**Philosophy:**

* No magic, no generators- explicit is better than implicit.
* Encapsulation – all apps should depend as little as possible on each other (but can and should depend on the core)
* Modular – things should allow integration in all points.

**The foundation:**

* Component resolution
  + Server code
* App ecosystem
* Configuration
* Deployment
* Service locator

**The core:**

* Persistence
* Logging
* API-rest/routing
* User/Account model
* Authentication/Authorization
* Testing framework
* Management commands
* Admin interface (responsive)
* Thread local storage?
* Security
* Static file management
  + Allow overriding static
  + Enable file processing
  + Derive angular dependencies

**Peripherals:**

* Emails (templating engine)
* Feeds
* Sitemap

**Technologies used:**

* Node.js
  + Express.js
    - Baucis.js
  + Passport
    - Passport-rememberme
  + Lograp
    - Winston
  + mocha
* Mongodb
  + Mongoose.js
* Angular.js
  + ui-router
  + karma
  + restangular

**Things to take care when creating an app:**

* only require moonshine with peerdependencies.
* Dependencies are defined in package.json – moonApps

**Terminology:**

* Moonshine-Core: the core features of the moonshine ecosystem – dependency resolution, configuration loading, service locator and extension/hook support.
* Component: a reusable package that extends or builds upon moonshine's core features- any functionality provided in moonshine that isn't in core is provided via a component. All components expose the component interface by exporting the moonshine.defineComponent function.
* Extension: a module that is automatically loaded by moonshine-core on setup.
* Load unit: code on each componenet that are loaded by unit loading extensions.
* Hooks: Components can provide hooks for other components to expand upon the components functionality when certain actions occur.

**Notes:**

* Why do we need a service locator, why not use require directly?
  + Since node resolve dependencies locally first, all modules must remember to put their shared dependencies in peerDependencies or risk using different copies of the same dependency.
  + Using a service locator, allows moonshine to enable more varied ways to initialize and setup dependencies (a single module can register several services for instance).
  + It enables overriding services, creating proxies over services, etc…
  + It allows lazy loading of services (with require, you'll have to require the module just before you use it).
* Why not use dependency injection:
  + All the current ways of dependency injection in node.js appear to be performance intensive.
  + Moonshine exposes native functionality – such as express routing, which means that it does not control all entry points into a process – so dependency injection would require users to manually call the moonshine injector in such cases (which is mostly the same as a service locator).  
    The other option is to provide our own implementation of every feature- which is against the policy of using the best in class services.