## Philosophy

The philosophy of the MashupJS is leveraging past experience to reduce technical debt moving forward.

## Technical Debt

Technical Debt according to Wikipedia.org

<http://en.wikipedia.org/wiki/Technical_debt>

“Technical debt (also known as design debt or code debt) is a recent metaphor referring to the eventual consequences of poor system design, software architecture or software development within a codebase. The debt can be thought of as work that needs to be done before a particular job can be considered complete or proper. If the debt is not repaid, then it will keep on accumulating interest, making it hard to implement changes later on. Unaddressed technical debt increases software entropy.”

"As an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain or reduce it."

— Meir Manny Lehman, 1980

## Examples of Technical Debt Reduction

According to the Wikipedia link above there are 10 causes of technical debt. Below are a list of causes, according to Wikipedia, followed by MashupJS philosophies which address the cause.

* ***“Lack of building loosely coupled components****, where functions are not* [*modular*](http://en.wikipedia.org/wiki/Modular_programming)*, the software is not flexible enough to adapt to changes in business needs.”*
* ***“Lack of test suite****, which encourages quick and risky* [*band-aids*](http://en.wikipedia.org/wiki/Band-aid_%28Computing%29) *to fix bugs.”*
* ***“Lack of documentation****, where code is created without necessary supporting documentation. That work to create the supporting documentation represents a debt that must be paid.”*
* ***“Lack of collaboration****, where knowledge isn't shared around the organization and business efficiency suffers, or junior developers are not properly mentored”*
* ***“Parallel development*** *at the same time on two or more branches can cause the buildup of technical debt because of the work that will eventually be required to merge the changes into a single source base. The more changes that are done in isolation, the more debt that is piled up.”*
* ***“Delayed refactoring****– As the requirements for a project evolve, it may become clear that parts of the code have become unwieldy and must be refactored in order to support future requirements. The longer that refactoring is delayed, and the more code is written to use the current form, the more debt that piles up that must be paid at the time the refactoring is finally done.”*
* ***“Lack of alignment to standards****, where industry standard features, frameworks, technologies are ignored. Eventually, integration with standards will come, doing sooner will cost less (similar to 'delayed refactoring').”*
* ***“Lack of knowledge****, when the developer simply doesn't know how to write elegant code.”*

Technical debt

## Abandoned code

Code that is abandoned because it works. Over time it’s forgotten and unattended to. The day comes when the business need new features or OS changes and breaks it.

**Example**

* A nightly batch process/program that has run well for many years. It copies critical application data other applications require. Over time, nobody in the IT organization knows about this critical process.

**Cost**

* Uncertainty – Unknown dependencies. Example: Your customer/product tables might be updated by a nightly process that is long forgotten. Estimates and support roles forget to take this into account because it’s always worked.
* Re-write – Once discovered, support for abandoned code often requires a complete re-write before new features can be added or the application can become considered supported.

**Solution**

* The Mashup attempts to combine applications using a single core. Even in cases where an application requires little attention its core components are kept up to date. The application remains updated, visible, and not forgotten.

# Avoided code

## Priority

Our primary concern is performance. All user requirements must be met and if extra time is required to address a performance issue then it must be addressed.

## Requirements

* Any application built in this model will also run as a local application on any device.
* No errors are ignored. It’s easy to allow JS or other errors to slide because they are difficult to locate and seem to have no impact on system operations but later these errors come back to affect performance or create unpredictability.
* All new code is reviewed by another Mashup programmer and best attempt is made to follow the guidelines.
* No inconsistent implementations. If a better approach is found then make a plan to implement it everywhere.

## Benefits

* A single code base can meet all of an applications needs. SPA applications with current framework technology can load from a web page, a directory on the network, a directory on the local machine, or as an application in any app store.
* New to the current release of SharePoint, SPA applications built in Angular can be hosted by and integrated with ShapePoint.
  + For deeper support and language knowledge I’d recommend building SharePoint applications this way.
* We can avoid locking the user base into a device OS because our applications will be device agnostic.
* Deep linking technology, part of the SPA model, allows the application to derive all of its state based on the URL.
  + The state is not stored in the URL but based on the deep linking URL the application knows how to rebuild it’s state.
* Development is faster because of deep linking. You no longer need to navigate to the page you are working on with each build.
* Speed is a characteristic of a well-designed SPA application.
  + All applications will find bottlenecks but SPA applications avoid most of the common performance pitfalls.
* Adopting the most active technologies in the development community means a great deal of support and experience is being created. Several Cincinnati based large corporations are also adopting the SPA model.
* Adopting a single technology stack sets us up to support each other and pull from each other’s experiences.
* We can back each other up more easily.
* Job security. Companies are begging for SPA experience.
* Consistent user experience.
* Build plumbing once.
* No apps left behind.
* Uniform implementation of best practices.
* Leverage open source community to help mature our Mashup.