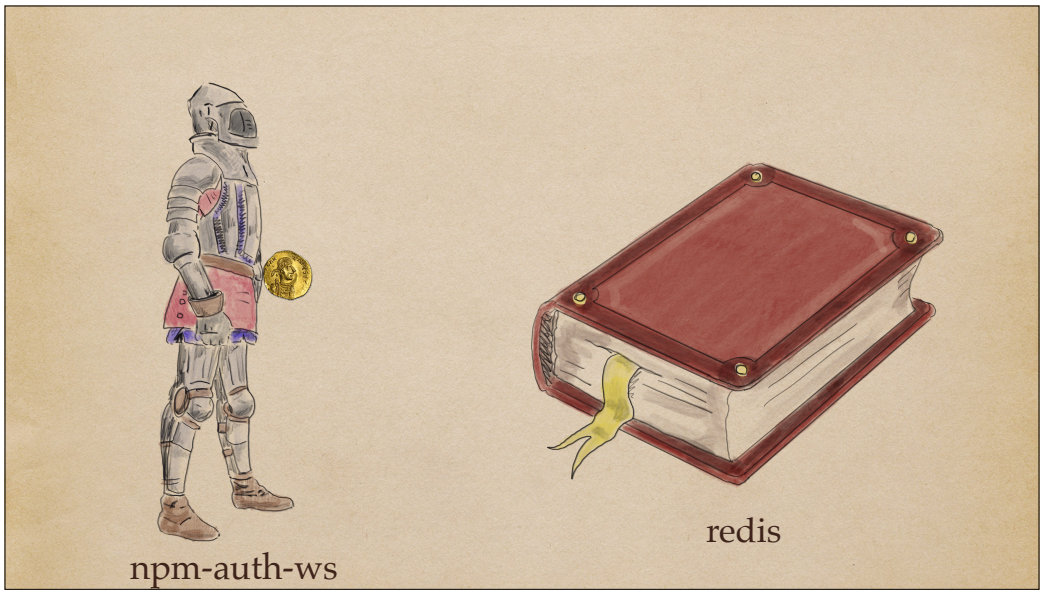
passes the credentials to user-acl-two for your logging in and it is successful.



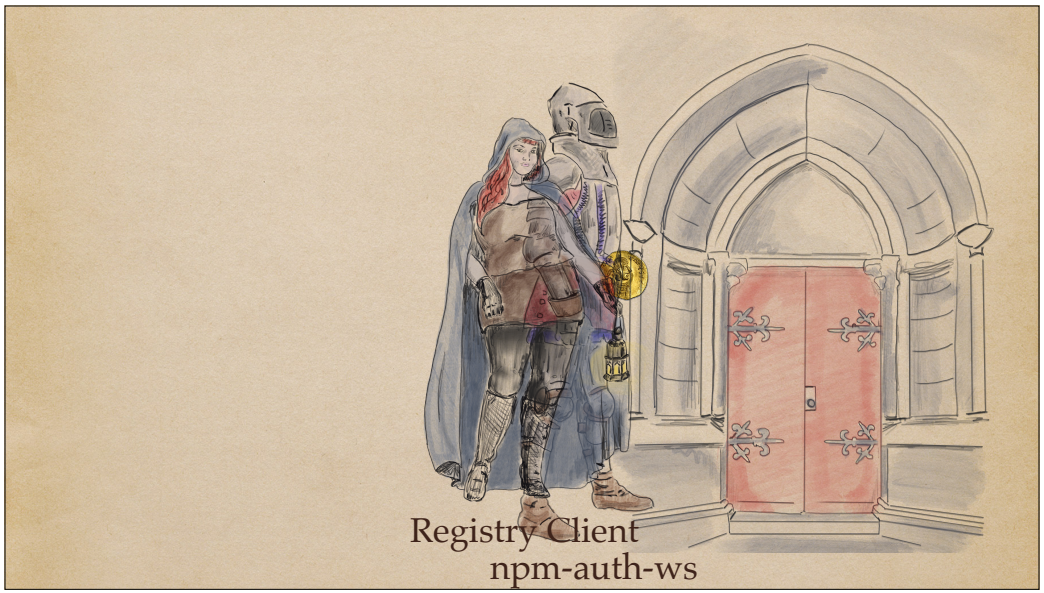
user-acl-two responds back with a token.



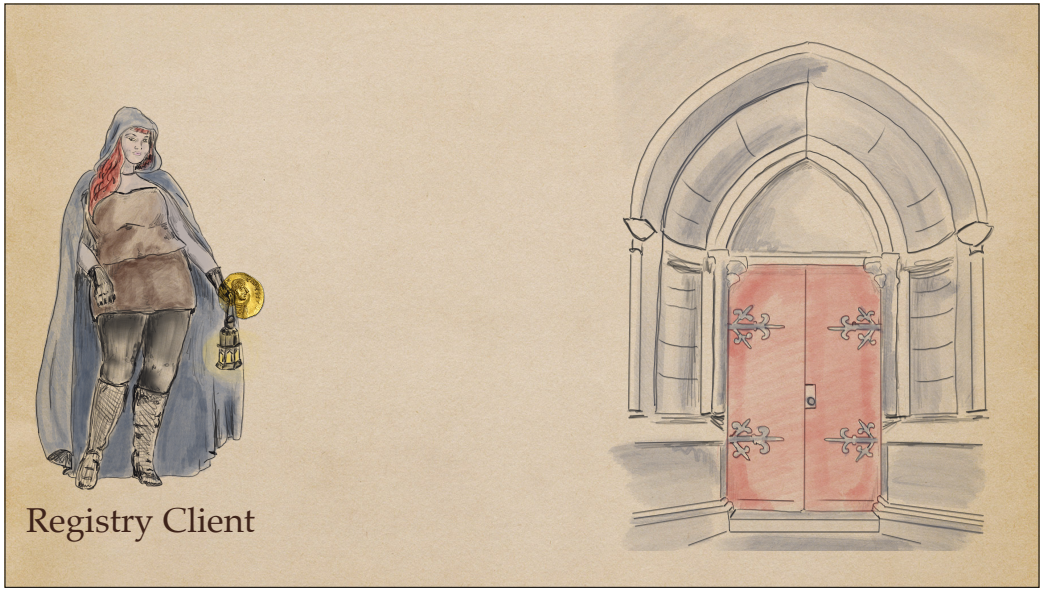
This token is passed back through the clients and to npm-auth-ws, who takes your token and saves it…



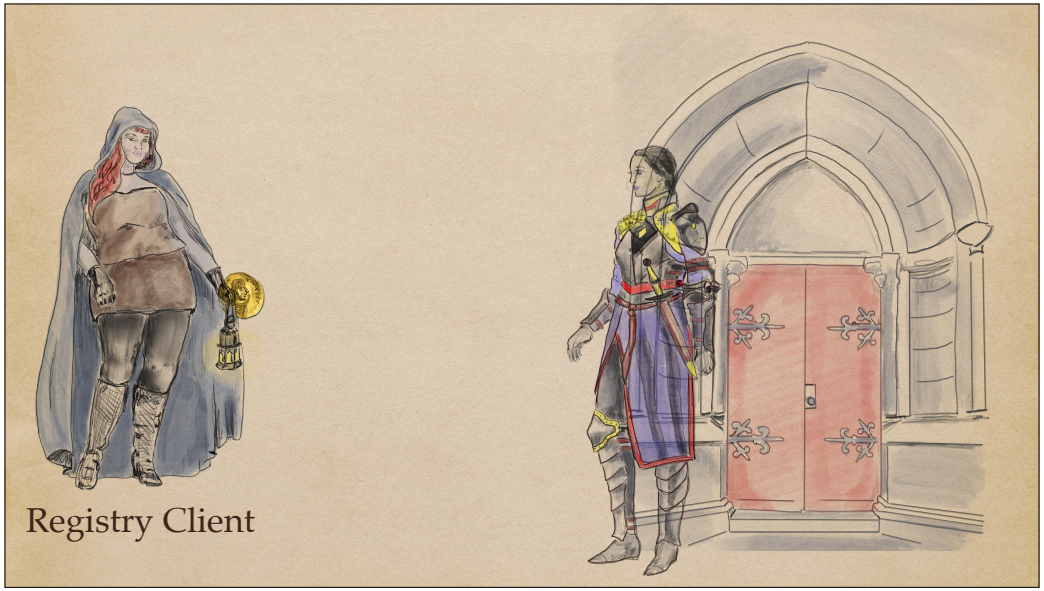
Into Redis, the great dictionary of keys and values. It will use this token in any instance where you need to access the registry again.



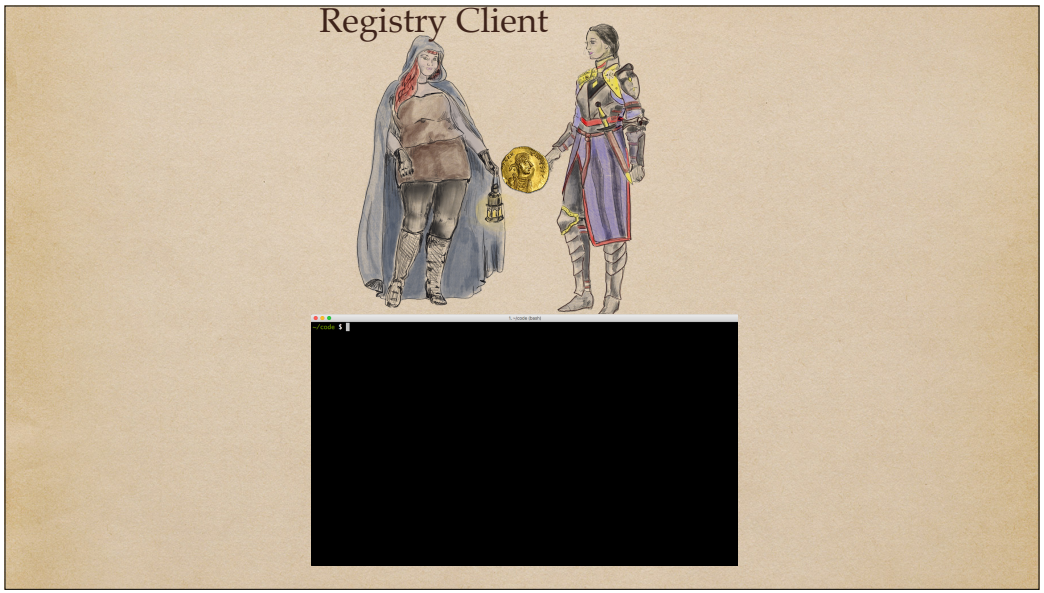
Once it has done that, it passes the token back to Foyer. The doors open and a hand slips out with the token attached.



Registry Client takes it, walks back to you…



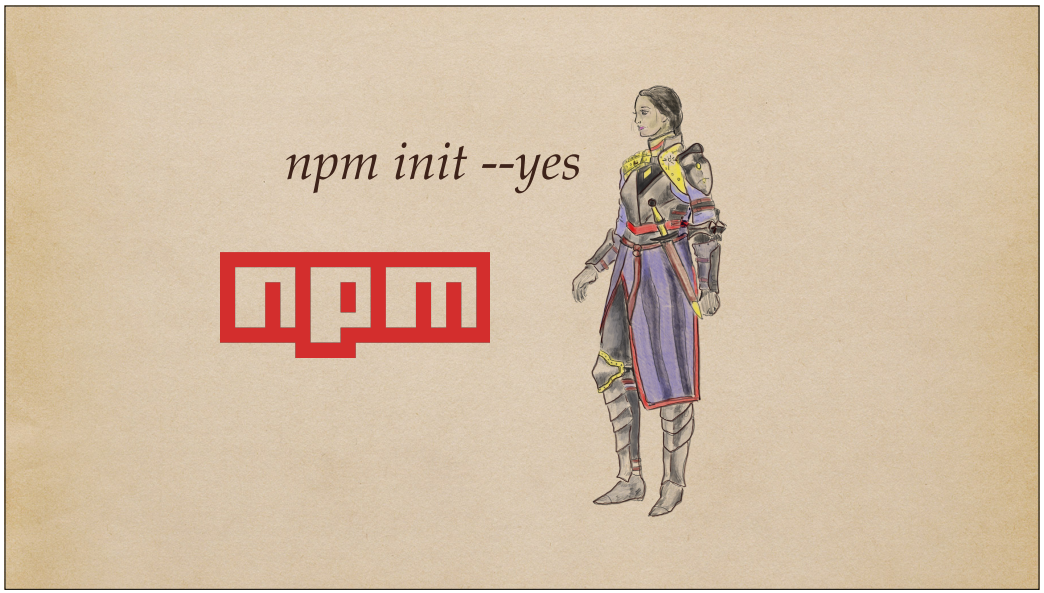
grabs your hand and you transport back to CDN and then back to Terminal Pointe.



Your token is written in your .npmrc file.

You are now logged in.

Book Three
Chapter One



Now that you are logged in, it’s time to install.

You look around and are surrounded by nothing. Terminal Pointe is empty with the exception of you and the CLI. Your mouth starts making a sound, like it had made it thousands of times before. [click]

npm init —yes

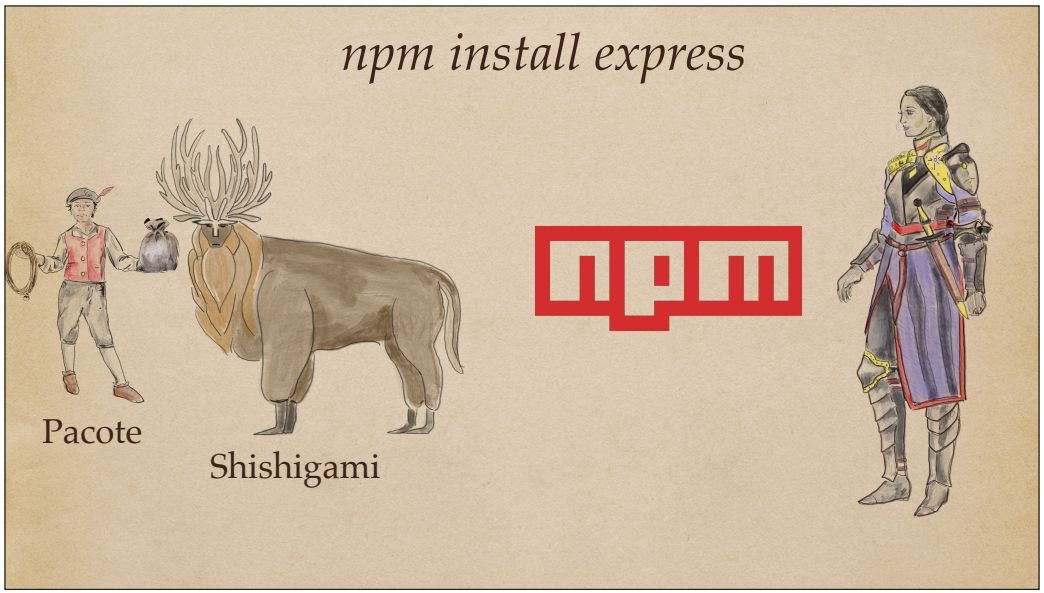


You’re not sure where that came from or where you’d heard it… if you’d ever heard it? But there it was.

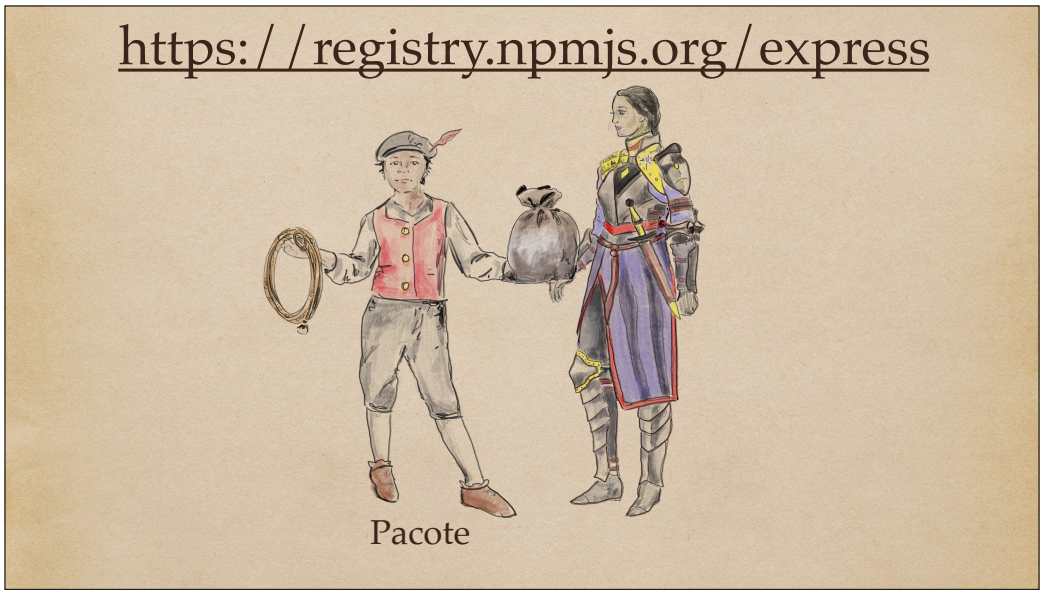
And now, in front of you, is a package.json file.



You look at the CLI once again and say [click] “npm install express”



Shishigami arrives and looks around, seeing no node_modules folder and no package-lock.json, Shishigami whistles and Pacote arrives, carrying cacache over her soldier (or, in her hand). They speak quietly.



They put together the address.

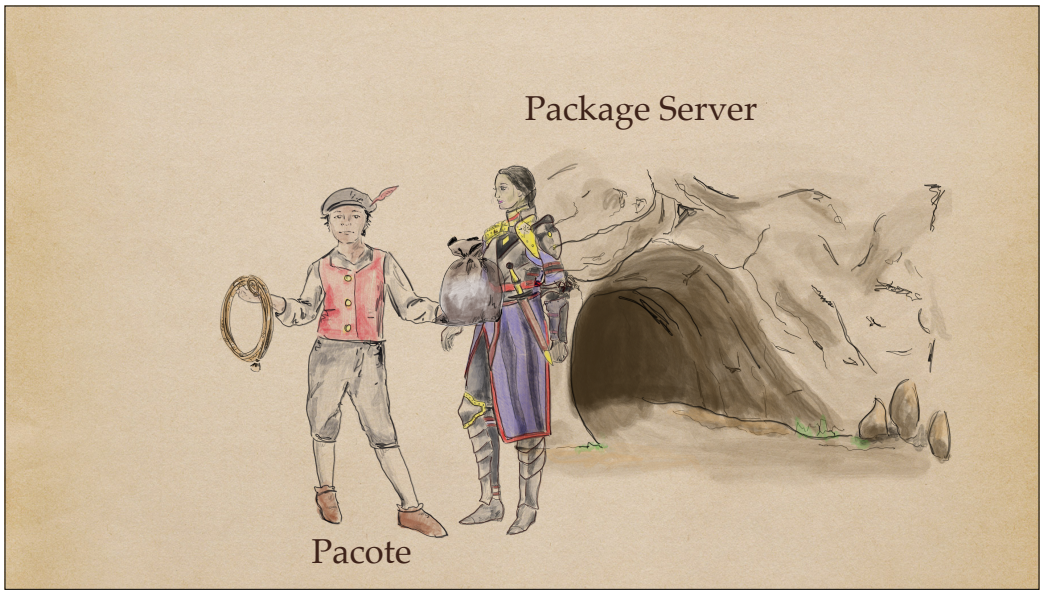
Pacote grabs your hand and you disappear again, back to CDN.

Book Three
Chapter Two



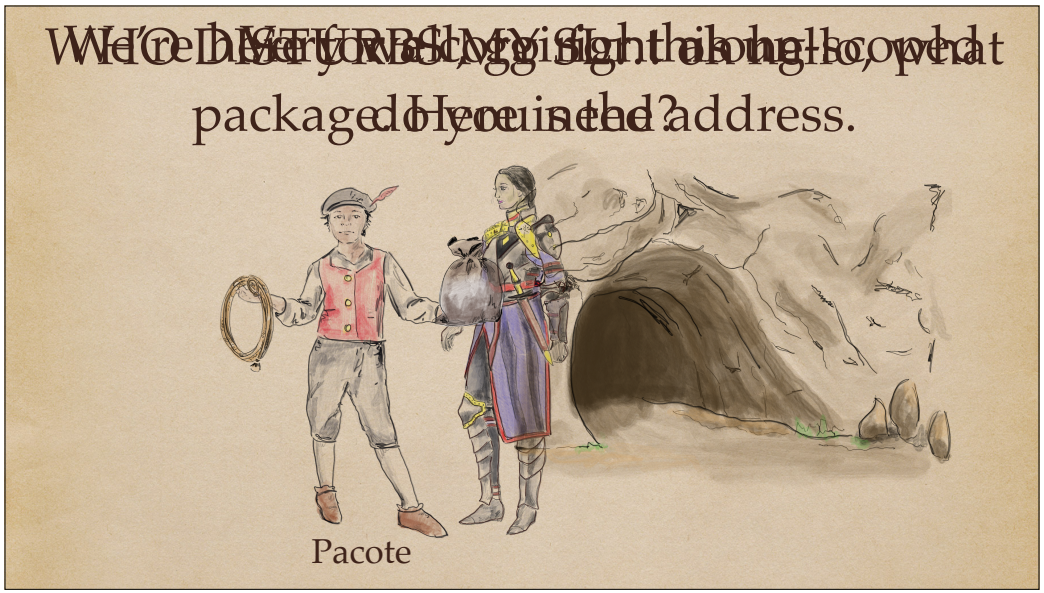
Once again, you walk up to the shopkeep. Pacote speaks

> Check my accept header
> Ah, I see you're here for
>>the Corgi for an unscoped
>>> package. Excellent, go along



You remember from before that a Corgi is a metadata file for the package requested. It's significantly smaller than your standard JSON representation of the package and it includes dist-tags, version numbers, and each version's dependencies.

Since the package is unscoped, it is a public package, so there is no security to worry about, CDN sends you to a specific address in the cave of wonders



You arrive at the Cave of Wonders, an s3 or google cloud storage instance that is the home of packages and their information, served right from the disk

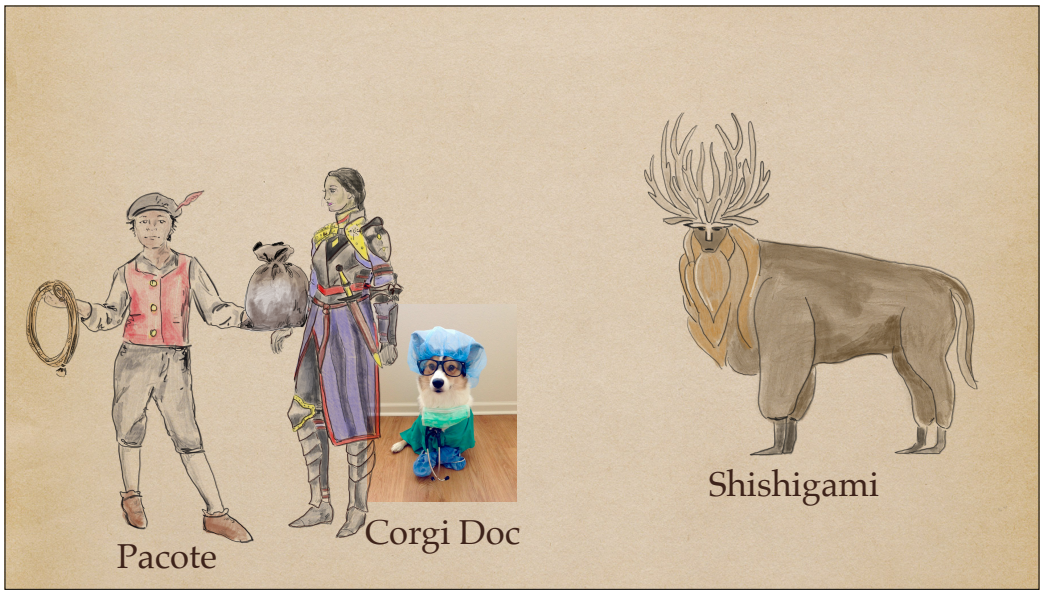
> WHO DISTURBS MY SL… oh hello, what do you need?
> We're here for a corgi for this un-scoped package. Here is the address.
> Very well, go right along



You get your corgi



and take it back to CDN, who then caches it - for quicker retrieval the next time it is needed. After that, you head back to Terminal Pointe.



Once arriving, you and Pacote hand over the corgi doc to Shishigami, who reads it and then sends the two of you back for the corgi of each dependency. This occurs until all of the corgi docs are retrieved.



Once Shishigami has all of the docs necessary, it produces a dependency tree.



YOU GAINED A DEPENDENCY TREE



Now that you have your dependency tree, the tool begins glowing again.

It calls upon Pacote to get the packages. Pacote sees the versions and packages needed, and uses those to create new routes to request. This time, it’s tgz files, tarballs.



Pacote checks inside of her bag, hrm. It’s empty. She pulls out make-fetch-happen, says a few words, grabs your hand, and once again, you’re on the road to CDN.



Tarballs are CDN’s specialty. Exciting!



Pacote walks up to CDN

> It says here I need to reach
>> This address
> Oh, let me see if this is in my cache…
>> It is! Here you are.
> Thanks!

And you’re on your way back. That was… faster than expected.



CDN's caching system stores cacheable tarballs at its points across the globe. This means that public packages that have been downloaded by other users can be saved after they are retrieved from the package servers. This makes for an extremely quick experience.



Pacote's make-fetch-again stores the package away for you in cacache. "Just in case you need it again"



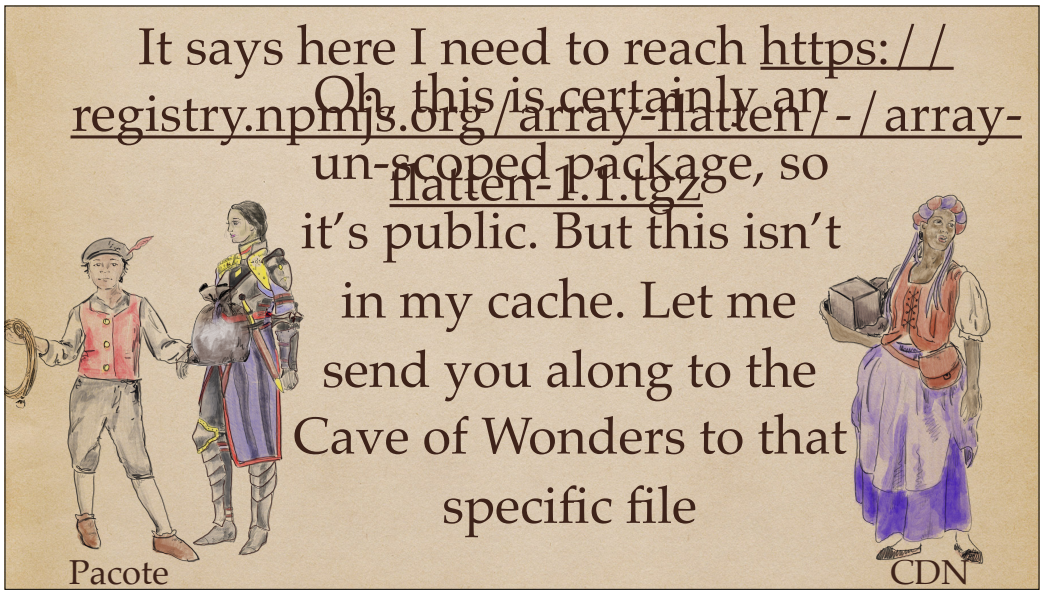
Onto the next package, array-flatten, version 1.1.1 [click] and back to CDN you go.



On the way, you ask Pacote “Do you really make one round trip at a time?”

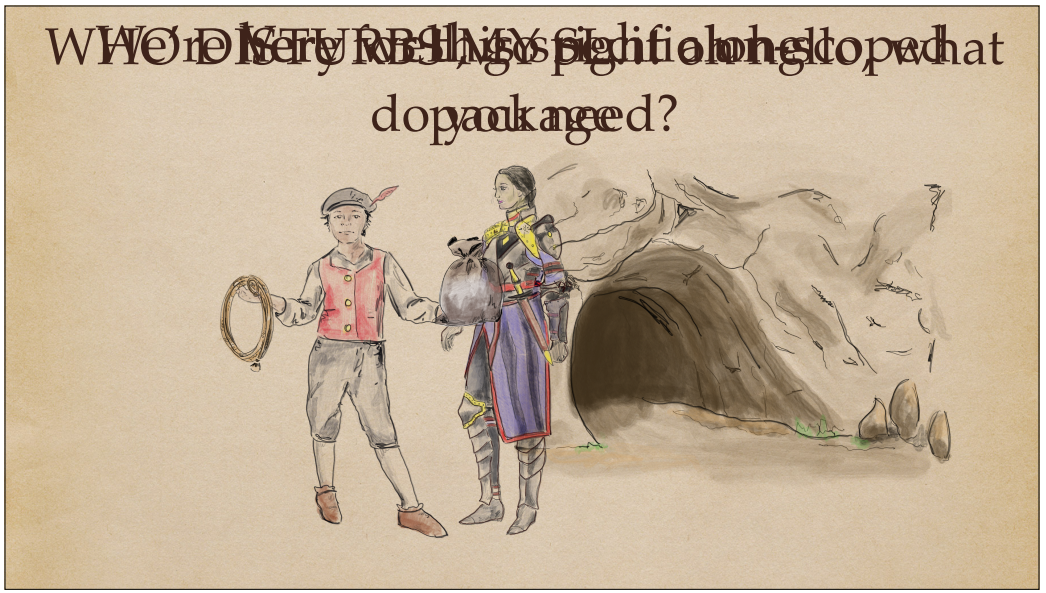
Pacote laughs, “Oh dear developer, this is Node.js - I make all of these requests in parallel, to the best of my ability. I’m walking through these serially for your learning benefit”

You nod.



> It says here I need to reach this heck of an address

> Oh, this is certainly an un-scoped package, so it's public. But, this isn't in my cache, let me send you along to the Cave of Wonders to the specific file you are looking for



You arrive at the cave.

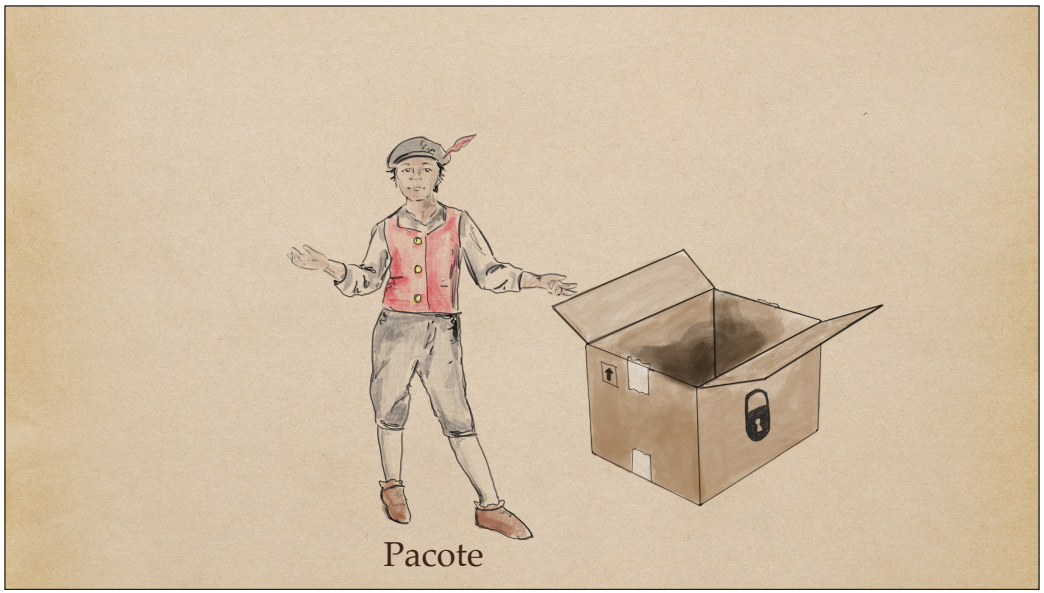
> WHO DISTURBS MY SL… oh hello, what do you need?
> We're here for this specific un-scoped package
> Very well, go right along



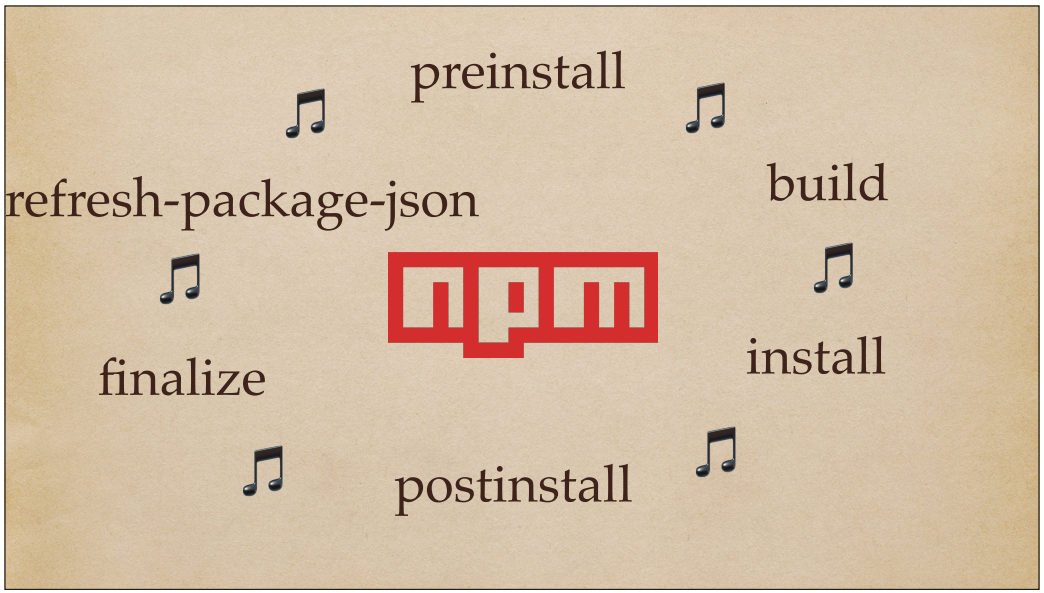
You grab the package and head back to CDN, who asks if she can make a copy, in case somebody else wants this later. Then you head back to Terminal Pointe.



This continues through all of these public packages. Pacote checks cacache, does not have the necessary package, [click] heads to CDN, she either has the package or not - if not she sends you to the closest package server and makes sure to cache the package on your way back.



Once each of the packages has been downloaded, Pacote begins the extraction. The tarballs are untar’d and placed into the node_modules directory to match the tree.

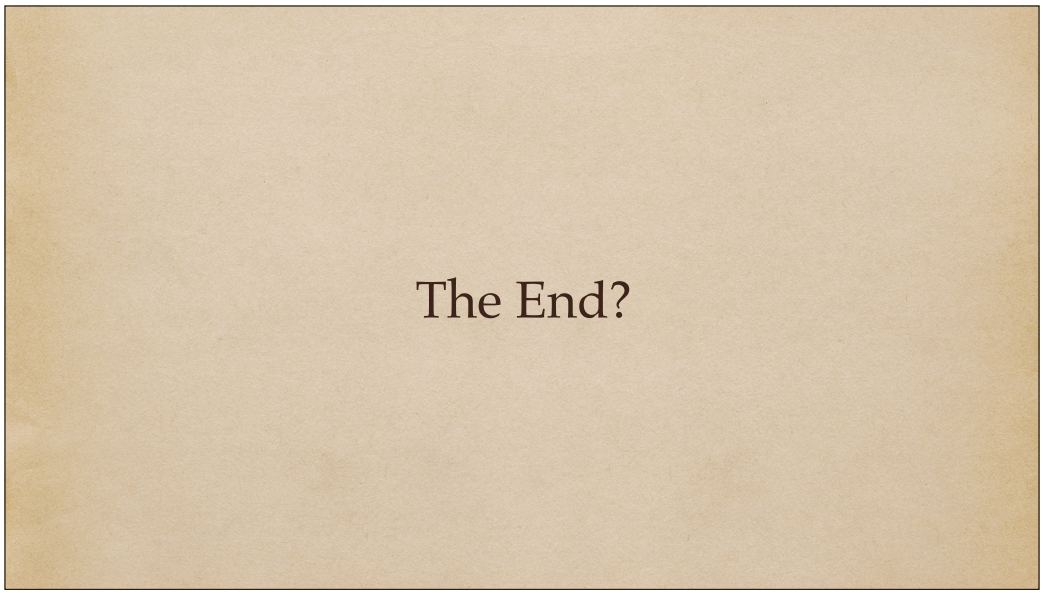


The tool glows again and begins to run its lifecycle scripts. You hear a song sing the words:

preinstall… build… install… postinstall… finalize… refresh-package-json



Your package.json is changed, all of your files are properly placed and built and they are now requirable from your project and you can use them!



The end?

Book Four
Chapter One

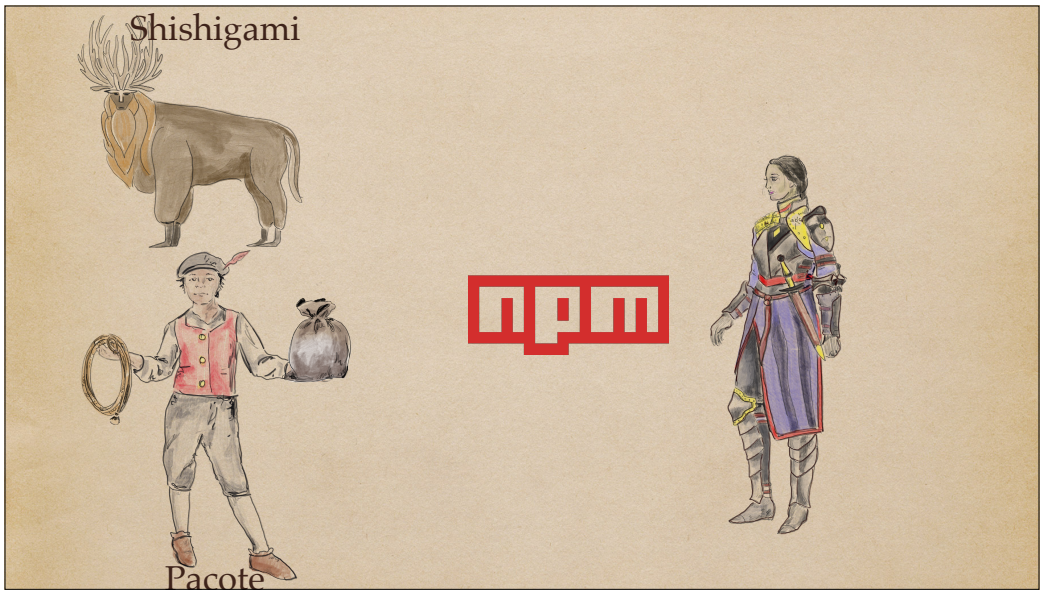


All things are peaceful again, you can move quickly and build your web application now that you have a framework in place - but you need another package, and this time - it’s a private one for your org.



You pull out your trusty CLI and speak into it:

npm install @fantasyworld/super-secret-code

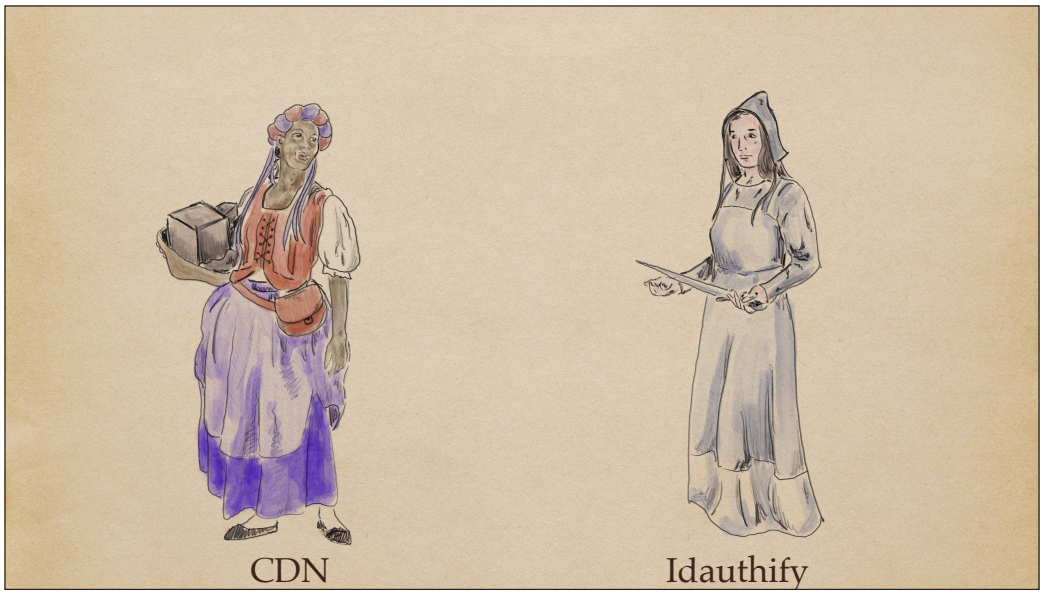


Shishigami arises again, and calls upon Pacote for you to go get the corgi for your scoped package

You and Pacote head off to CDN.

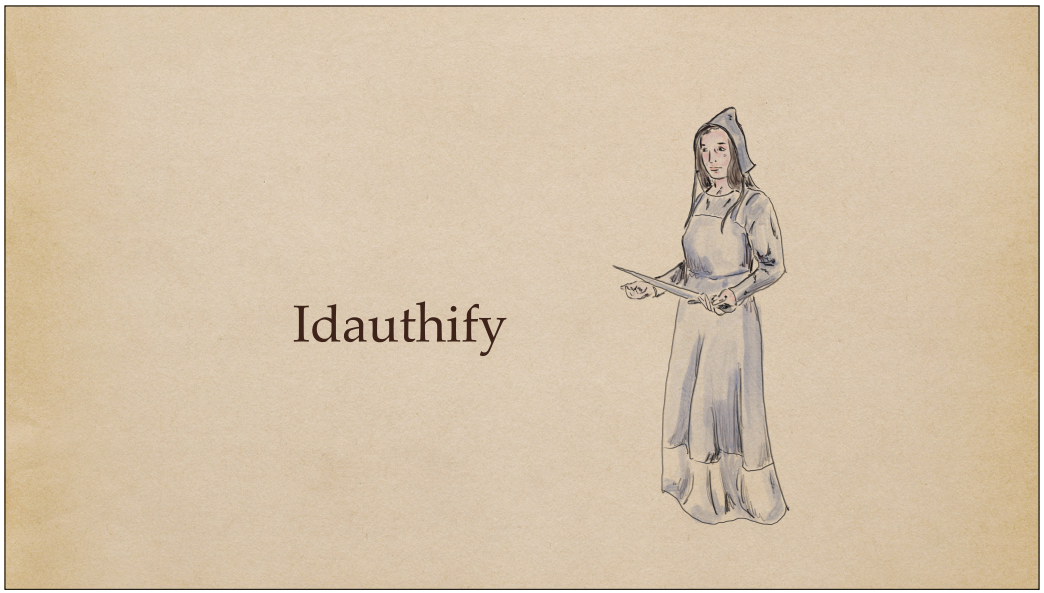


CDN asks what you are here for and notices that your corgi doc request is for a scoped package



Before you can say anything else, she opens the door and sends you along to the next step on our quest, Idauthify, master of all things Auth

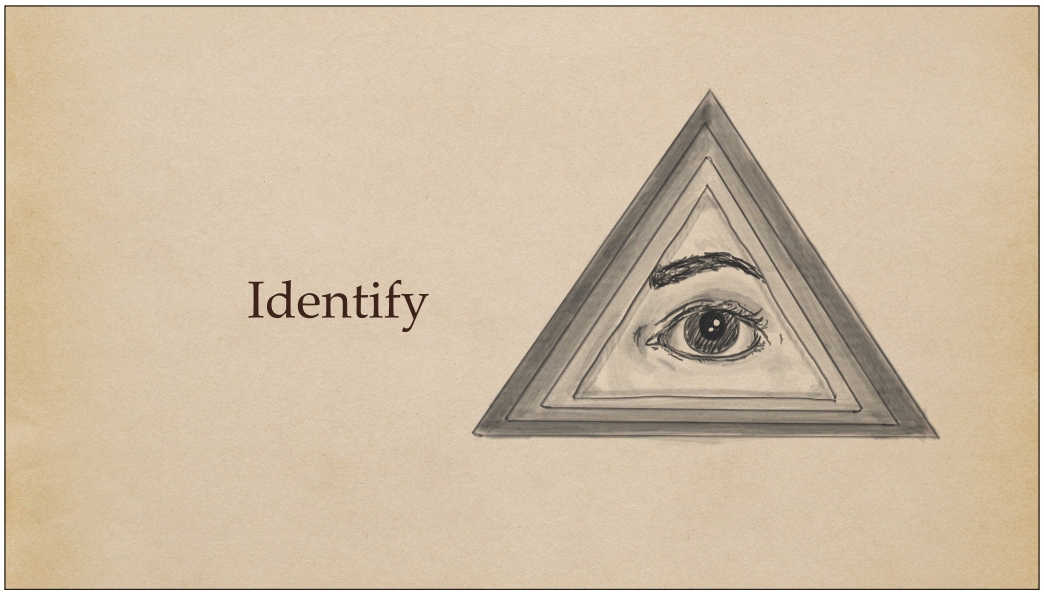
Book Four
Chapter Two



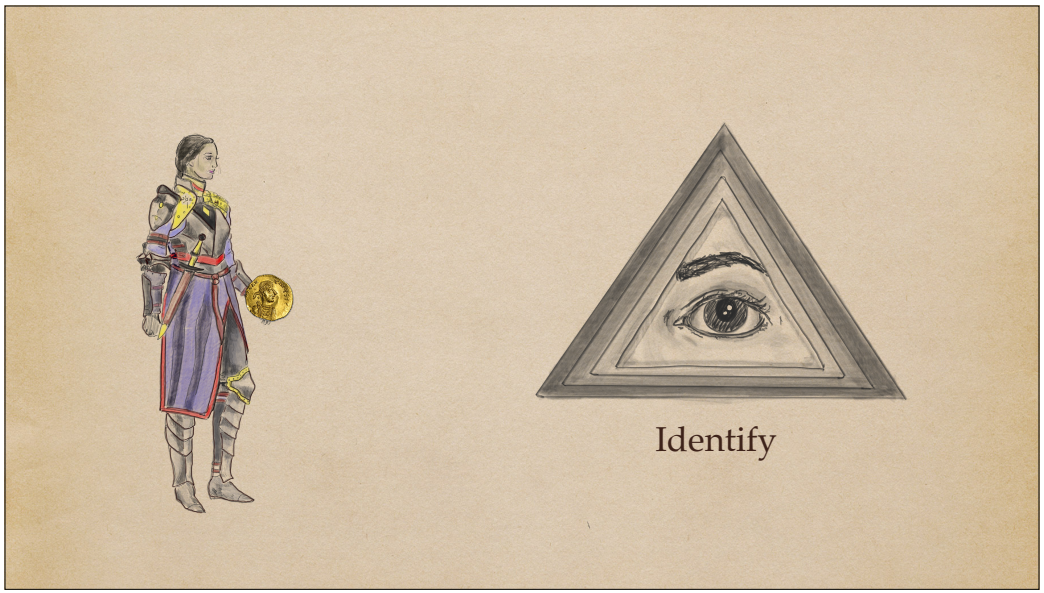
Idauthify, the master of Auth and protector of the cave, carries the keys to your journey to package server. She will make sure that you are who you say you are and that you are capable of viewing that package, then she will tell CDN if you are worthy



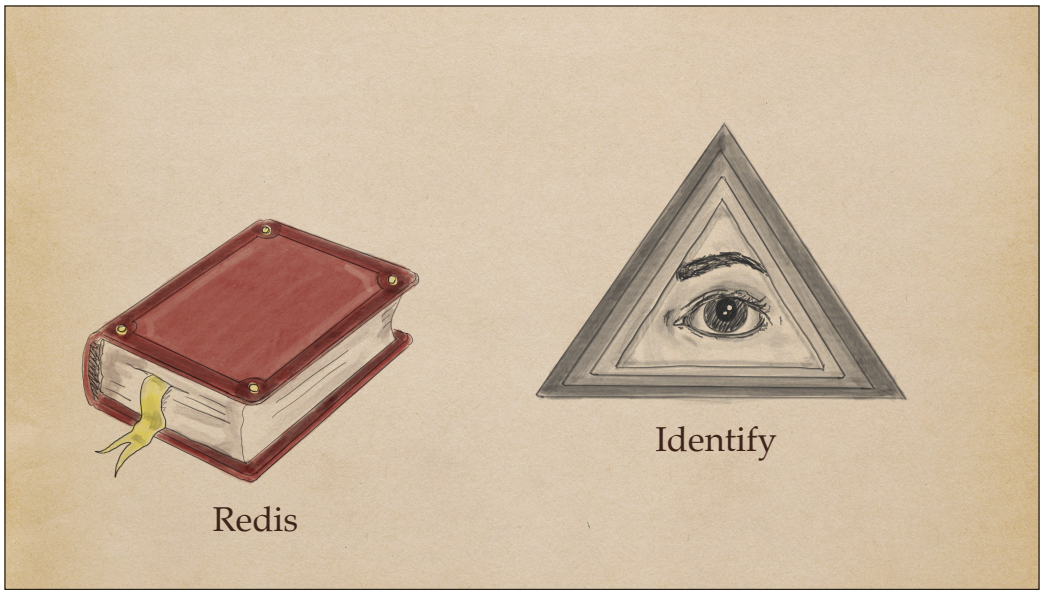
She asks you for your credentials, mainly the authentication headers that CDN makes sure you have and sends you to the Eye of Auth’n



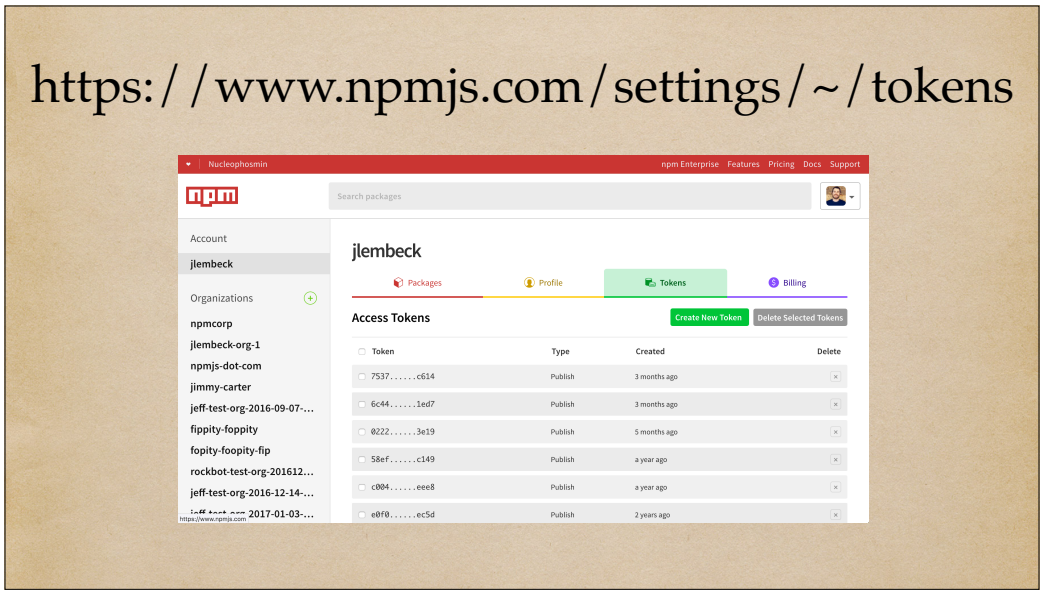
The Eye’s true name is Identify and you come to it from Idauthify. It wants the exact request you came with and it examines your headers. In particular, it cares for your `authorization`, `cookie`, and `CDN-client-ip` headers.



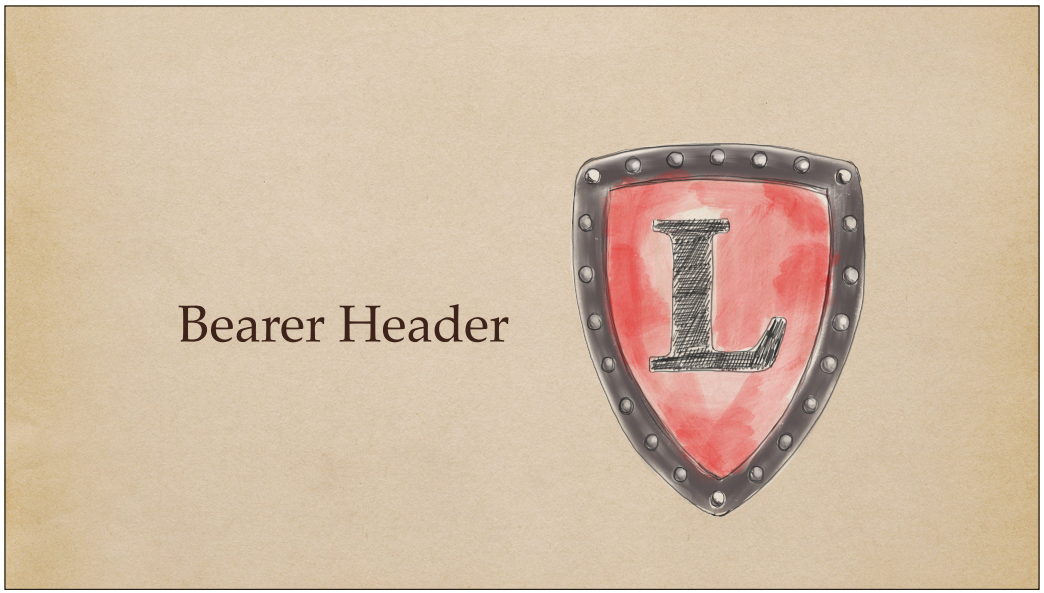
Since we were unable to bring a session cookie with which to feed the Eye, it will have to look into our authorization header. [click] An authorization header includes the scheme and the credentials being used. For the Eye, the scheme is ‘bearer’ and the credentials are your authentication token bestowed upon you when you logged in. If your credentials match the records that Identify keeps, it gift you with the Sigil of your House. You will no longer be questioned and you will be sent along your way. Now. How does it check?



Remember logging in? When your token was created, it was stored in a redis instance that is now being referenced by Identify. As long as that token remains in redis, it is valid to reference and can be used to get through Identify.

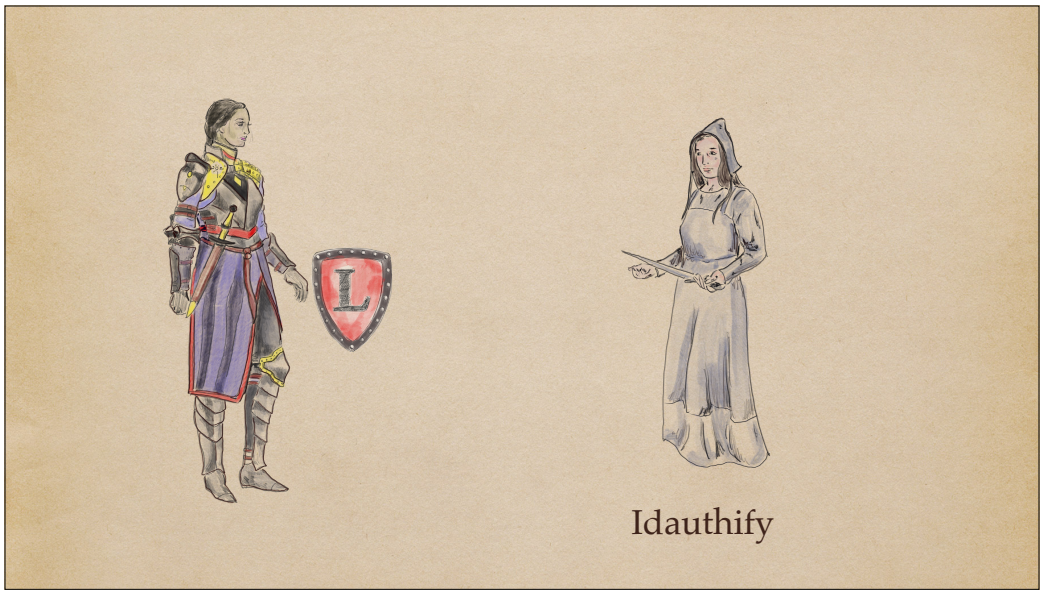


The tokens that you've used are available here: https://www.npmjs.com/settings/~/tokens (we will never display the entire token).

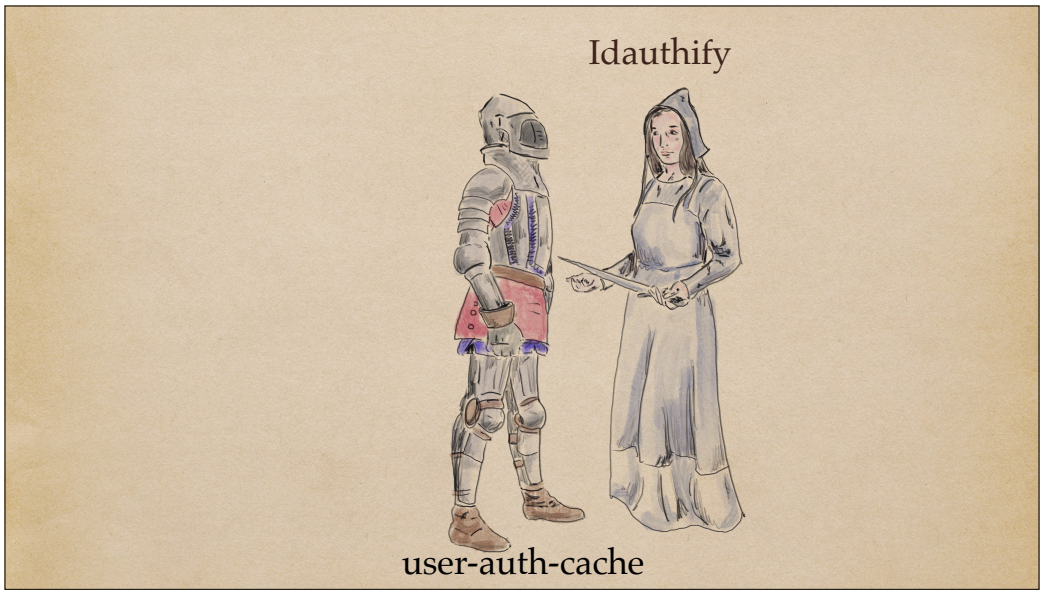


Now that your token has been retrieved, it is set as the value on the response back to Idauthify. A bearer header. That header is your sigil. You have successfully been authenticated. Congratulations.

YOU FOUND HOUSE SIGIL

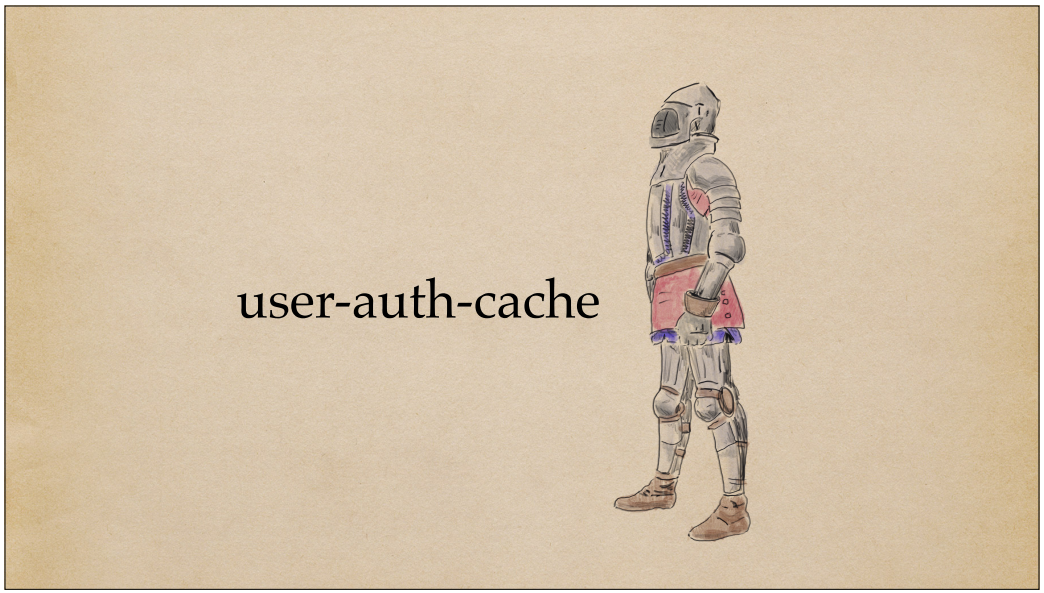


Now that you have proven that you are who you say you are, you show your Bearer header to Idauthify, who then will find out if you are capable of accessing the package that you'd like to install.

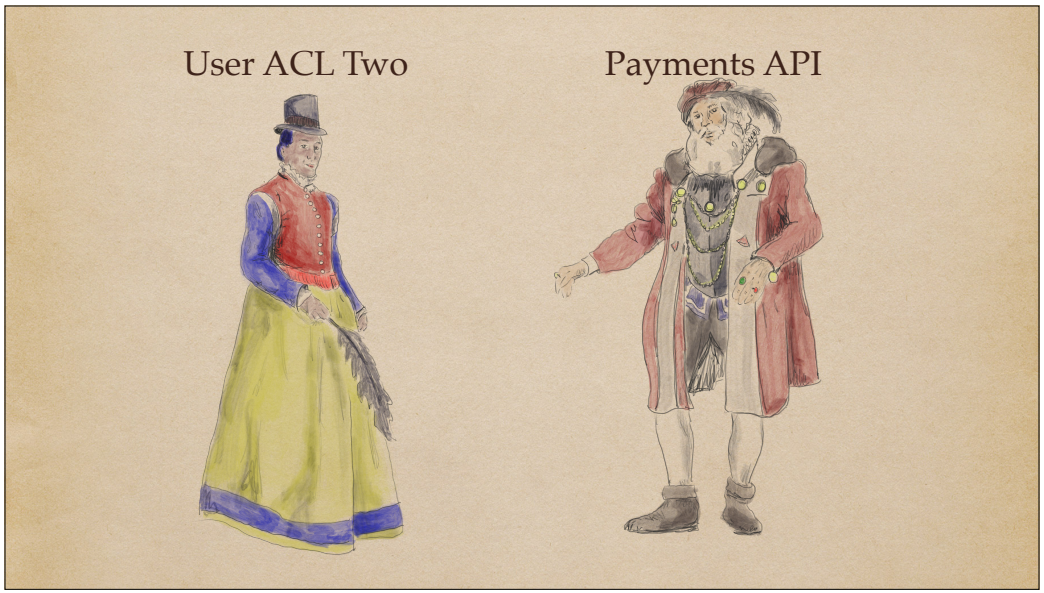


In order to get this information, Idauthify checks with the Keeper of User Authentication Data, user-auth-cache

Book Four
Chapter Three



user-auth-cache serves a very specific purpose: Does this user have access to this package? If so, is it read or write access? If it does not know, it reaches out to synecdoche-api to find out, which passes through to user-acl-two.



As mentioned earlier, user-acl-two is the true master of knowledge. It and Payments-API hold the final say on all things. Since any writes in our system go through to user-acl-two and payments-api first, once the package and user relationship is cached, if anything happens that could affect that relationship, uesr-acl-two and payments-api set a value in a queue that is consumed by ACL Invalidator, which proceeds to replace the data in user-auth-cache and simultaneously invalidate any caching of the document in CDN.



Today, the sun shines upon you. Nothing has been invalidated in the wrong way. Idauthify returns with news from the Keeper, you are allowed, proceed back to CDN with your head held high. Had user-auth-cache returned with bad news for Idauthify, you would have received mark of FORBIDDEN.

Book Four
Chapter Four



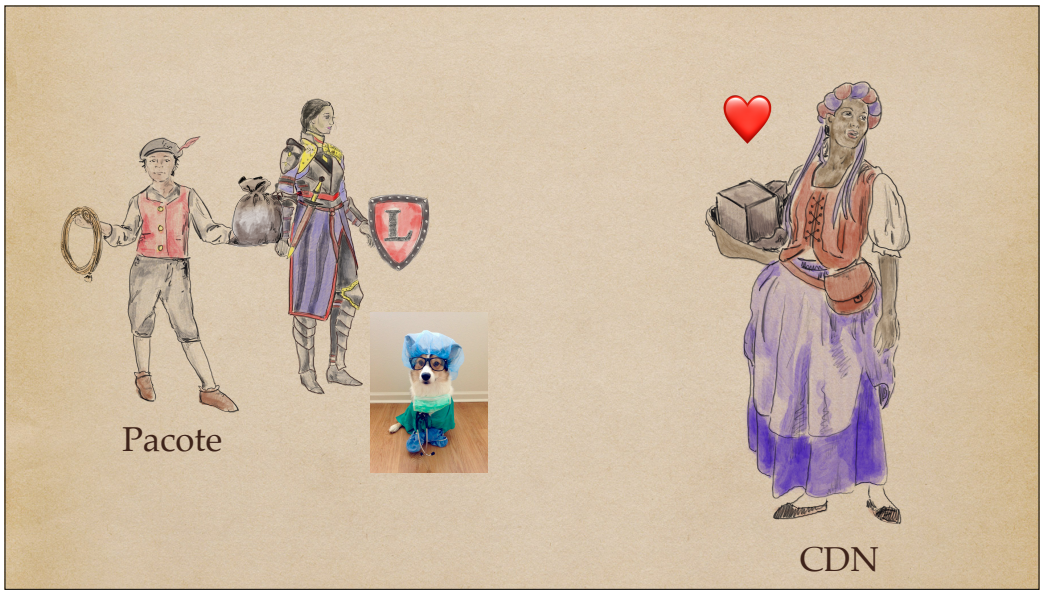
Now that you have permission from Idauthify, you return to CDN with new credentials. She checks her cache to see if there is an entry with these credentials and this package. It is not there. She opens the door to the path of the Cave of Wonders.



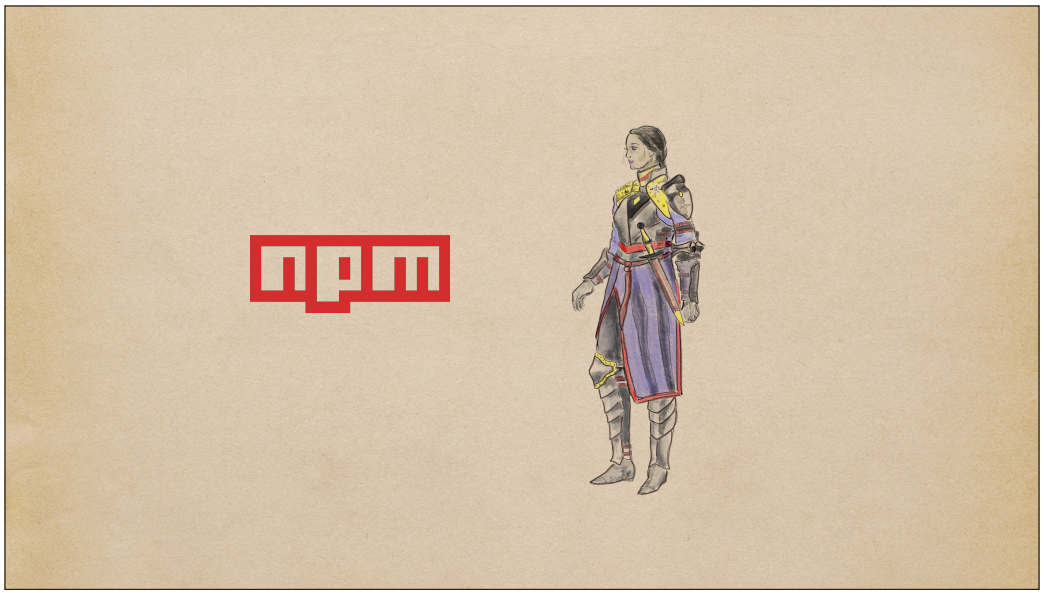
You go get that little fuzzy doctor, I mean, corgi doc



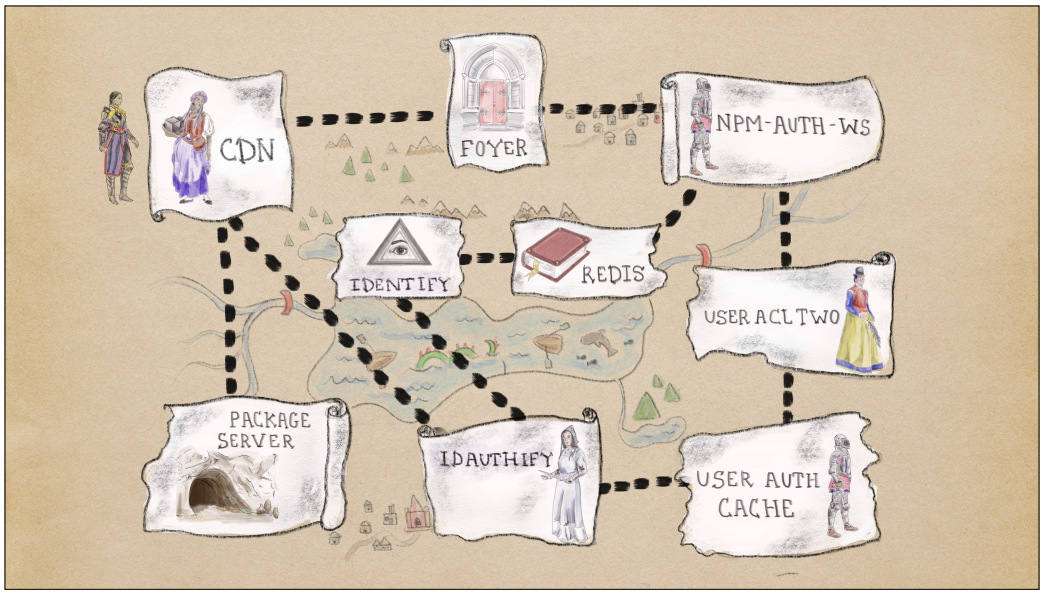
you turn around and head back through CDN.



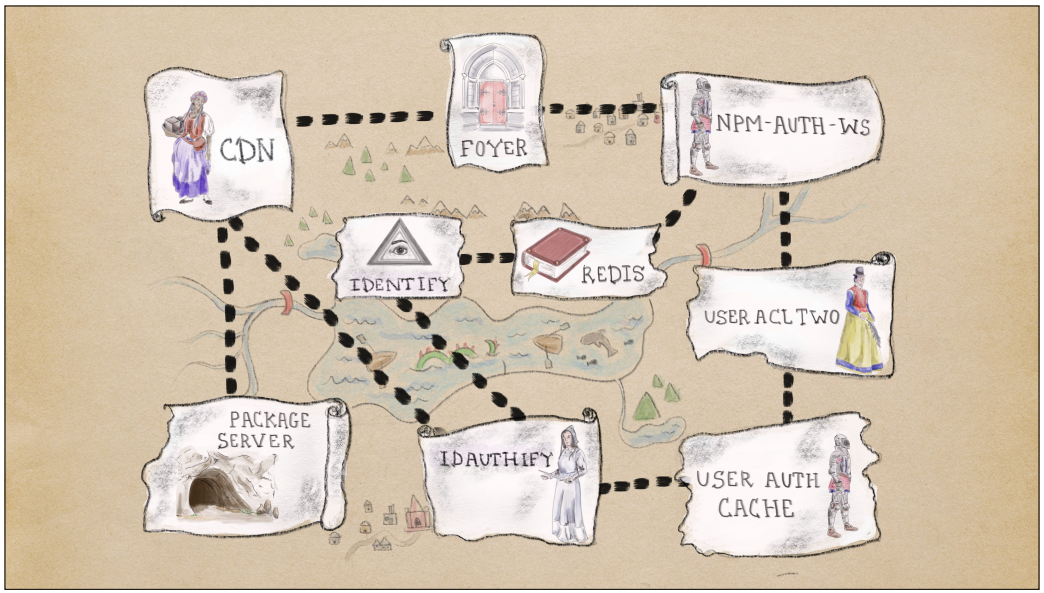
She caches the package in such a way that only you can retrieve it. You go back to Terminal Pointe



You’ve got the hang of this. You have the metadata, you have your tool of valor, you know the path.



Join Pacote and follow the same route to get your packages.



With your bravery, hero, you've once again opened the doors to the promise land of the npm registry. Now, the 10 million voices that call out for hope once again can appease their gods and their masters. That is, until they overthrow both.

The End.

illustrations by Racheal Lembeck (@rachlembeck)