Init.js: A Guide to the Why and How of Full-Stack JavaScript

by Alejandro Hernandez - JavaScript Developer @ Toptal
#Node.js #Backbone.js #Marionette.js #Express.js #RequireJS #Grunt.js #Mocha.js #Chai.js

The Story

So, you and your co-founder have this great idea for a business, right?

You’ve been adding features in your mind.

Frequently, you ask potential customers for their opinions, and they all love it.

Ok, so people want it. There's even some money to be made. And the only reason they can't have it is because you haven't implemented it—yet.

So finally, you sit down one day and say, "Let's do it!" Soon, you're trying to figure out how to implement your app's business logic, the killer feature that will drive the product forward: you have an idea of how to do it, and you know you *can* do it.

"Done! It works!" you say. Your proof of concept is a success! All that's left is to package it up into a web app.

"Ok, let's create the site", you say.

And then, you realize the truth: you need to choose a programing language; you need to choose a (modern) platform; you need to choose some (modern) frameworks; you need to configure (and purchase) storage, databases, and hosting providers; you need an admin interface; you need a permissions system; you need a content manager.

You want to be lean, you want to be agile. You want to use technologies that will help you succeed in the short- and long-term. And they're not always easy to pick out.

You have tens upon tens of architectural decisions to make. And you want to make the right ones: you want to use technologies that allow for rapid development, constant iteration, maximal efficiency, speed, robustness, and more. You want to be lean, you want to be agile. You want to use technologies that will help you succeed in the short- and long-term. And they're not always easy to pick out.

"I'm overwhelmed," you say, as you feel overwhelmed. Your energy is not the same as it once was. You try to piece things together, but it's too much work.

Your proof of concept slowly withers and dies.

The Proposal

After abandoning tons of ideas myself in this way, I decided to engineer a solution. I call it the 'Init' project (or, init.js).

The core of the idea is to have a single project to start them all, to let the developer or the technical founder make all of these essential decisions at once, and receive an appropriate starting template based on those decisions. I know what detractors are going to say, "One solution can't apply to every problem" (haters gonna hate). And they might be right. But we can do our best to create an approximate solution, and I think Init comes pretty close.

To best achieve this objective, we have to keep a few key ideas in mind. When developing Init, I considered:

- **Components**

  Componentization is a key characteristic of any system as it allows you to reuse software components across different projects—which is the main objective of Init. But componentization also comes with a by-product, "replaceability", which will be our best ally in attacking several different problems with “almost” the same solution.

- **Ease of Development**

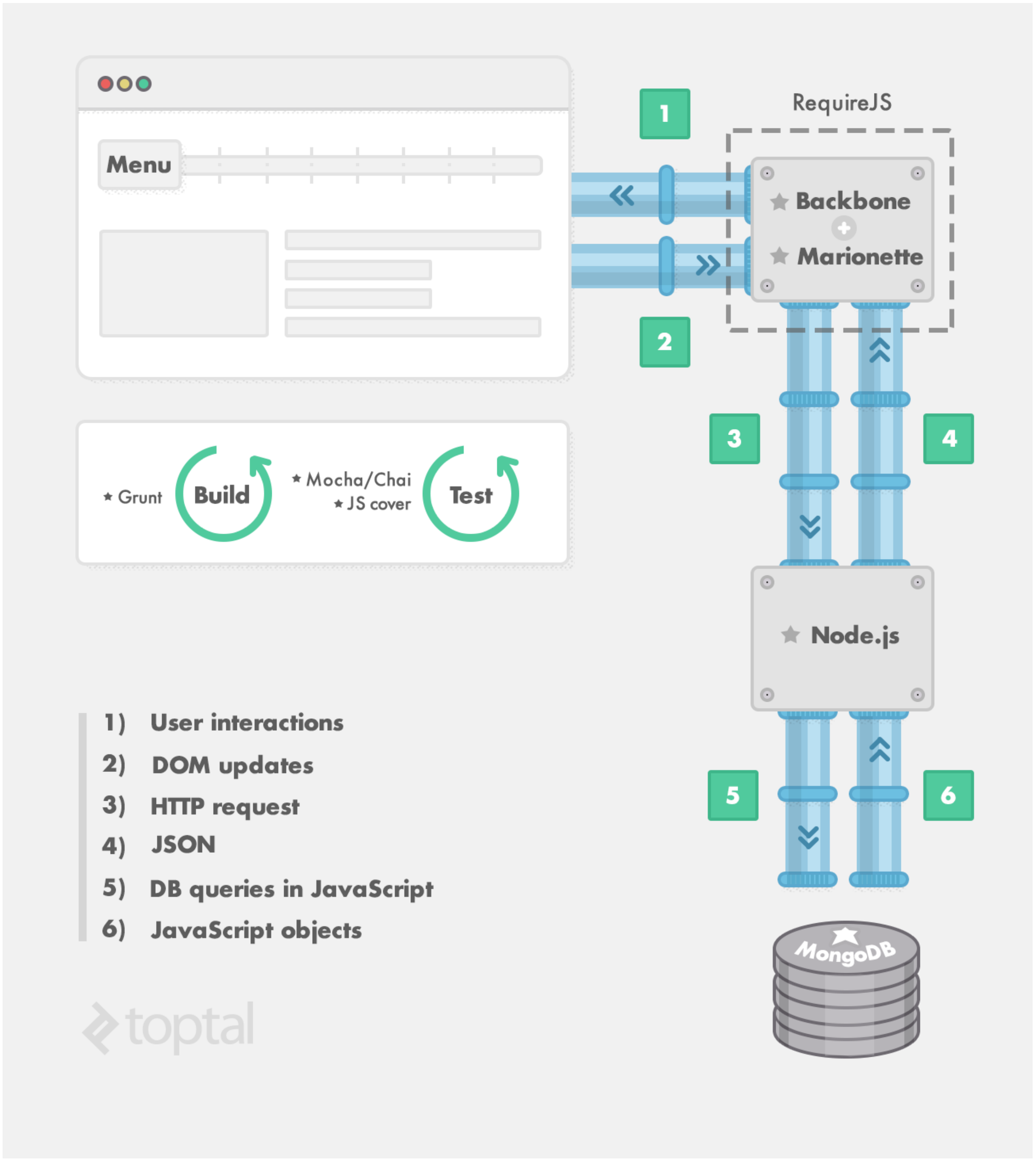
  Some problem, somewhere has a solution best written in Brainf*ck. But implementing that solution (in Brainfuck) will be nearly impossible to write, let alone read. It will cost you time and a tremendous amount of effort. In general, you should use languages and platform that make development easier, not harder for you (or anyone that might work on it later).

- **Community**

  Whatever platform you choose, make sure it has a big community, and one that can help you with the most common and uncommon problems. Remember: jQuery might not be the fastest, cleanest, or most elegant library—but is a winner just because of its community.

Keeping these goals in mind, I’ll next show you how I made my own decisions in creating Init.

At its core, Init takes advantage of the ‘full-stack JavaScript’ paradigm (some people refer to it, or a subset of it, as the MEAN Stack). By working with such a stack, Init is able to use just a single language while creating an incredibly flexible and fully featured environment for developing web apps. In short, Init lets you use JavaScript not only for client and server development, but also for building, testing, templating, and more.



But lets slow down for a moment and ask ourselves: is it really a good idea to use JavaScript?

Why I Chose JavaScript

I've been a web developer since 1998. Back then we used Perl for most of our server side development, but even since then we've had JavaScript on the client side. Web server technologies have changed immensely since then: we went through wave after wave of languages and technologies such as PHP, AP, JSP, .NET, Ruby, Python, just to name a few. Developers began to realize that using two different languages for the client and server environments was complicating things. The initial attempts to unify under a single language tried to create client components on the server and compile them to JavaScript. This didn't work as expected and most of those projects failed (for example: ASP MVC replacing ASP.NET web forms, and GWT arguably being replaced in the near future by Polymer). But it was a great idea, in essence: a single language on the client and the server, allowing us to reuse components and resources (this is the keyword: **resources**).

The answer was simple: put JavaScript on the server.

JavaScript actually born with JavaScript Server Side in Netscape Enterprise Server, but the language simply wasn't ready at the time. After years of trial and error, Node.js finally emerged which not only put JavaScript on the server, but also promoted the idea of non-blocking programming, changing the way we write an "fread" (I/O) forever (read here for more).

In one sentence: non-blocking programming aims to put time-consuming tasks off to the side, usually by specifying what should be done when these tasks are complete, and allowing the processor to handle other requests in the meantime.

But those ideas weren't new—so why did they become so popular with Node.js? Simple, non-blocking programing can be achieved in several ways. Perhaps the easiest is to use callbacks and an event loop. In most languages, that's not an easy task: while 'callbacks' is a common feature in some other languages, an event loop is not, and you often find yourself grappling with external libraries (for example: Python, with Tornado). But in JavaScript, callbacks are built into the language, as is the event loop, and almost every programmer who has even dabbled in JavaScript is familiar with them (or have at least used them, even if they don't quite understand what the event loop is). Suddenly, every startup on Earth could reuse developers (i.e., resources) on both the client- and server-side, solving the "Python Guru Needed" job post problem.

Suddenly, every startup on Earth could reuse developers (i.e., resources) on both the client- and server-side, solving the "Python Guru Needed" job post problem.

So now we have an incredibly fast platform (thanks to non-blocking programing) with a programing language that's incredibly easy to use (thanks to JavaScript). But is it enough? Will it last? I'm sure JavaScript will have an important place in the future. Let me tell you why:

- **Functional Programming**

  JavaScript was the first programming language to bring the functional paradigm to the masses (of course, Lisp came first, but most programmers have never built a production-ready application using Lisp). Lisp and Self, Javascript's main influences, are full of innovative ideas. Those ideas could free our minds to explore new techniques, patterns, and paradigms. And they all carry over to JavaScript. Take a look at monads, Church numbers, or even (for a more practical example) Underscore.js's collections functions, which can save you lines and lines of code.

- **Dynamic Objects and Prototypal inheritance**

  Object-Oriented Programming without classes (and without endless hierarchies of classes) allows for fast development (create objects, add methods, and use them) but, most importantly, reduces refactoring time during maintenance tasks by allowing the programmer to modify instances of objects instead of classes. This speed and flexibility paves the way for rapid development.

- **JavaScript is the Internet**

  JavaScript was designed for the Internet, it's been here since the beginning, and it's not going away. All attempts to destroy it have failed: see, for example, the downfall of Java Applets, VBScript's replacement by Microsoft's TypeScript (which compiles to JavaScript), and Flash's demise at the hands of the mobile market and HTML5. It's impossible to replace Javascript without breaking millions of web pages, so our goals going forward should be to improve it. And there's nobody better for the job than Technical Committee 39 from ECMA.

  Ok, alternatives to JavaScript are born everyday, like CoffeeScript, TypeScript and the millions of languages that compile to JavaScript. These alternatives might be useful for development stages (via source maps), but they will fail to supplant JavaScript in the long run for two reasons: their communities will never be bigger, and their best features will be adopted by ECMA script (i.e., JavaScript). JavaScript is not as an assembly language: it's a high-level programming language with source code you can understand—so you should understand it.

Now thanks to the Esprima project, you can create your own tools to play with the source code, modifying it, changing its style, adding comments, instrumenting, and all kind of things you can imagine by playing with the Abstract Syntax Tree of your program as if you where working with a DOM Tree.

End-to-End JavaScript: Node.js and MongoDB

So, those are the reasons to use JavaScript. Now, I'll use JavaScript as a reason to use Node.js and MongoDB.

- **Node.js**

  Node.js is a platform for building fast and scalable network applications—that's pretty much what the Node.js site says. But Node.js is more than that: it's the preferred runtime environment for any JavaScript application with I/O access. Even if you don't plan to write your main server application with Node.js, you can use tools built on top of Node.js to improve your development process. For example: Mocha.js for unit testing, Grunt.js for automated build tasks, or even Brackets for full-text code editing.

  So, if you are going to write JavaScript applications for the server or the client, you should get familiarized with Node.js, because you'll need and use it on a daily basis. There are some interesting alternatives), but none of them are even at 10% of the Node.js community.

- **MongoDB**

  MongoDB is a NoSQL document-based database that uses JavaScript as its query language, allowing me to complete the end-to-end JavaScript platform. But that's not even the main reason to choose this database.

  MongoDB is a schema-less database that allows you to persist your objects in a flexible way and thus adapt faster to changes in requirements. Plus, it's highly scalable and map-reduce based, which make it suitable for big data applications. MongoDB is so flexible that it can be used as a schema-less document database, a relational datastore (although it lacks transactions), or even as a key-value store for caching responses.

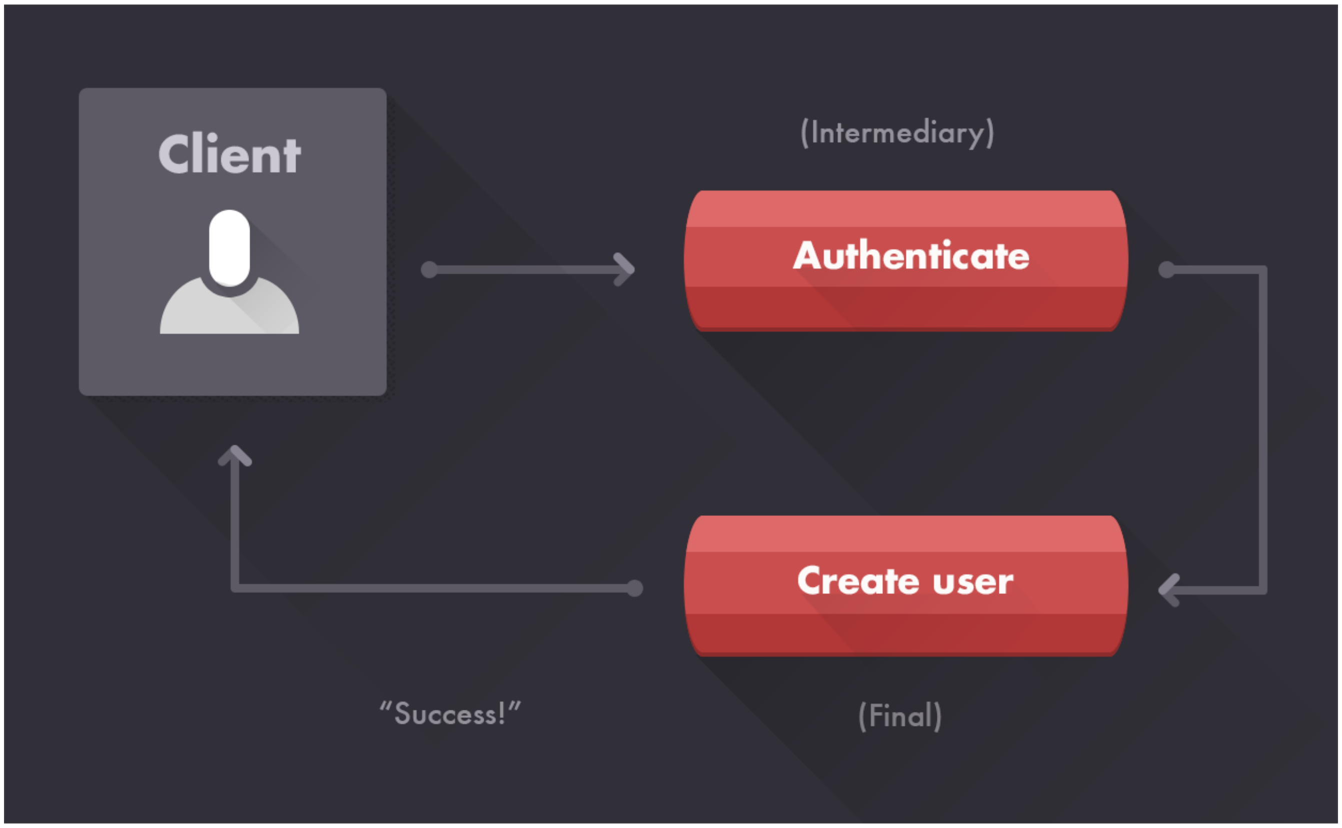
Server Componentization with Express.js

Server-side componentization is never easy. But with Express.js (and Connect.js) came the idea of 'middleware'. In my opinion, middleware is the best way to define components on the server. If you want to compare it to a known pattern, it's pretty close to pipes and filters.

The basic idea is that your component is part of a pipeline. The pipeline processes a request (input) and generates a response (output), but your component isn't responsible for the entire response. Instead, it only modifies what it needs to and then delegates to the next piece of the pipeline. When the last piece of the pipeline finishes processing, the response is sent back to the client.

We refer to these 'pieces of the pipeline' as 'middleware'. Clearly, we can create two kinds of middleware:

- **Intermediates**: Those that process the request and the response, but are not fully responsible for the response itself, so they delegate to the next middleware.
- **Finals**: Those with full responsibility over the final response. They process and modify the request and the response, but don't need to delegate to the next middleware. In practice, it's recommended that you delegate to a next middleware anyway to allow for architectural flexibility (i.e., adding more middleware later), even if that middleware doesn't exist (in which case the response will go straight to the client).



As a concrete example, consider a 'user manager' component on the server. In terms of middleware, we'd have both finals and intermediates. For our finals, we'd have such features as creating a user and listing users. But before we can perform those actions, we need our intermediates for authentication (as we don't want un-authenticated requests coming in and creating users). Once we've created these authentication intermediates, we can just plug them in anywhere we want to turn a previously un-authenticated feature into an authenticated feature.

Single-Page Applications

The Init project focuses on creating single-page applications (SPAs). Most web developers have been tempted more than once to try their hand at SPAs. I've built several (mostly proprietary), and I can say with confidence that they are simply the future of web applications. Have you ever compared an SPA to a regular web app on a mobile connection? The difference in responsiveness is on the order of tens of seconds.

Have you ever compared an SPA to a regular web app on a mobile connection? The difference in responsiveness is on the order of tens of seconds.

SPAs are the future of the web—so why would you build your product in a legacy form? A common argument I hear is that people are worried about SEO. But if you handle things correctly, this shouldn't be an issue: Google itself has a very good tutorial on how to do so, and there are some good comments here as well.

Client Side MV* with Backbone.js, Marionette.js, and Twitter Bootstrap

Much has been said about MVC* frameworks for SPAs. It's a tough choice, but I'd say that the top three are Backbone.js, Ember.js, and Angular.js.

All three are very well regarded. But which one is best for you?

Unfortunately, I must admit that I have very limited experience with Angular.js, so I'm going to leave it out of this discussion. Now, Ember.js and Backbone.js represent two different ways of attacking the same problem.

Backbone.js is minimal, simplistic, and offers you just enough to create a simple SPA. Ember.js, on the other hand, is a complete and professional framework for creating SPAs. It has a more bells and whistles, but also a bigger learning curve.

Depending on the size of your application, the decision can be as easy as looking at the featuresUsed/featuresAvailable ratio, which will give you a big hint.

In the case of Init, I wanted to cover most scenarios, so I picked Backbone.js for easy SPA creation, with Backbone.Marionette.View for componentization. In this way, every component is a simple application, and the final app can be as complex as we want it to be.

Styling is a challenge as well, but we can again count on frameworks to bail us out. For CSS, there's no better than Twitter Bootstrap, which offers a complete set of styles that are both ready to use out of the box and easy to customize.

Bootstrap was created using the LESS language, and it's open source, so we can modify it if need be. It comes with a ton of UX controls that are well documented on the Bootstrap site. Plus, there's a customization model that allows you to create your own. It is definitely the man for the job.

Best Practices: Grunt.js, Mocha.js, Chai.js, RequireJS, and CoverJS

Finally, we should define some of our best practices, and look at how Init can help you implement and maintain them. Our solution is centered on several tools, which are based on Node.js themselves.

- **Mocha.js** and **Chai.js**:

  These tools allow you to improve your development process by applying TDD or BDD, providing the infrastructure to organize your unit tests and a runner to automatically run them.

  There are thousands of unit test frameworks for JavaScript. So why use Mocha.js? The short answer: it's flexible and complete.

  The long answer: it has two important features (interfaces, reporters) and one significant absence (assertions). Let me explain.

  - *Interfaces*: maybe you're used to TDD concepts of suites and unit tests, or maybe you prefer BDD ideas of behavior specifications with “describe” and “it should”. Mocha.js lets you use both approaches.
  - *Reporters*: running your test will generate reports of the results, and you can format these results using various reporters. For example, if you need to feed a Continuous Integration server, you can find a reporter to do just that.
  - *Lack of an assertion library*: far from being a problem, Mocha.js was designed to let you use the assertion library of your choice, giving you even more flexibility. There are plenty of options, but here is where Chai.js comes into play.

  Chai.js is a flexible assertion library that lets you use any of the three major assertion styles:

  - *Assert*: Classic assertion style from the TDD old school. E.g.:

    ```
    assert.equal(variable, ”value”);
    ```

  - *Expect*: Chainable assertion style, most commonly used in BDD. E.g.:

    ```
    expect(variable).to.equal(“value”);
    ```

  - *Should*: Also used in BDD, but I prefer Expect because Should sounds repetitive with the behavior specification ‘it (“should do something..”)’. E.g.:

```
    variable.should.equal("value");
```

Chai.js combines perfectly with Mocha.js. Using just these two libraries, you can write your tests in TDD, BDD, or any style imagineable.

- **Grunt.js**:

  Grunt.js allow you to automate build tasks, anything ranging from simple copy-paste and concatenation of files, to template pre-compilation, style language (i.e., SASS and LESS) compilation, unit testing (with mocha.js), linting and code minification (e.g., with UglifyJS or Closure Compiler). You can add your own automated task to Grunt, or search the Grunt registry, where there are hundreds and hundreds of plugins available (again, using tools with great communities behind them pays off). Grunt can also monitor your files and trigger actions when they are modified.

- **RequireJS**:

  RequireJS may sound like just another way to load modules with AMD, but I can ensure you that it is much more than that. To understand why, we first need to mention the idea of module namespacing (e.g., demo.views.hello), which avoids polluting the global namespace by wrapping each module in its own namespace. The problem is, these modules are not reusable: if you modify the namespace of one 'instance', you're modifying the namespace of all 'instances'. In contrast to that, RequireJS lets you define reusable modules right from the start. (In addition, it'll help you embrace Dependency Injection to avoid having your modules accessing global variables.)

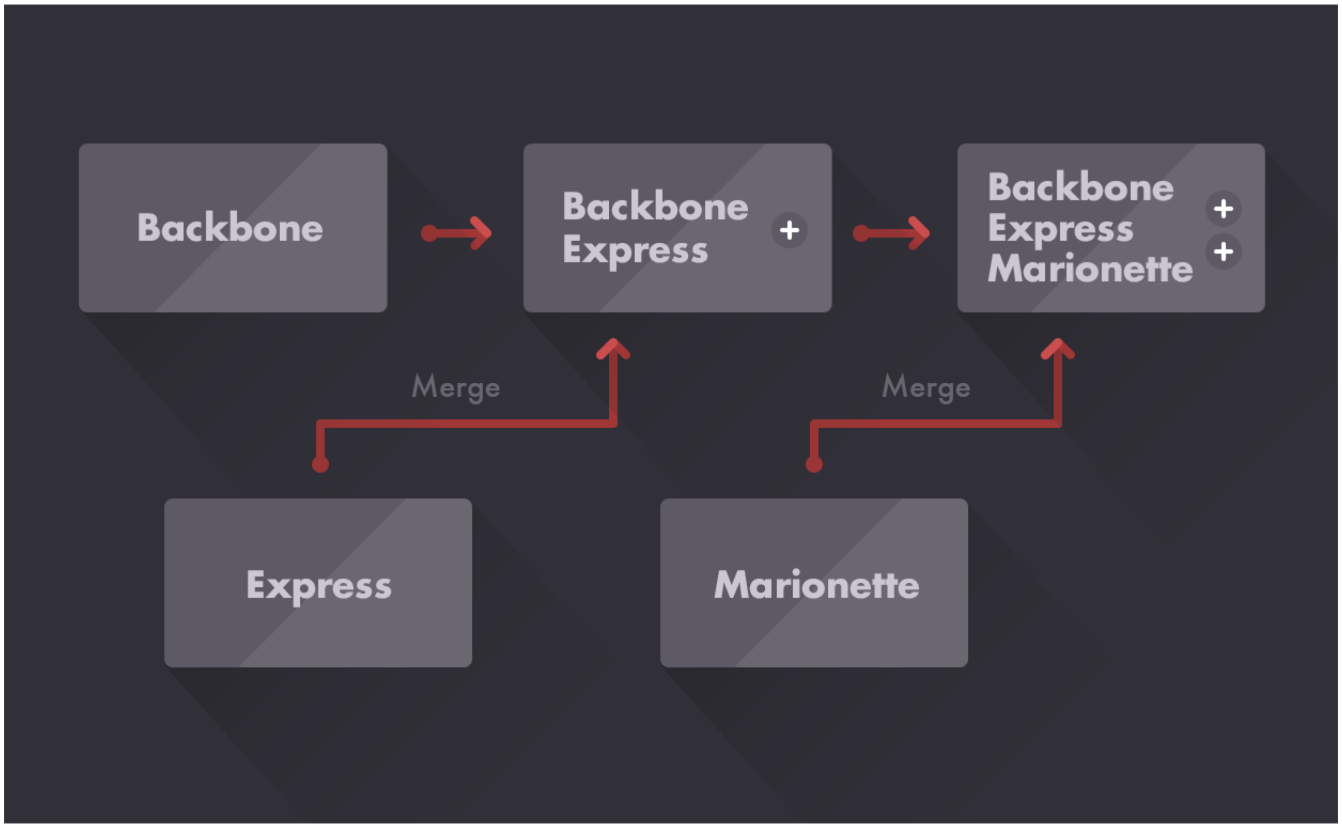
- **CoverJS**:

  Code coverage is a metric for evaluating your testing. As the name implies, it tells you how much of your code is covered by your current test suite. CoverJS measures your tests' code coverage by instrumenting statements (instead of lines of code like JSCoverage) in your code and generating an instrumented version of your code. It can also generate reports to feed your Continuous Integration Server.

Using Branches to Toggle Features

When I started Init, I needed a way for users to activate and deactivate various features that they might want in their project. I decided to take a radical approach to git's branch system to implement this functionality.

In essence, each branch represents a feature or functionality that a user may want to include. If you're starting a project from the ground up, begin on the minimal branch you'd need, and then add other technologies by merging with the desired branches. For example, let say that you want to start your project with Backbone.js and Marionette.js. Well, you can start on the Backbone.js branch and merge it with Marionette branch, continuing onward for every bit of functionality you want to add.



For now, this idea of merging to add functionality can only be used for technology templates (e.g., Backbone, Node, Express). But in the future, you'll be able to switch between back-end (e.g., from MongoDB to Postgres) and client implementations.

Start a Project with Init and Deploy to Heroku Today

There has never been an easier way to start a project. Just head to the GitHub repo, check for the branch with the latest commits (right now it's usermanager, although this might change in the future) and then:

1. Create the directory for your project (or use an existing one).
2. Create your repository with "git init" (or use the existing repository).
3. Add a remote with init

```
git remote add init git://github.com/picanteverde/init.git
```

1. Get the branch you want

```
git pull init usermanager
```

1. Get the Heroku process file

```
git pull init heroku-webprocess
```

1. With the Heroku Toolbelt installed, create a Heroku app

```
heroku create
```

1. Push your master branch to Heroku

```
git push heroku master
```

1. Visit your app, up and running on Heroku!

Now you can start to develop your killer feature with just a few lines of code. Not only that, but you'll be developing with the latest, most efficient technologies in a development suite that's as automated as it can be.

I hope that you can use Init to kick start your next big idea. Remember to check the Init repository for new fixes and features—its development is very much live, and I look forward to hearing your feedback.

Looking to hire top engineers? Check out Toptal's JavaScript developers!

14 comments ★ 17

Join the discussion…

Best Community Share

Avatar hookdump • 4 months ago

Hands down, this is the most awesome technology stack I've worked with. Great article!

14 • Reply • Share ›

Avatar Johan Steenkamp • 4 months ago

Great read - thanks! My stack is Angular, Node and CouchDB - using the same language throughout is really liberating.

I've also been using web sockets more - not just for the real time stuff but also replacing xhr - it's can simplify your code by essentially having an event model between client and server.

With Couch you can extend that to the database as well via the update API.

7 • Reply • Share ›

Avatar Stoikerty • 3 months ago

How come you didn't mention the "meteor" js full-stack project? It goes a step further and removes the need for managing database interactions on client views. Many things happen automatically thanks to meteor's reactivity. For a small team with the usual time constrains, no other technology stack comes close IMO

Of course, it's still in development but so are many other technologies like node. Meteor's community is growing very fast too. If you have the time, check out their smart package philosophy. Hakuna Matata baby

2 • Reply • Share ›

Avatar

lujaw • 4 months ago

great article..

2 • Reply • Share ›

Avatar

Lior Kesos • 3 months ago

Great artice - beyond meteor which is a very viable option in this space (and shaking and reinventing the web as we speak) there is also the MEAN project at http://mean.io which is gaining momentum specifically we use Angular and not backbone but we share many of the motivations and ideas and we totally agree that full stack javascript frameworks are the way to go!
Awesome writeup, very inspirational on the way we should reach out and communicate our vision as well :)

1 • Reply • Share ›

Avatar

Francisco Rimoldi • 4 months ago

It's just GREAT

1 • Reply • Share ›

Avatar

bobiblazeski • 21 hours ago

I've replaced express with socketstream https://github.com/socketstrea... but beside that I'm using almost the same stack. Great article anyway.

• Reply • Share ›

Avatar

Saleh Abdulaziz • 2 days ago

Wonderful article .. Thanks

• Reply • Share ›

Avatar

demesne • a month ago

Nice article.

• Reply • Share ›

Avatar

Marcus Kemper • a month ago

Excellent and very helpful article. Thanks!

• Reply • Share ›

Avatar

Juan Pablo Kaniefsky • 2 months ago

Very nice. I would consider using yeoman for the frontend part, though.

• Reply • Share ›

Avatar

Fizer • 2 months ago

Avatar Fizer • 3 months ago
Really great article. We are using exactly what you MEAN it.
• Reply • Share ›

Avatar andrewvijay • 3 months ago
I'm working on the MEAN stack but still im not confused here cos javascript is a total gift for me! I think there should be a central place for people like you mingle these multiple JS technologies into a single stack. Not a question blog though , some place where all the JS techs can come together make wonders!!!
• Reply • Share ›

Avatar Benjamin Dean • 4 months ago
Very similar to work I've done on Fuel Throttle (open source tech stack), without the HubExchange focus + marionette.
• Reply • Share ›

ALSO ON TOPTAL WHAT'S THIS?

Why Are There So Many Pythons? From Bytecode to JIT
27 comments • 2 months ago
fisadev — Really nice article!! But there is one little thing that isn't correct: you wrote that in python 3 and ironpython the "default

The Trie: A Neglected Data Structure (with diagrams)
8 comments • 2 months ago
Sean Wolcott — Clojure makes extensive use of tries for its immutable data structures, so it's not exactly neglected, just not …

Build Dumb, Refactor Smart: Massaging Problems Out of Ruby on Rails Code
6 comments • 3 months ago
Carlos Alexandro Becker — Hi Daniel, nice article! Following your example, let's suppose that the "project_list_item" is

With a Filter Bypass and Some Hexadecimal, Credit Card Numbers Are Still, Still …
4 comments • 10 days ago
Roman Vashchegin — nice article! Thank you Gergely

Subscribe Add Disqus to your site

Latest articles

- With a Filter Bypass and Some Hexadecimal, Credit Card Numbers Are Still, Still Google-able
- A Step-by-Step Guide to Your First AngularJS App
- Introduction to Responsive Web Design: Pseudo-Elements, Media Queries, and More
- Why Design Documents Matter
- SQL Database Tuning for Developers
- Hunting Memory Leaks in Java
- Why Are There So Many Pythons?

- Building an Infinite Runner on iOS: Cocos2D, Automation, and More
- The Trie: A Neglected Data Structure
- Scaling Play! to Thousands of Concurrent Requests

Show more
Relevant Technologies

- Backbone.js
- Express.js
- JavaScript
- Node.js
- Twitter Bootstrap

Popular authors

Join Toptal:
Hire a developer
or
apply as a developer
Get the fact sheet
A one–page summary
Download

About Toptal

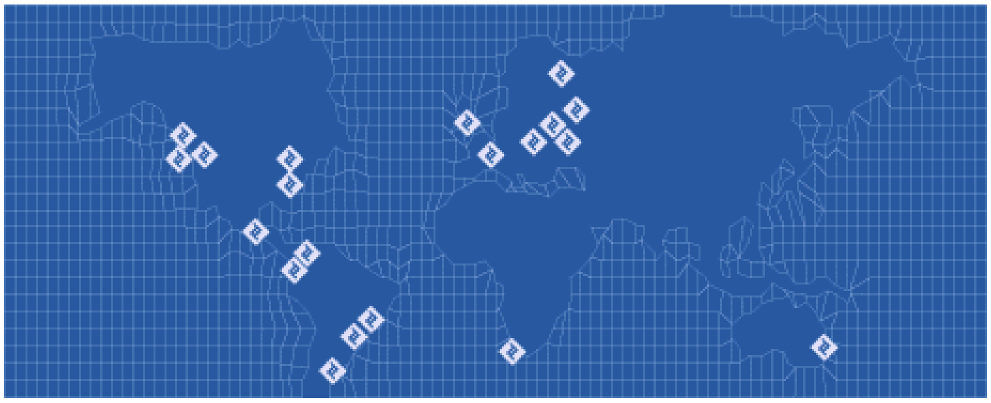
Toptal connects start-ups, businesses, and organizations to a growing network of the best developers in the world.

Our engineers are available full- or part-time and are able to seamlessly integrate into your team.

- What
- Why
- How
- Clients
- Team
- Blog
- FAQ

Contact

- Apply for work
- Become a partner
- Send us an email
- Call 888.604.3188



- Privacy policy

Toptal developers around the World