Making Software Soft Again

What, Why and How to architect the new generation of HTML5 Web apps

White paper by

http://A.lgorithms.com

What?

What's in an architecture? You can hear this question more and more, as sophistication and complexity of computer systems grow. Software architects are realizing that there is something more to developing a system, than writing an ordered bunch of **if, then, else**s and sometimes **for** loops. And they are looking for a structured strategy to organize their world around it.

Technologies come and technologies go, but in many developers' eyes, the new world of HTML5, JavaScript-centric "Single Page Apps" (SPA) is finally the proper way to build Web systems. It changes the way web apps are built: instead of building webpages on the server and sending them to the browser client, it loads *code* into the browser and then puts power in its hands to build the application, load the visualization templates and get the data. It uses the customer's CPU, reduces unnecessary round trips to the server, allows to eliminate statefull servers and makes a full circle again to the client-server paradigm, the buzzword of the early nineties. But unlike those early systems, it now allows to change the application itself on the fly, with no need for installations. And with the rise of Node.JS, it even allows technology unification between client and server, which is a huge step towards simplification, after all these complex years.

We will put this new technological generation in the center of our discussion, but the principles that we will explore can be applied to many systems. We will try to answer the question 'What's in an architecture' in an hierarchical, top-down way. First, we will discuss the 'Why': the high-level goals of a

software architect. Second, the 'how': we will translate them to concrete engineering strategies. Last, we will get down to a list of 13 practical methods to achieve these goals and strategies. We will do all that from the perspective of the ***Software*** Architect. We see many architecture articles directed towards CTOs and other high-rankers, talking about the high-level modules, like load balancers, physical machines and networks. This is not where we are going - yet. We'd like to talk to developers about development.

What's not in this article? So many things. We need to be humble as we cannot possibly cover all bases of such a diverse and wide field. We only talk from our own experience and about the things that matter to us, with many subjects left out. For example, we will not talk about databases and their data: this is a vast topic by itself. We will not go for the lower level stuff, like design patterns and UMLs. There's plenty to explore, but this article slices the field at a certain point – not too low, not too high – and with software development as its main focus.

Why should we give thought to see what's in an architecture? First, because it puts things in order. Many software architects would employ the principles listed here naturally and instinctively, without understanding why they work it the way they do. Giving names to things is one of the first steps towards ordering any field of thought, as it organizes one's work process around a limited check list, and maybe even quiet some of the constant dilemmas and trade-off balance acts, that are the daily bread and butter of software architects. Second, for the brilliant developers who always felt as architects – and rightfully so, since every developer is the architect of her own system – we would like to present this list to enhance their systems and grow their craft. The third reason is "because it's there". It's interesting, it's fun and it has no specific reason now – but we know that having this understanding might help us in ways we cannot yet foresee.

But isn't this one of what architects try to do when they design a system -- to answer problems that they cannot yet foresee?

Why?

Indeed, some say that the architect's work is all about ***scaling***. I can agree with that: it definitely is. But, contrary to what some developers think, it is not only about performance scaling. When young developers think about architecture, they think about data throughput, concurrent users, and many other similar metrics. Many times, they would first try to get some metrics there and then offer a design. Tell me how many concurrent users you have and I'll build the system for you.

But actually you should look at two aspects of scalability. Yes, you need to squeeze as much throughput *today* from the system that you build. But a good architecture is about the *future*: how do you add more features, support more load, increase your team size – all with minimal amount of *man hours* involved. In terms of cost, *buying more computational power is cheap*. Doubling the company's CPU power would

increase the operational cost in, say, 5%. But try to live with a bad architecture: adding 5% more features could double your yearly cost.

So the architect goal is therefore to ease two types of scaling:

- Performance scaling
- Process scaling

Process scaling is about *making software soft again**.*** Life is dynamic, and systems change: whether because our startup has so much success that its user base grows exponentially; or because it failed so miserably that we need to change its target market completely. Or because management suddenly asks for new added functionality without consulting us dev people – all of these are changes. And unlike popular belief, changes happen. This is why they say that a good design is a ***design for a change***. It is up to you, the architect, to make sure that your system will withstand those changes with the least amount of man hours possible. Like minimally invasive surgeries, you want your system to have the shortest healing time possible. Get in, get out.

That's why I love the great book called ***Peopleware***. I didn't know too much about its contents, but just admired the title, because I felt that building software is not about the algorithms that will process the data. It's about the people that build and rebuild these systems. The book, by the way, is truly mind-opening – it talks about systems from more than 40 years ago and it is still so, so relevant today, giving a 15,000 foot perspective over development processes. It vividly demonstrates how everyone working in this field belongs to a great group of individuals that creates a living and evolving tradition.

How?

So the next question is naturally 'how'. How to achieve this wonderful utopian architecture that is so flexible, so future-facing, so scalable and yet so robust…

Principles

In fact, the principles that you want to bake in your system can be summarized down to a list of four:

- Isolation
- Reduction
- Methodology
- Reliability

Isolation has been called in many names such as modularity, loose coupling, components and more. In essence, you want your system to be divided into compartments in order to limit problems to where they belong. It's one of the longest standing principles in engineering: from fuse boxes that isolate room failure points to processes with separate memory spaces, engineers and architects recognized that its best to divide systems into modules. Whatever happens in a module, stays in that module.

Reduction is the word I used to argue that your framework needs to allow the developers to write less code. Reduction is amazing by the way it allows register only the most needed data to convey the information in a system with minimal size. Much the same way your developers should only write what's really needed. Usually, the way to reach such developers' happiness and less boiler-plating is by providing reusable services like caching, binding and others (see below) which the developers will call and use. This helps in two ways: first, it frees their time for more functional, application-specific programming. Second, by helping them code less, you help them make fewer mistakes. Less code, less bugs.

Methodology is really about leadership. You would like to instill quality processes among your team such that your SDLC processes will maximize quality. For example, mandate unit testing. Or use a peer code review tool that documents results. Utilize a Continuous Integration system to automatically test for all the steps. Divide your team along functional lines, create bug reporting processes – all of these are ways to provide better ***process scalability***.

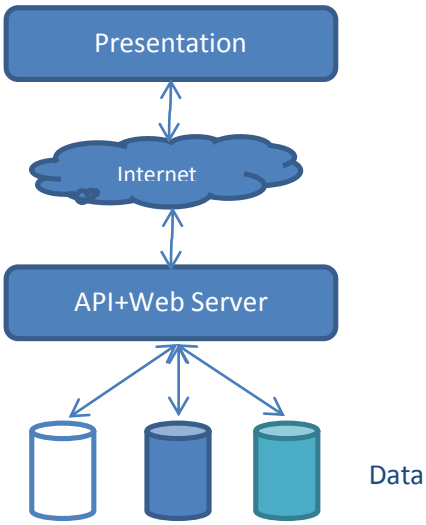
Reliability needs to be a top concern for any system but even more so, once you start to performance-scale it. Adding more hardware and storing more data increase the chances of failures – and failures will happen. There are many techniques that help mitigate that risk: redundancy usually is implemented at the physical level, clustering databases, adding data centers and fail-over machines as a backup for any unforeseen event, external natural hazards and internal unexplained bugs. On the software level, automatic monitoring and process management should be implemented to accommodate for these issues that did leave the machine on, but killed its inhabitant processes.

Practicalities

On the practical level, there are few techniques that evolved over the years, to achieve the high-level principles listed above. First, of course, is the classic web-oriented (but not only) division between 'tiers':

- Presentation tier
- Middle tier
- Data tier

In the past the middle tier was called the 'business tier' but in SPAs this changes, as we see more of the code moving to the client side where the business logic is coded in JavaScript and runs on the browser itself. The middle tier then becomes an 'orchestration tier' that really takes data from different data sources and provides it to the presentation tier through API:



In a Single Page web App the orchestration layer would also provide the 'static files' needed to run the application: JavaScript, HTML, CSS and image files. This is the only thing in which it might be similar to the classic Web Server.

This separation seems pretty obvious but it's important. Let's dive in a bit and look at the techniques inside each tier with more granularities. Although they depend on the tier – backend systems need usually a different set of tools than front end ones – there's a lot of similarities. On any case, even if you don't use some of them, as an architect you must be aware of all of them.

- MVC: Model-View-Controller
- Data binding
- Hierarchical Eventing
- Caching
- Dependency Injection
- CI and Unit testing
- Database adapters
- Versioning
- Routing
- Team partitioning
- Asynchronicity
- Monitoring
- Logging

MVC: This modularization technique has proved itself for many front-end systems and presentation tiers. It separates the data (model) from the presentation layer (view). Interestingly enough, it is a special case of the global data-business-presentation tier separation we discussed above. On the browser-side code, you would build a model, a view and a controller for each functional aspect, like account, product, cart, etc. There are several prominent JavaScript frameworks that provide this scaffolding, with the three major ones seem to be Backbone.JS by Jeremy Ashkenaz, Knockout.JS supported by Microsoft, and my personal choice, Angular.JS from Google.

In such a framework, a '*view*' is defined as a 'template', which is an HTML snippet ('partial') that has special 'blanks' to fill in with specific '*model'* members. This model-blank relationship is called 'binding', discussed in the next section. A *controller* Javascript component would call an API from the API layer to bring in data from the server, massage it and store it in the *model*. The framework data binding would then update the template view to reflect the changes in the model. This could also run the other way: changes in the view (for example, entering data into input fields) would be bound to the model and eventually

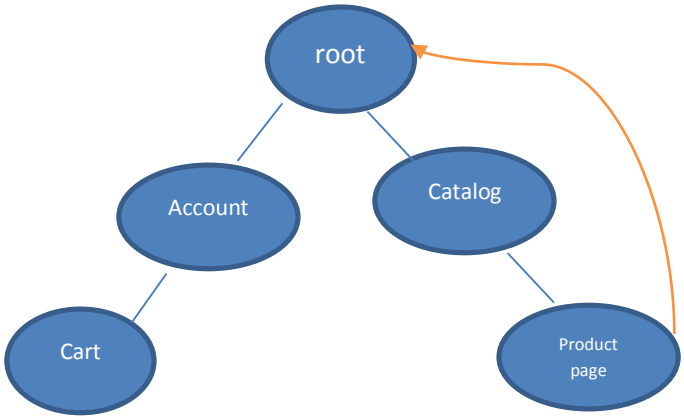
A good MVC framework provides also related services, with the first one being such as RESTfull API calling library which works out the details of using XHR directly in native Javascript. It usually also augments JavaScript with more functionalities, such as foreach iterators, events (see below), object extensions and the like.

Data binding: as explained, allows this goes together with MVC to allow easy binding between model and view. This is called reactive programming: like dependent cells in an Excel spreadsheet, where a change in one cell is changing other cells automatically. Here, a Model field change is automatically reflected in its Views, and vice versa. This is an essential part of the MVC platforms, but it's important enough so we list it here independently

Hierarchical Eventing: is a great way to loosely couple modules. If you have a Cart module that you'd like to call when 'Add to cart' button was clicked inside a product page (which belongs to, say the Product module), don't you never call it directly. Use a ***publisher-subscriber*** pattern to raise an event, to which other modules can subscribe and act upon. The benefit of this lies, as usual, in the future: imagine that a new module needs to get word of the 'add to cart' click – for example, to sum the users monetary activity. With eventing, the new module would just subscribe to the same event. Without eventing, the developer would need to add a call to new module directly from the button click handler, doubling the coding efforts and doubling the chances for bugs.

The best publisher/subscriber design, by the way, is hierarchical and is combined with a modularization scheme across the architecture. Each module has a reference to its parent and the root, and can create its children. To publish an event from Product page module, for example, you would call the global root, which would propagate the event down the tree, ready for everyone to subscribe if needed.

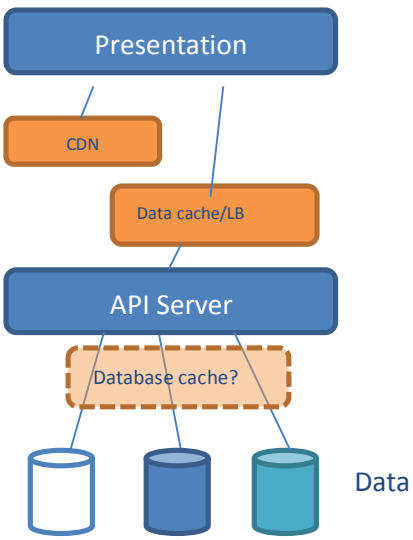
In this way you can define modules that could contain other modules, and events would be published down a tree:



Hierarchy is very natural when it comes to the client, as HTML in its nature is hierarchical and therefore promotes hierarchy in the modules. But remember hierarchy is a good way to organize everything – not only visual modules, not only client modules, but the server logic as well.

Caching: Caching relies on the biblical law "What has been will be again, what has been done will be done again; there is nothing new under the sun". If the user fetches a certain data item, chances are that item would be needed soon again. It proves to be one of the most successful performance enhancing mechanisms, and every system should have one. You need to create a central re-usable mechanism, which they can easily use. Of course, this comes into play in all layers! The presentation layer will use a memory or browser-based caching, the orchestration tier might use a caching server and the database would have its own caching mechanisms – but everyone needs caching.

Server-side caching can be placed on several positions along the request route. For static files we usually would use a CDN like Akamai or others. This is good for files that don't change often, and that do not have to have only one version. We will talk more about it on the versioning section, but since you could have several versions of your app live at the same time, the slow update rate of CDNs is ok to have your *code* served from there.



Your *data*, on the other hand, is a different story. Since you want your data to be consistent (or at least parts of your data like, say, prices), you need a consistent caching mechanism that could be invalidated at will. Usually this could be served by putting the cache on one of two different positions: either before your API server (your middle tier) or after. Remember that data is served in a text format, like JSON (in SPA) or XML, so it's easy to put it everywhere. If put before, memory-based Varnish or even file based Nginx could be used. If put after, between the middle tier and the database, usually memcachd is the regular choice. We found that putting it before the middle tier is very convenient in SPAs since you could use Varnish for application load balancing too, directing load to different API server based on different rules and request contents.

To tie in the full circle, we need to answer the question of how does your application knows what to request from the CDN cache and what to direct to your own servers. This could be done by having the CDN act as your global load balancer, having your requests always hitting it first.

Dependency Injection: in summary, DI is like your centralized object factory. Some call this factory IOC (Inversion of Control) which you call in order to get any object you need. For example, instead of writing

Product * product = new Product()

You would go

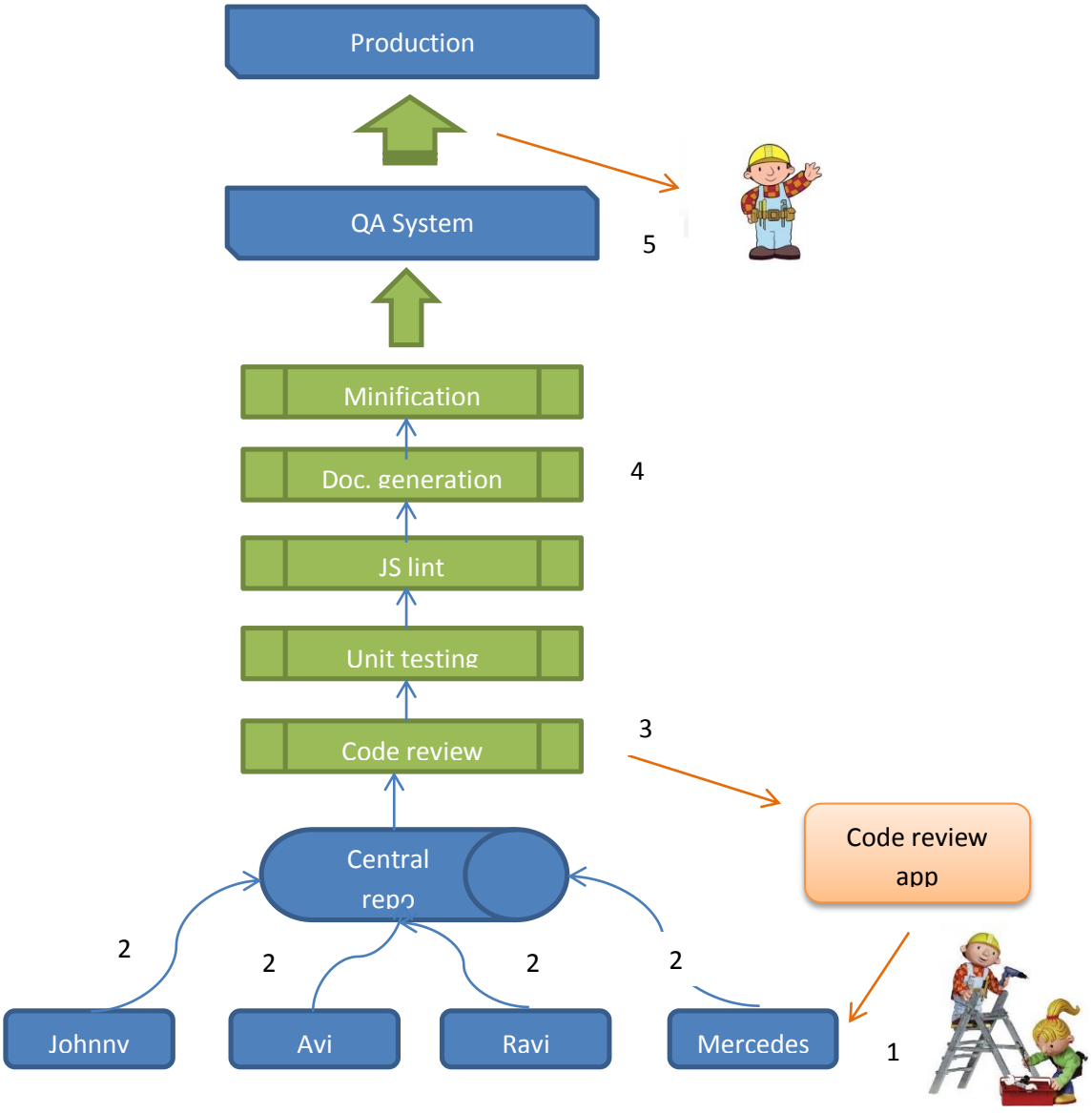
Product * product = IOC(Product) or other syntax, depending of your language (on the presentation tier it would be JavaScript, of course)

The benefit of that is that the dependencies between modules become abstracted. Changing the Product implementation is easier, as long as the new implementation adheres to the same contract.

Some frameworks provide automation for DI. They look at the code and call the IOC for you. One of the advantages of AngularJS is that it provides automated DI out of the box: you can create an injectable object and pass it along to injected-able functions.

Dependency Injection is very important in another field: creating mockups and stubs for Unit Testing.

CI and Unit Testing: Gone are the days where the software architect takes care only of code. Continuous Integration, and the software architecture that achieves it, is one of the most important methodologies in achieving process scaling. CI takes the code and passes it through a workflow of several steps – code review, code cleaning, automated documentation, compilation (called minification on the client side) and most importantly, the automated testing.



An example of a CI processes. 4 developers use their local Git repositories. A peer reviews code (1) using a tool (ie Crucible). Developers push (2) CI takes over, verify that code was reviewed (3) and continues with all other steps (4). Once ready functional testing is done at QA site (5). In this process a human approves push to production

Testing could be divided into two main categories. One is the good old Functional Testing, in which testers (ideally automated but many humans, too) will look at the whole system as one and would work it out as a real user. The second one is Unit Testing, in which the system is broken off to little modules, independent of each other, and tested in isolation. Unit tests are written by the developers themselves, and to allow unit independence we need the Dependency Injection. How can you test the Cart module,

for example, if it is dependent on the Product (for example, by calling Product.Name property)? Only if your IOC could give the unit test a stub in which you will provide a pre-defined details (say, 'test product' for the product name) that the test will expect. The test can then check if other aspects of the module do what they are supposed to do – for example, check if the price is doubled when the quantity is two. The tests are really simple and stupid – but if you have a lot, chances are that if something is wrong, one of the tests will break and you can catch it early on.

Having the developers develop unit tests does mean more coding time. So you'll need to calculate it into your estimates. But in the long run, it means less coding time – when you include iterative bug fixes in this time. I can't forget how one of my team leaders once expressed his wish to have a dedicated tester, so he can focus on coding… oh, the good life! Well, this is as close as it gets.

Database Adapters: Talking about a big topic. Single page apps don't differ from 'classic' web apps: the client stands on one end, and database stands on the other. The difference is in where the 'state' lies, where is the persistent location of the data. It's interesting to see that the middle tier is stateless in essence. In classic web apps middle tiers would hold sessions, where the server would need to dedicate a memory area for every connected user. How non-scalable is that! Luckily, in the new world the browser can hold state pretty comfortably, either in memory, cookies, or 'local storage', the browser's local database. So state resides on both ends. We need therefore to build data adapters, so that it would isolate the code from the database technology.

On the browser side, there's not a lot to do. The local storage is standardized by the HTML5 spec, and on any case it is just saving the JSON as-is. On the database side, things vary much more. But the main thing to remember, is that once you build into your architecture a layer of data adapters – usually on the middle tier itself – you are half way there, whatever method you choose to implement your adapters. The other half is determined by the database you choose. In case you chose or have to work with a SQL database, you should use some ORM. There are several nice choices, depending on your other technology selection. For example, Java would come with Hibernate, .NET will use Entity, but the essence is that they all help you create database access code automatically, based on the tables structure. This will adhere to the DRY principle as you would limit your changes to one place - the database itself. Some of the new NOSQL databases are pretty object-based, and have a non-rigid update mechanism, so you can send you objects as-is for update. This is the case in MongoDB, and if your server-side language is JavaScript, then definitely you don't need any ORM.

The other important note is to limit the scope of data access code to only CRUD operations. I've seen entire subsystems written in SQL, which usually means that this code skipped the regular battery of quality procedures like code reviews and unit testing. This is easier said than done, and you would need to a solution for the more complex queries. For example, a complex join could be done through a view table. There are other solutions, like native extensions or LINQ, but it is always important to remember: you don't want to put code in stored procedures.

Versioning: I've put it here since this is an area that is often overlooked until it's late in the game and it becomes pretty pricy to implement. The question is how to publish a new version of your application and still have the old version active, for users that are currently connected. The usual solution is to use the load balancer, directing new traffic to new servers. But this means that you need to have enough new servers, and many times you need to request that from another team, which is even worse...

You need to come up with a Versioning solution. The is huge risk of you deploy it module by module, since a module might have dependencies on other modules and these might go to the old version. So better do it as a whole. Luckily, in this day and age of heavy RESTfull URIs, you can put the version number in the beginning of it (kind of a base ref) and have all the rest of the path the same way. For example, have your JavaScript downloaded from /v12/js, and when it's time to move to a new version, change it to /v13/js .

Versioning on the database is far more complicated since the database holds only one version of the data, at least for some data. This means we cannot have two different databases live be at the same time. That's why data structure changes are so less frequent, since people understand the risk. Building a new version here should be preceded by a broad analysis of the changes impact, and followed by careful reviews and tests, before releases.

Routing: routing is needed on both the client and the server. Some mechanism is needed in order to look on a 'request' and have the right components executed. On the client, the 'request' is really the URL line, which could change because a user clicked on a link or because she typed a new URL. You can build routing as part of your components – for example, subscribing to URL changes events, analyzing it and performing correct action – or, by building a central component: t***he router***. Such a component is configured with the path and the corresponding module to handle it, and once the URL and its parameters change, it will activate the right module with the new state. Now, the question is what are these "modules" that the client is made of? On a browser the components would usually include the model-view-controller components, or *widgets*. HTML facilitates hierarchy and nesting, so a widget can include other widgets, and your router should take it into consideration. The UI-router which I used in our AngularJS project supports nesting and uses it for its benefits. It allows lazy-loading of the template, entry and exit hooks, animations, layout/widget separation and more. You can read more here on the way I've designed and used it – it is actually the core component of the client architecture.

On the server, the router's job is to look on the coming request and route it to appropriate handler. Since we will probably use RESTFull requests, the router could look on the path and use it to send the call down the hierarchy to the right component – somewhat similar to the client. As already mentioned, a nice hierarchical approach could be beneficial in the server world too – so your router should take it into consideration. If the server language is JavaScript, maybe a unified abstracted router is in place, for both client and server? Wow, definitely something we haven't yet seen around the industry.

Team partitioning: How to organize the teams amongst the developers? Some might say that this is not a question for a software architect. But I would beg to differ. A software architect should build her system to scale developmentally, and team partitioning is a part of the ***methodology*** side of it. In some companies, there would be no one else to turn to! And as the leader in design, the architect should push and promote a correct organization for the teams.

One natural way to partition by subject. Some developers might be responsible for area 1 (ie product API), some for area 2 (feeds). That goes without saying and is usually a natural way of creating functional expertise. But a huge benefit would also come for splitting the team horizontally to two: infrastructure, or 'core'guys and functional, application developers. The infrastructure team would create the engine, the framework and would continue to improve it endlessly, offering better ***reduction*** and better performance. They would pick reused functionalities and developers boilerplate methodologies and generalize services and automations that would make the work more productive and the results with better performance. The teams should separate their layers in an Open/Close fashion: open for discussion and communication, but maximally closed to penetration and special-case patching.

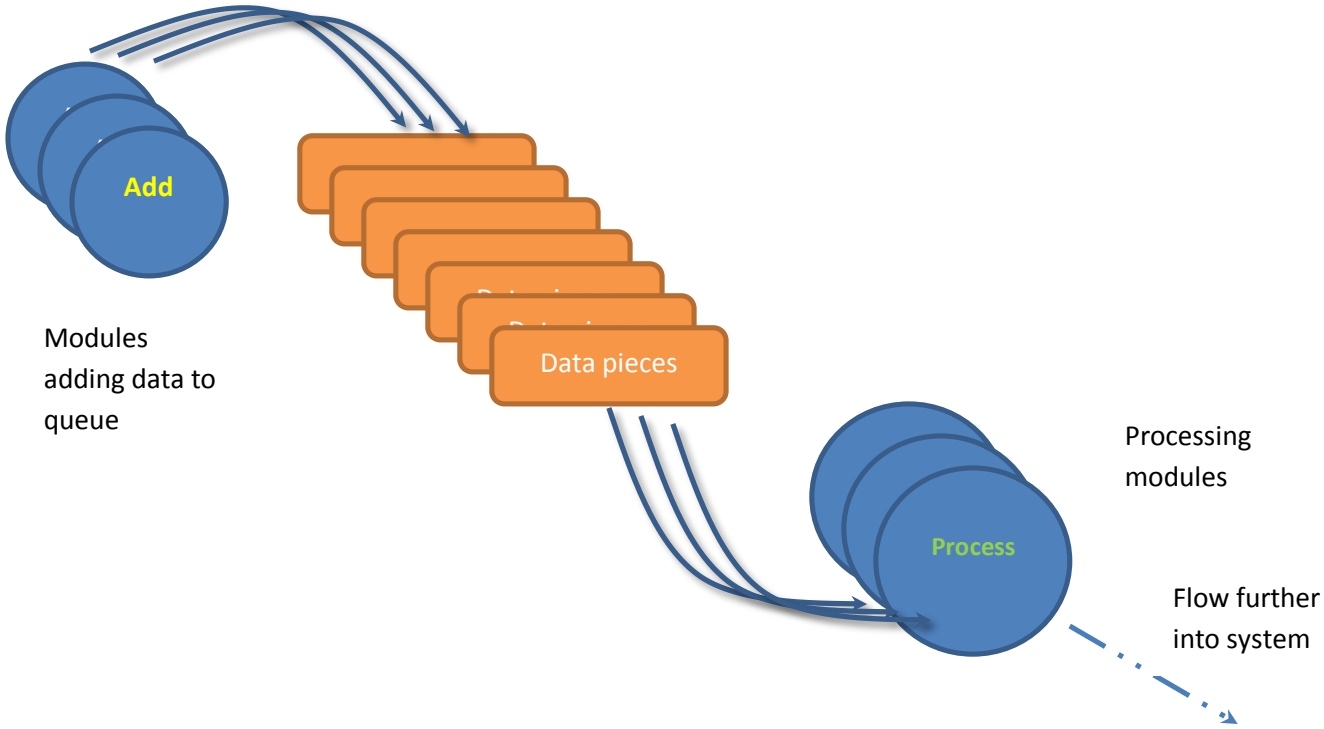
Asynchronicity: This is a principle from the performance section, and in one sentence, it means that your system should have a way to schedule actions to happen later. It might sounds counter intuitive: how on earth would scheduling for later improve performance? Definitely not a good moral lesson, but in fact, it helps performance if you evaluate the system performance over time and as a whole. It is simply because big systems are comprised of many smaller systems that have different bottlenecks and different performance profiles. When you schedule tasks to happen later, you free up system resources and increase the chances that these resources will be used for other bottlenecking executions.

Asynchronous execution can help in all levels. JavaScript as a language is so successful because of the built-in convenient mechanism that allows it. The ability to create anonymous functions allows to easily use callbacks that will be executed later with the results, instead of sitting and waiting now. The closure mechanism saves the context. Yes, you could do all that with other languages too, but you'll need to write ad-hoc handlers and context variables for each call. See how easy it is to read from a database, for example, in Node.JS (please forgive the use of plain old SQL):

```
handleRequest(req,resp) {

   dbDriver.read("select * from users where
userid="+req.id,function(data) {

      resp.write("username is  " + data.name);

      resp.end();

}
```

The request *req* is received from the browser, sent to the database, and Node is ready for another request. When it returns, JavaScript still holds the specific instance of the response object (*resp*) ready for the specific data. How cool is that? Imagine what the same approach would take in Java or C++.

Another mechanism to achieve asynchronous performance gains is with queues. For example, data feeds importing stuff into your system should not be processes in a batch or in a one loop. Instead, some task should add data pieces on the queue, and another task should take it from the queue and process it, feeding the data further into deeper parts of the system. Not always these tasks reside on the same machine – and therefore, those queue could be implemented in volatile and non-volatile memory alike (i.e. a file system). Not only it frees the CPU on the first task side to continue doing other chores, but it also allows your system to scale easily, simply by adding more handlers on either side, each just working on the queue.



Monitoring: Monitoring, or in its less boring name, the watchdog, is the only bullet that relates to 'reliability'. Building a watchdog system means that you need to build a module that would watch other modules and take some actions if they crash. For example, it can look at you server process. If it crashes, it would start another process instead and maybe send you a text message, alerting that something bad just happened. You could also generalize it to the machine level, or reduce it to specific health checks. Either way, it's a back office backup.

In many sections of this article we talked about ways to minimize bugs. But what if all else failed, and a bug happened on production? What if the server crashed because of a hardware failure? Sometimes, you need to know immediately and act immediately, without even trying to understand. This is the watchdog.

Logging: logging is last not because of its significance -- this is one of the very first services every architect is putting in place when writing the framework. It is here because we should leave this discussion with it in mind. It exemplifies exactly those unified services that should be similar to all developers, and these days have standardized on the same pattern where 3 levels (INFO, WARNING, ERROR) are logged, several options for targets (file, console, remote client). It also has open-source support in the log4*-derived projects. In fact, it is so important and so unified, that nowadays there are several products that are dedicated to logging and supply it off the cloud, like Loggly, splunkstorm and others. It seems much better to go with them as they provide great UI – usually the Achilles' heel of any back-end utility developed in-house.

Summary

There are so many things we left out, and probably we don't even know half of them! Animation, for example, is nowadays a must for modern front-ends. Object Oriented Programming patterns is missing, and maybe for a reason, as some say that OOP is starting its decline in the ever changing technology life cycle of nature. There are probably many lower-level guidelines that some readers were expecting. We, however, tried to focus on the high-level stuff:

The architect's job is to make the application scalable:

- Performance scalability
- Development scalability

We focus here mainly on achieving the two using the second. Therefore this vision translate into 4 strategic areas of work:

- Isolation
- Reduction
- Methodology
- Reliability

This tactically, means we need to use the following 13 approaches:

- MVC
- Data binding
- Hierarchical Eventing
- Caching
- Dependency Injection

- CI and Unit testing
- Database adapters
- Versioning
- Routing
- Team partitioning
- Asynchronicity
- Monitoring
- Logging

Notice that there is no mention of HTML5. It's not coincidental: this set of methods is good for any client-server model. In that respect, we are noticing that architecture approaches are iterative in nature, too. What was the buzzword of the nineties is coming back to life, but with so much more new ideas and glamour!