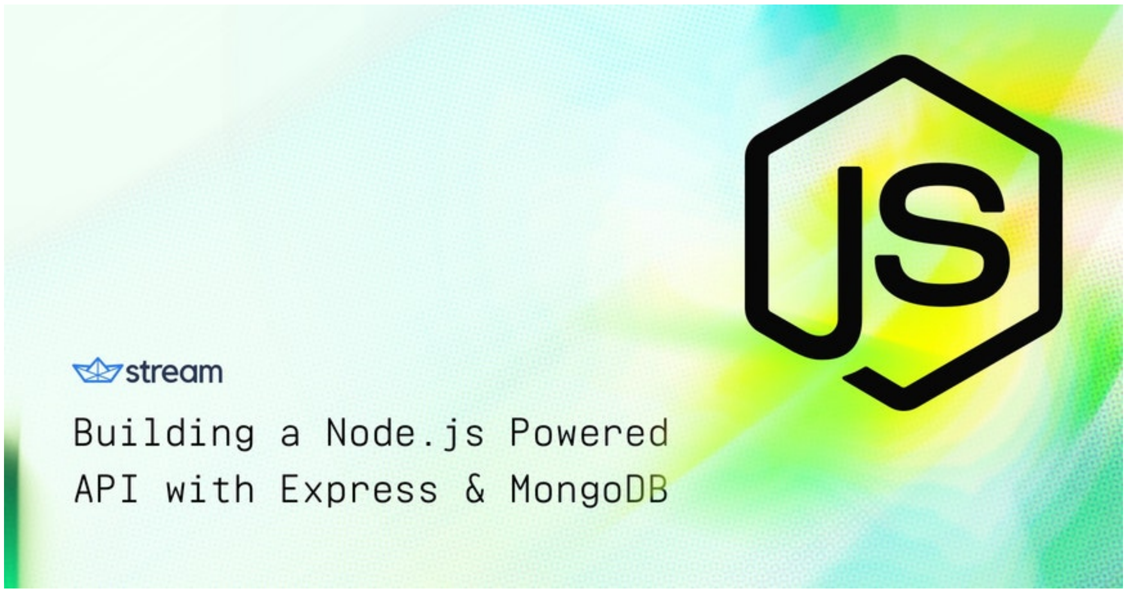
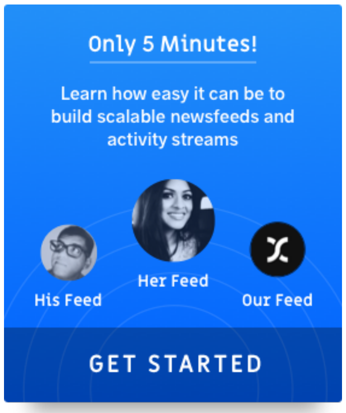


Building a Node.js Powered API with Express, Mongoose & MongoDB

Written by: Nick Parsons (https://getstream.io/blog/author/nparsons/)
Published on: February 22, 2018 at 4:19 pm.
Updated on: February 22, 2018 at 4:19 pm.
Tags: API (https://getstream.io/blog/tag/api/), ES6 (https://getstream.io/blog/tag/es6/), Express (https://getstream.io/blog/tag/express/), MongoDB (https://getstream.io/blog/tag/mongodb/), Mongoose (https://getstream.io/blog/tag/mongoose/), RESTful (https://getstream.io/blog/tag/restful/)

One of my favorite parts of my job as a Developer Evangelist at Stream (https://getstream.io) is building sample applications. It is an enthralling way to engage and interact with potential and existing customers, as well as show off the fun technology we use and build with every single day. The applications I build range from small code snippets outlining how to perform basic operations, such as marking an item as "read" in Stream, to large microservice based applications that generally require robust backend architectures, like Winds (https://github.com/GetStream/Winds).


Only 5 Minutes!
Learn how easy it can be to build scalable newsfeeds and activity streams
His Feed
Her Feed
Our Feed
GET STARTED

Last year, I created a post on how to build a RESTful API with Restify (https://medium.com/@nparsons08/in-depth-guide-on-building-a-rest-api-with-node-js-restify-mongodb-a8e92efbb50f). Now that Express (http://expressjs.com/) and Restify (http://restify.com/) are nearly neck and neck (https://raygun.com/blog/node-js-performance-2017/) in terms of requests per second, I thought it might be interesting to show you how I go about structuring my APIs with *Express* (just to toss in a little friendly competition / play devil-oper's advocate 😉).

Structuring Your API

The way you choose to structure your API is one of the most important decisions you'll make. You must ensure that it's smart, flexible, and easy to use – this is a **must**. If it's not easy to use, other developers will not understand what you're building nor will they be able to figure out how to build on top of it. Think before you build (I know. Planning sucks. Especially when you are excited to get going, but it *pays off*).

```
├── build.sh
├── dist
│   ├── …
├── package.json
├── src
│   ├── config
│   │   └── index.js
│   ├── controllers
│   │   ├── …
│   ├── models
│   │   ├── …
│   ├── package.json
│   ├── routes
│   │   ├── …
│   ├── server.js
│   ├── utils
│   │   ├── …
```

All source code is stored in **/src**. It compiles down from ES6+ to ES5 into the **/dist** directory for execution on the server. You're probably asking yourself why you'd take the extra step to write in something that is just going to be compiled down? Good question. ES6+ standards provide some pretty killer additional functionalities, such as arrow functions, modified scoping, destructuring, rest/spread parameter handling, and more (http://es6-features.org/)!

pm2-node-js-in-production-environments/)

- **Winds – An in Depth Tutorial on Making Your First Contribution to Open-Source Software (https://getstream.io/blog/ an-in-depth-tutorial-on-making-your-first-contribution-to-open-source-software/)**
- **JavaScript: Promises and Why Async/Await Wins the Battle (https://getstream.io/blog/ promises-and-why-async-await-wins-the-battle/)**
- **Simple Steps to Optimize Your App Performance with MongoDB, Redis, and Node.js (https://getstream.io/blog/ steps-to-optimize-your-app-performance/)**
- **Integrating Algolia Search in Winds – A Beautiful RSS & Podcast Application (https://getstream.io/blog/ algolia-search-in-winds-a-beautiful-rss-podcast-application/)**

Let’s have a look at the compilation that takes place in the **build.sh** file:

```
#!/bin/bash

rm -rf dist && mkdir dist
npx babel src --out-dir dist --ignore node_modules
cp src/package.json dist
cd dist && yarn install --production --modules-folder node_modules
```

com/nparsons08/fa0de121579ae9bd35b013238ff7a1f6/raw/0dd8c8002b25c29a2f70c6f0a7596c588753ee6a/build.sh)
build.sh (https://gist.github.com/nparsons08/fa0de121579ae9bd35b013238ff7a1f6#file-build-sh) hosted with ♡ by **GitHub (https://github.com)**

That is ALL you need to be able to write in a super awesome language while having it still be supported in all the usual places! That said, the code above *may* look like gibberish, so let’s break it down 🤓:

1. #!/bin/bash
   - Denotes that this is an executable bash file

2. rm –rf dist && mkdir dist
   - Removes the **/dist** directory if it exists (cleanup).
   - Creates a new **/dist** directory.

3. npx babel src —out–dir dist —ignore node_modules
   - Compiles every file to ES5 and moves the files to the **/dist** directory, with the exception of **node_modules** (those are already compiled).

4. cp src/package.json dist
   - By design, npx doesn’t migrate json files, so we need to copy it ourselves using the **cp** command.

5. cd dist && yarn install —production —modules–folder node_modules

Move into the **/dist** directory and install the npm modules using yarn (https://yarnpkg.com/en/)

Running the build is as simple as running the following command from your terminal:

```
$ ./build.sh
```

.com/nparsons08/a67264e091070bf7b512c9de5a32d4c1/raw/cfa4f782a69778a9547d33187013481eb2bae241/build-

build-1.sh (https://gist.github.com/nparsons08/a67264e091070bf7b512c9de5a32d4c1#file-build-1-sh) hosted with ♡ by **GitHub (https://github.com)**

Note: *You will need to ensure that the build.sh file is executable…*

OR if you are like me and enjoy automating *everything*, you can create an npm script like so:

```
"scripts": {
    "build": "./build.sh"
 }
```

parsons08/36d69aeb777b1d962b727089305f13d0/raw/4101b8461d91557a92f450163773d11855e5ad8b/package.json)
package.json (https://gist.github.com/nparsons08/36d69aeb777b1d962b727089305f13d0#file-package-json)
hosted with ♡ by **GitHub (https://github.com)**

Which can be executed by running the following from your terminal:

```
$ yarn build
```

com/nparsons08/64c88e9e62a46c3097be3f20eda3a02d/raw/b0ddb393fe3017943d95a6e7a589e496cc42bbed/yarn-

yarn-build.sh (https://gist.github.com/nparsons08/64c88e9e62a46c3097be3f20eda3a02d#file-yarn-build-sh)
hosted with ♡ by **GitHub (https://github.com)**

The Main File

The following file, **server.js**, contains the most important logic and sits on the top-level of our codebase. The beginning portion imports all of the necessary npm modules, followed by our **config** and **logger utility**.

Next, we take advantage of the Express **use** method to invoke several of our imported middleware libraries (cors (https://www.npmjs.com/package/cors), compression (https://www.npmjs.com/package/compression), and our body-parser (https://www.npmjs.com/package/body-parser)). **Please note** that there are several other middleware libraries that we include for additional functionality (e.g. email, logging, jwt authentication, etc.). Last but not least, after a bit of initialization, we dynamically include all routes and pass the **API** context to the route for binding.

```
// import npm modules
import fs from 'fs';
import path from 'path';
import express from 'express';
import bodyParser from 'body-parser';
import cors from 'cors';
import winston from 'winston';
import compression from 'compression';
import expressWinston from 'express-winston';
import winstonPapertrail from 'winston-papertrail';
import jwt from 'express-jwt';
```

```


// import custom configuration and utilities
import config from './config';
import logger from './utils/logger';


// initialize the api
const api = express();


// initialize middleware
api.use(cors());
api.use(compression());
api.use(bodyParser.urlencoded({ extended: true }));
api.use(bodyParser.json());


// ignore authentication on the following routes
api.use(
	jwt({ secret: config.jwt.secret }).unless({
		path: [
			'/',
			'/auth/signup',
			'/auth/login',
			'/auth/forgot-password',
			'/auth/reset-password',
		],
	}),
);


// throw an error if a jwt is not passed in the request
api.use((err, req, res, next) => {
	if (err.name === 'UnauthorizedError') {
		res.status(401).send('Missing authentication credentials.');
	}
});


// initialize our logger (in our case, winston + papertrail)
api.use(
	expressWinston.logger({
		transports: [
			new winston.transports.Papertrail({
				host: config.logger.host,
				port: config.logger.port,
				level: 'error',
			}),
		],
		meta: true,
	}),
);
```

```
// listen on the designated port found in the configuration
api.listen(config.server.port, err => {
        if (err) {
                logger.error(err);
                process.exit(1);
        }

// require the database library (which instantiates a connection to mongodb)
        require('./utils/db');

// loop through all routes and dynamically require them – passing api
        fs.readdirSync(path.join(__dirname, 'routes')).map(file => {
                require('./routes/' + file)(api);
        });

// output the status of the api in the terminal
        logger.info(`API is now running on port ${config.server.port} in ${config.env} mo
});

module.exports = api;
```

om/nparsons08/de13406e98461f2cf88568aa7f57fb7a/raw/44a0d9a74b2044b526bb366adc1efc1b505b3ab7/main.js)
main.js (https://gist.github.com/nparsons08/de13406e98461f2cf88568aa7f57fb7a#file-main-js) hosted with ♡ by GitHub (https://github.com)

Note: *The customer logger module can be used with most logging services (Papertrail (https://papertrailapp.com/), Loggly (https://www.loggly.com/), etc.). For this demo, as well as other projects, I like to use Papertrail. You may need to adjust the settings and ENV variables if you use something other than Papertrail.*

Routing

To keep things tidy and organized, all routing logic (e.g. GET /users) is kept in its own route file inside of a **/routes** directory.

```
import User from '../controllers/user';

module.exports = api => {
       api.route('/users').get(User.list);
       api.route('/users/:userId').get(User.get);
       api.route('/users).post(User.post);
       api.route('/users/:userId').put(User.put);
       api.route('/users/:userId').delete(User.delete);
};
```

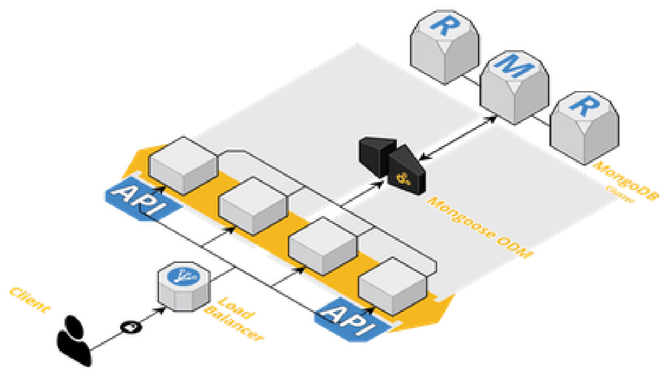
ı/nparsons08/7a6d882e2dfc257777f3297ba3b4c7ab/raw/9c6a859a1a1962ab67c6b2a24fb92622fcbbb2e8/routing.js)
routing.js (https://gist.github.com/nparsons08/7a6d882e2dfc257777f3297ba3b4c7ab#file-routing-js) hosted with ♡ by GitHub (https://github.com)

As you can see, the contents of the route file above hold all references to the controllers for **GET**, **POST**, **PUT**, and **DELETE** operations. This works because we import and reference the **User Controller**, passing along the necessary parameters and/or data with *every* API call.

Controllers

Controllers include the database model associated with the data that they will be handling, receiving data from the routes, and then making an informed decision on how to handle the data. Finally, the controllers communicate through the models which then talk to the database, and return a status code with a payload.

If you're a visual person, a production instance should look a little something like this:



And, the code for an example user controller would look like this:

```
// import npm modules
import async from 'async';
import validator from 'validator';

// import user model
import User from '../models/user';

// import custom utilities
import logger from '../utils/logger';

// retrieve a list of all users
exports.list = (req, res) => {
       const query = req.query || {};

       User.apiQuery(query)
```

```
			// limit the information returned (server side) – e.g. no password
			.select('name email username bio url twitter background')
			.then(users => {
				res.json(users);
			})
			.catch(err => {
				logger.error(err);
				res.status(422).send(err.errors);
			});

};

// retrieve a specific user using the user id (in our case, the user from the jwt)
exports.get = (req, res) => {
const data = Object.assign(req.body, { user: req.user.sub }) || {};

	User.findById(data.user)
		.then(user => {
			user.password = undefined;
			user.recoveryCode = undefined;

			res.json(user);
		})
		.catch(err => {
			logger.error(err);
			res.status(422).send(err.errors);
		});
};

// update a specific user
exports.put = (req, res) => {
	const data = Object.assign(req.body, { user: req.user.sub }) || {};

	if (data.email && !validator.isEmail(data.email)) {
		return res.status(422).send('Invalid email address.');
	}

	if (data.username && !validator.isAlphanumeric(data.username)) {
		return res.status(422).send('Usernames must be alphanumeric.');
	}

	User.findByIdAndUpdate({ _id: data.user }, data, { new: true })
		.then(user => {
			if (!user) {
				return res.sendStatus(404);
			}

			user.password = undefined;
			user.recoveryCode = undefined;

			res.json(user);
		})
		.catch(err => {
```

```
                        logger.error(err);
                        res.status(422).send(err.errors);
                });
};

// create a user
exports.post = (req, res) => {
        const data = Object.assign({}, req.body, { user: req.user.sub }) || {};

        User.create(data)
                .then(user => {
                        res.json(user);
                })
                .catch(err => {
                        logger.error(err);
                        res.status(500).send(err);
                });
};


// remove a user record (in our case, set the active flag to false to preserve data)
exports.delete = (req, res) => {
        User.findByIdAndUpdate(
                { _id: req.params.user },
                { active: false },
                {
                        new: true,
                },
        )
                .then(user => {
                        if (!user) {
                                return res.sendStatus(404);
                        }

                        res.sendStatus(204);
                })
                .catch(err => {
                        logger.error(err);
                        res.status(422).send(err.errors);
                });
};
```

/nparsons08/0b6418f12e78c3d0ef122cecc3124183/raw/6cddcc4dbdd016cfdbc7bb766c92e5c2c81bd834/controller.js)
controller.js (https://gist.github.com/nparsons08/0b6418f12e78c3d0ef122cecc3124183#file-controller-js) hosted with ♡ by **GitHub (https://github.com)**

Mongoose Models (MongoDB)

Mongoose (http://mongoosejs.com/) is a wonderful ODM (Object Data Modeling) library for Node.js and MongoDB (http://mbsy.co/mongodb/228644). If you're familiar with the reference ORM (Object Resource Mapping) and libraries for Node.js, such as Sequelize (http://docs.sequelizejs.com/) and Bookshelf (http://bookshelfjs.org/), Mongoose is pretty straightforward. The massive benefit with Mongoose is how easy it is to structure MongoDB schemas – there's no need to fuss around with custom business logic.

What's even more exciting are the many goodies like middleware, plugins (http://plugins.mongoosejs.com/), object population, and schema validation either baked in, or one yarn (https://yarnpkg.com/en/) (I love yarn) or one npm (https://www.npmjs.com/) install away. It's truly remarkable how popular the project has become among developers who use MongoDB.

When it comes to Mongoose models, I tend to keep things somewhat flat (or at least a maximum of 3 deeply nested objects) to avoid confusion. Here's an example of a user model pulled directly from a project currently under development here at Stream:

```
// import npm modules
import mongoose, { Schema } from 'mongoose';
import bcrypt from 'mongoose-bcrypt';
import timestamps from 'mongoose-timestamp';
import mongooseStringQuery from 'mongoose-string-query';


// import custom utilities
import logger from '../utils/logger';
import email from '../utils/email';
import events from '../utils/events';


// build user schema
export const UserSchema = new Schema(
        {
                email: {
                        type: String,
                        lowercase: true,
                        trim: true,
                        index: true,
                        unique: true,
                        required: true,
                },
                username: {
                        type: String,
                        lowercase: true,
                        trim: true,
                        index: true,
                        unique: true,
                        required: true,
                },
                password: {
                        type: String,
				required: true,
				bcrypt: true,
			},
			name: {
				type: String,
				trim: true,
				required: true,
			},
			bio: {
				type: String,
				trim: true,
				default: '',
			},
			url: {
				type: String,
				trim: true,
				default: '',
			},
			twitter: {
				type: String,
				trim: true,
				default: '',
			},
			background: {
				type: Number,
				default: 1,
			},
			interests: {
				type: Schema.Types.Mixed,
				default: [],
			},
			preferences: {
				notifications: {
					daily: {
						type: Boolean,
						default: false,
					},
					weekly: {
						type: Boolean,
						default: true,
					},
					follows: {
						type: Boolean,
						default: true,
					},
				},
			},
			recoveryCode: {
				type: String,
				trim: true,
				default: '',
			},
			active: {
				type: Boolean,
				default: true,
			},
			admin: {
				type: Boolean,
				default: false,
			},
		},
		{ collection: 'users' },
);


// pre-save hook that sends welcome email via custom email utility
UserSchema.pre('save', function(next) {
		if (!this.isNew) {
			next();
		}

		email({
			type: 'welcome',
			email: this.email,
		})
			.then(() => {
				next();
			})
			.catch(err => {
				logger.error(err);
				next();
			});
});


// pre-save hook that sends password recovery email via custom email utility
UserSchema.pre('findOneAndUpdate', function(next) {
		if (!this._update.recoveryCode) {
			return next();
		}

		email({
			type: 'password',
			email: this._conditions.email,
			passcode: this._update.recoveryCode,
		})
			.then(() => {
				next();
			})
			.catch(err => {
				logger.error(err);
				next();
			});
});


```

```
// require plugins
UserSchema.plugin(bcrypt); // automatically bcrypts passwords
UserSchema.plugin(timestamps); // automatically adds createdAt and updatedAt timestamps
UserSchema.plugin(mongooseStringQuery); // enables query capabilities (e.g. ?foo=bar)

UserSchema.index({ email: 1, username: 1 }); // compound index on email + username

module.exports = exports = mongoose.model('User', UserSchema); // export model for use
```

m/nparsons08/4d399a2b6eabfa938552b07aa1cc51a3/raw/20f6137254eac98a9712fb1b1eb62a5f153d9149/models.js)
models.js (https://gist.github.com/nparsons08/4d399a2b6eabfa938552b07aa1cc51a3#file-models-js) hosted with ♡ by **GitHub (https://github.com)**

Note: *When it comes to hosting and running MongoDB, I like to use* *MongoDB Atlas* (http://mbsy.co/mongodb/228644)*. It's a database as a service provided by the makers of MongoDB themselves. If you don't want to use a free MongoDB Atlas instance, you're welcome to use a local version. Additionally, if you want to monitor your data,* *MongoDB Compass* (https://www.mongodb.com/products/compass) *is an excellent choice!*

Utilities

Custom utilities can be used for a variety of things – basically, anything you want. I generally reserve them for separating concerns and keeping my code clean. Some examples include establishing database connections, sending emails, logging to an external service, and even communicating with HTTP based service here at Stream.

I'm often asked the question of when to turn something into a utility and my answer is always the same… When you find yourself **1)** reusing code OR **2)** jamming third-party services into code where it just doesn't feel right.

Here's an example of a utility I wrote to help called the Stream Personalization REST API (https://getstream.io/personalization/). This integration was completed in about a dozen lines of code:

```
// import npm modules
import axios from 'axios';
import jwt from 'jsonwebtoken';

// import custom utilities
import config from '../../config';

const personalization = data => {
        // setup promise
        return new Promise((resolve, reject) => {

                // build jwt for signing the API call
        const token = jwt.sign(
            {
                action: '*',
                feed_id: '*',
                resource: '*',
                user_id: '*',
            },
            config.stream.apiSecret,
            { algorithm: 'HS256', noTimestamp: true },
        );

// initiate call via axios (http module)
        return axios({
            baseURL: config.stream.baseUrl,
            headers: {
                Authorization: token,
                'Content-Type': 'application/json',
                'Stream-Auth-Type': 'jwt',
            },
            method: 'GET',
            params: {
                api_key: config.stream.apiKey,
                user_id: data.userId,
            },
            url: data.endpoint,
        })
            .then(res => {
                // map over results and deserialize
                const data = res.data.results.map(result => {
                    return result.foreign_id.split(':')[1];
                });
                // successfully resolve call and return deserialized data
                resolve(data);
            })
            .catch(err => {
                // catch and reject with error
                reject(err);
            });
    });
};

export default personalization;
```

.com/nparsons08/138a60d3215a5bf32b881f777eb408bf/raw/434c62d81204559abcf47c30723b398ef462cbe8/util.js)
util.js (https://gist.github.com/nparsons08/138a60d3215a5bf32b881f777eb408bf#file-util-js) hosted with ♡ by GitHub (https://github.com)

The code above can now be called from any file like so:

```
personalization({
    endpoint: '/user_recommendations',
    userId: req.user.sub, // id is extracted from the jwt
})
.then(users => {
    // iterate over users and enrich
    users.map(user => {
        // do something with the user data
    });
})
.catch(err => {
    res.status(503).send(err.response.data);
});
```

om/nparsons08/f4f10bf0e36ab4469f8a5629b1cda1ad/raw/d5b50a741ee77860f77c0b3167181ed64f63f9c5/execute.js)
execute.js (https://gist.github.com/nparsons08/f4f10bf0e36ab4469f8a5629b1cda1ad#file-execute-js) hosted with ♡ by GitHub (https://github.com)

Final Thoughts

APIs are the building blocks of modern applications. They govern how an application can talk to another, as well as to the database. While we have other flavors of API structures (GraphQL (https://graphql.org), etc.), RESTful APIs continue to pull their own weight and aren't going anywhere soon.

If you're interested in seeing a fully built out skeleton for a REST API built with Node.js, Express, Mongoose, and MongoDB, head over to this GitHub repo (https://github.com/GetStream/node-express-mongo-api).

As always, if you have any questions, please don't hesitate to reach out to me on Twitter (https://twitter.com/nickparsons) or below in the comments. Thank you!

Tags: API (https://getstream.io/blog/tag/api/), ES6 (https://getstream.io/blog/tag/es6/), Express (https://getstream.io/blog/tag/express/), MongoDB (https://getstream.io/blog/tag/mongodb/), Mongoose (https://getstream.io/blog/tag/mongoose/), RESTful (https://getstream.io/blog/tag/restful/)

Sketchfab Utilizes Stream's Feed Technology to Increase Performance and Reliability (https://getstream.io/blog/sketchfab-utilizes-stream-to-increase-performance/)
Dubsmash Switches to Stream from an In-House Solution (https://getstream.io/blog/dubsmash-switches-to-stream-from-an-in-house-solution/)

FEATURES

- Get started (https://getstream.io/get_started/)

QUICK LINKS

- Blog (https://getstream.io/blog/)

COMPANY

- Team (https://getstream.io/team/)

BUILD SOCIAL NETWORKS (https://getstream.io/build/social-networks/) WITH

- Overview (https://getstream.io/feat
- Personalization NEW (https://getstream.io/personalization/)
- Customers (https://getstream.io/case_studies/)
- Enterprise (https://getstream.io/enterprise/)

- Activity Feed Design (https://getstream.io/activity-feed-design/)
- Pricing (https://getstream.io/pricing/)
- Docs (https://getstream.io/docs/)
- Based Feeds UI Kit (https://getstream.io/based-feed-ui-kit-sketch/)
- Based Mobile App UI Kit (https://getstream.io/based-mobile-ui-kit-sketch/)

- Careers (https://getstream.io/careers/)
- Contact Us (https://getstream.io/contact/)
- System Status (http://status.getstream.io/)
- Policies (https://getstream.io/legal/)

- Node JS (https://getstre networks/node/)
- Ruby (https://getst networks/ruby/)
- Golang (https://getst networks/golang/
- Python (https://getstre networks/python/)
- PHP (https://getst networks/php/)
- Scala (https://getst networks/scala/)
- Java (https://getstre networks/java/)
- Laravel (https://getst networks/laravel/)
- Meteor JS (https://getst networks/meteorj
- Django (https://getstre networks/django/)
- Rails (https://getst networks/rails/)
- .NET (https://getst networks/net/)

(https://twitter.com/getstream_io)

(https://github.com/GetStream)

(https://plus.google.com/107168613155747146780)