**2018年の最先端バックエンドエンジニアに必要なスキルについて考えてみました。**

[AWS](https://qiita.com/tags/AWS)[gcp](https://qiita.com/tags/gcp)[キャリア](https://qiita.com/tags/%E3%82%AD%E3%83%A3%E3%83%AA%E3%82%A2)

<https://qiita.com/poly_soft/items/fb649573c19b7a5c0227>

536

[@rana\_kualu](https://qiita.com/rana_kualu)さんの[2018年の最先端バックエンドエンジニアになろう](https://qiita.com/rana_kualu/items/19ab559dddbca6674417)という翻訳記事がとても興味深かったのですが、記事内で提示されているロードマップに関して微妙に違和感を感じる部分もありましたので、

* 記事に記載されているスキルは現場でどの程度必要なのか
* 記事に記載されていないが現場において重要なスキルは何か

といった辺りを、自分なりの意見を交えてちょっと書き出してみました。

自分をエンジニアとして最先端だとは全く思っていないのですが、最近のバックエンドのトレンドに一応多少なりともきちんとキャッチアップしてるかなとは思うので、若い方や、まだ経験の短いエンジニアの方たちのご参考になりましたら幸いです。

**言語**

ロードマップに記載されていた言語のうち、私は一応

* Elixir
* Scala
* Java
* .NET (C#とVB.NET)
* Python
* Ruby
* PHP
* TypeScript
* Golang
* Rust

といった辺りに関しては「プライベート」ではなく「業務」においてそれぞれ数ヶ月以上の使用経験があります。(上記以外だとサーバーサイドKotlinも経験があります)

ユースケースが比較的似ている言語もあればそうでないものもありますが、私の個人的意見としては

* 動的型付けのスクリプト言語(Python/Ruby/PHP等)
* 静的型付けのコンパイル言語(Golang/Rust/Java等)
* 静的型付けの関数型コンパイル言語(Scala/Haskell等)

上記の3種類に関してそれぞれ一つ以上(例えばPython/Golang/Scala等)経験しておけば十分なのではないかと考えております。

逆に言うと、例えば「スクリプト言語しか経験がない」とか「コンパイル型言語しか経験がない」とか、あるいは「この言語しかやりたくない」というのはかなりバランスが悪いというか、様々な新しい技術を身につけていく上で、言語が「ネック」というか「足枷」になってしまう可能性があるので、言語に関しては「必要なものはきちんと学びつつ」も「こだわりすぎない」という態度でいた方が、待ちを広くできるような気がします。

(例えば将来なにかとても魅力的なプロジェクトに参画したくなった際に、その業務が「コンパイル型言語」での経験が必須という条件だとすると、**スクリプト型言語の経験しかないエンジニアの方は対象外にされてしまう可能性が高い**でしょうし、「関数型言語以外はやりたくない」みたいなこだわりを持ってしまうと、**せっかく言語以外の部分が魅力的なプロジェクトなのに参加する機会を逸してしまう**ようなことが発生してしまうので、ちょっともったいないような気がします)

ちなみに.NET(C#/VB.NET)に関しては、Web業界のバックエンドエンジニアにとってはあまり使い道がない言語なので、これは無視して問題ないと思います。(アプリ開発にも興味があってXamarinやUnityを使ってみたいという方は別だとは思いますが)

**パッケージマネージャー**

現時点では、かなり多くの言語が◯◯Fileと◯◯File.lockの組み合わせによるパッケージ管理方式を採用していますので、この方式にある程度慣れていれば十分なのではないかと思います。(Pythonも最近は上記方式を採用したPipenvというパッケージマネージャーが徐々に広まっているようです)

ただ、npmに関しては、Web業界におけるJavaScript(TypeScript)の重要度(サーバーサイドでもフロントエンドでもアプリ開発でも使用される)を考えると、知っておいた方が何かと得だとは思いますので、これに関しては業務で使う機会がなくても個人的に勉強しておいた方が良いとは思います。

Scalaのsbtや、Java(Kotlin)のgradle等に関しては、これらの言語に特に関心が無いようであれば、プライベートで勉強しておく必要は無いと思います。

**セキュリティ**

こちらに関しては奥が深すぎるので全てを理解するのは不可能だと思いますが、最低限SQLインジェクション、XSS、CSRF等の基本的な脆弱性攻撃に対する対策手法は押さえた上で、

* 重要な情報はデータベース等に平文で保持しない(必ず暗号化する)
* 秘匿情報をGithub等のソース管理サービスにアップしない(.envや環境変数を活用する)
* クラウド(AWSやGCP)上ではIAM等を使用したアクセス制御をなるべく細かく行っておく
* 常時SSL化

といった辺りの手法も理解しておく必要があると思います。

当然のことながら、セキュリティをきちんと設定しておかないとサービスもしくは会社の破綻に繋がるわけですが、セキュリティ要件を厳しくし過ぎると開発効率が非常に悪くなる(エンジニアの離脱にも繋がる)ので、そのサービスの重要度に応じたセキュリティ設定を行うことが必要になりますし、そういった落とし所に関していくつかの引き出しを持っておくことが重要だと思います。

**テスト**

* Rspec(Ruby)
* unittest(Python)
* testing(Golang)
* ExUnit(Elixir)
* ScalaTest(Scala)

上記のような、担当プロジェクトで使用している各種言語の標準的なテストフレームワークの使用方法に関してはなるべく細かい部分(例えばフィクスチャの設定方法やsetupとteardownの適用範囲等)まで把握しておく必要がありますが、どういったテストフレームワークも基本的には似通った部分を持っているので、一つのテストフレームワークに習熟しておけば、他の言語のテストフレームワークに関しても比較的容易に理解出来るのではないかと思います。

それ以外にも単体テストと統合テストの違いを理解しておくことは重要ですし、DI(Dependency Injection)の概念とそれを使用したテスト方式に関してはエンジニアの必須教養になっていると思いますし、Test Double、モック、スタブ、および状態中心テストと相互作用中心テストの使い分けに関してもある程度理解しておいた方が良いと思います。

[DI・DIコンテナ、ちゃんと理解出来てる・・？ - Qiita](https://qiita.com/ritukiii/items/de30b2d944109521298f)  
[テストダブル - Wikipedia](https://ja.wikipedia.org/wiki/%E3%83%86%E3%82%B9%E3%83%88%E3%83%80%E3%83%96%E3%83%AB)  
[モックとスタブの違い　状態中心のテスト - [lib]](http://d.hatena.ne.jp/devbankh/20100208)  
[モックとスタブの違い　相互作用中心のテスト - [lib]](http://d.hatena.ne.jp/devbankh/20100206)

個人的意見ですが、DBや外部ストレージ等に依存する機能の単体テストに関してはモッキングで対応する方式とローカル環境上に構築したダミーの外部サービスを使用する方式の2種類が存在すると思うのですが、Dockerの登場等によって外部サービスを使用する場合の作業面および速度面のコストが非常に小さくなってきているため、モッキングで対応するよりも単体テストにおいてもローカルの外部サービスに直接アクセスするという方式の方がメリットが大きくなってきている、という印象です。(AWSの場合、各種サービスをエミュレート可能なlocalstackというDockerイメージが存在しますし、CircleCI等のCIサービスもDockerに対応しているので、ローカル上でもCI環境上でも同じ方式で単体テストが行える状況になっています)

ちなみにTDD(テスト駆動開発)に関しては、私も一度は経験してみたいなと思うものの、まだそういった現場でお世話になった経験が無いのでなんとも言い難い部分があるのですが、

* 開発の初期段階からエンジニア全員がその言語およびフレームワークにある程度習熟している必要がある
* ヒットするかどうかも分からないサービスにTDD分の開発コストの増大を許容出来る会社さんは多くはない

といった辺りのことを考えると、ユースケースというか効果を発揮できる条件がかなり限られるというか、今後Web業界内で大きく広がっていく可能性は低い(そもそもTDDが登場してから何年も経過していますが未だにあまり広まっていない)と考えて良いと思いますので、(テスト自体は非常に重要ですし書くのが当然だと思いますが)、TDDに関しては注力して勉強してもあまりメリットがないというのが現時点の状況だとは思います。

**RDBMS**

こちらはもう言わずもがなだと思いますが、NoSQLサービスがたくさん登場している現在においても、リレーショナル・データベースが非常に重要な存在であることに変わりはありませんし、特にMySQLに関しては今後も重要なRDBであり続ける可能性が高いと思いますので、MySQLの基本的な管理方法やコマンドに関しては必ず知っておく必要があると思います。

SQLに関しては、最低限

* 基本的なDDL(CREATE TABLE等)
* 基本的なDML(SELECT/INSERT/UPDATE/DELETE)
* 基本的なDCL(GRANT等)

といった辺りは理解しておく必要がありますし、文字コード、インデックス、トランザクション、デッドロック,マスター・スレーブ等に関しても把握しておく必要があると思います。

最近はRDS等のフルマネージドなRDBを使用することがほとんどだと思いますので、RDBの非常に詳細な部分まで理解しておく必要はないと思いますし、そういった知識が必要になる機会も少ないとは思うのですが、以前と比較すると若いエンジニアの方のRDBやSQLに関する知識がかなり低下してきているような印象なので、かなりもったいないRDBの使い方をしているケースを見かけることが増えているというか、何かトラブルがあった時に大丈夫かなというか、もう少しRDBの基本的な部分に関して勉強しておいて欲しいなと思うことが多いです。

**Webフレームワーク**

* Rails(Ruby)
* Django, Flask(Python)
* Revel(Golang)
* Phoenix(Elixir)
* Finch(Scala)
* Iron(Rust)
* Vert.x(Kotlin)

上記は私がここ数年で実務で使用したWebフレームワークですが、これらに限らず、各言語におけるスタンダードなWebフレームワークのどれか一つに習熟しておけば、その知識を応用して他の言語のWebフレームワークも理解しやすくなると思います。

言語と同様に、こちらに関しても「一つのフレームワークにこだわり過ぎる」のはデメリットの方が大きいと思いますので、「Railsしかやりたくない」みたいな頑固な態度は避けて、「モダンなフレームワークだったら何でもやってみよう」くらいの柔軟な姿勢で色々なフレームワークに手をつけていった方が、知識の引き出しが増えると思います。

最近の傾向として、アプリやSPAのバックエンドとして、HTMLではなくJSONを返すAPIサーバを開発するケースが非常に増えてきていますが、その場合はよりシンプルで軽量なWebフレームワーク(PythonだとFlask等)を求められることが多く、そういった選定においては[TechEmpower](https://www.techempower.com/benchmarks/" \t "_blank)というサイトのWebフレームワークのベンチマーク結果がかなり参考になりますので、もし皆さんがそういう役割を任された場合はチェックしてみると良いかもしれません。

**NoSQL/キャッシュ**

* memcached
* Redis
* DynamoDB
* Cassandra

私がここ数年で経験したNoSQLやKVSは上記のような感じになりますが、DynamoDBに関しては一度経験しておいて損はないというか、RDBとNoSQLの使い分けを考える上では、DynamoDBのホットパーティション問題やRCUやWCUの調整作業等を経験して、NoSQLのメリットとデメリットを実感しておくのは有用だと思います。

私の個人的見解としては、RDBで捌ける箇所に関しては全てRDBに任せてしまった方がよいというか、特にAWSのAuroraのようにフルマネージドかつ簡単にスケール出来るRDBが登場かつどんどん進歩(オートスケール機能やマルチマスタ機能等)していることを考えると、**NoSQLからRDBへの揺り戻しが発生している時期**なのかなというか、ほとんどのサービスにおいてはデータストアはRDBのみで問題なくなるのではという印象ではあるのですが、各種クラウドのNoSQLサービスに関しては一応ある程度の知識を持っておいた方が良いとは思います。

[Amazon Aurora – 自動スケーリングサーバーレスデータベースサービス – AWS](https://aws.amazon.com/jp/rds/aurora/serverless/)  
[【速報】Amazon Auroraのマルチマスタ機能が発表されました！ #reinvent ｜ Developers.IO](https://dev.classmethod.jp/cloud/aws/reinvent-2017-aurora-multi-master/)

**REST API**

REST APIとRESTful APIの違いみたいなことに関して理解する必要はない(RESTという設計思想に従って作成されたAPIがRESTful APIであるという程度の認識でよい)と思いますし、RESTの設計思想の一つであるステートレスという概念を厳密に適用しているサービスは(認証やクッキーが不要なサービスは別として)私は今まで経験したことがないですし、恐らく今後もそういう厳密すぎるAPIの作り方が主流になることはないと思いますので、RESTというガイドラインにほぼ従っているAPI定義の方法というか、REST-likeなAPI定義手法とそこで使われるツールに関して理解しておけば良いのではないかと思います。

[REST入門 基礎知識 - Qiita](https://qiita.com/TakahiRoyte/items/949f4e88caecb02119aa)

具体的には

* GET
* POST
* PUT
* DELETE

というメソッドが実現すべき挙動と適切なURIの設計方法は理解しておく必要がありますし、APIの仕様を記述するフォーマットというか規約としてOpenAPI(Swagger)やRAMLというものが存在しているということは理解しておく必要があると思います。

[GoConの前哨戦として各種API仕様記述フォーマットについて概要を述べておく - Qiita](https://qiita.com/vvakame/items/c21f7862f6d30a6b2a7b)  
[APIドキュメント標準化 現状確認 : エキサイト公式　エンジニアブログ](https://blog.excite.co.jp/exdev/25417365/)

基本的にはOpenAPI(Swagger)が最も一般的だと思いますので他のAPI定義フォーマットは学習しなくても問題ないと思います。(ちなみにAWSのAPI GatewayやGCPのCloud Endpointsは、OpenAPI(Swagger) 2.0を使用してAPIを定義することが可能になっていますが、他のAPI定義フォーマットには対応していないようです)

**認証**

記事の方で定義されている様々な認証方式の詳細や具体的な仕様まで理解しておく必要はないと思いますが、下記の記事等を読んで、OAuthやOpenID Connectの認証フローに関してざっくりと把握しておく必要はあると思います。

[一番分かりやすい OAuth の説明 - Qiita](https://qiita.com/TakahikoKawasaki/items/e37caf50776e00e733be)  
[一番分かりやすい OpenID Connect の説明 - Qiita](https://qiita.com/TakahikoKawasaki/items/498ca08bbfcc341691fe)  
[JSON Web Token の効用 - Qiita](https://qiita.com/kaiinui/items/21ec7cc8a1130a1a103a)

ちなみに、(認証に限りませんが)全ての仕様の詳細を把握しておくことは不可能ですし、大抵のことはライブラリが対応してくれますし、クラウドを使用する場合は例えばAWSならcognitoというサービスが認証周りの処理を片付けてくれますので、おおよその概要を把握しておけば十分だと思います。(cognitoの使用方法を理解するのは中々大変ではありますが)

**メッセージング**

* Amazon SQS
* Amazon SNS
* Cloud Pub/Sub

私がここ数年で使用したメッセージングサービスは上記のようになりますが、SQSは一般的なメッセージキューサービス、SNSとCloud Pub/SubはPub/Sub型のメッセージングサービスになります。

この2種類のメッセージングサービスの違いは、恐らく解説を読んだだけでは分からないというか、自分で実際にサンプルコードを書いてみて実感するのが一番良いと思います。

重たい処理をバックエンドに流したり、なんらかの通知を行うための手法としてメッセージングサービスは非常によく使われますので、これに関しては業務で使用する機会が無くてもプライベートで学習しておいた方がよい技術だと思います。

[よくある質問 - Amazon Simple Queue Service（SQS） | AWS](https://aws.amazon.com/jp/sqs/faqs/)

Amazon Simple Queue Service（SQS）および Amazon SNS は共に AWS 内のメッセージングサービスですが、開発者にそれぞれ異なった利点を提供します。Amazon SNS を利用すれば、アプリケーションは「プッシュ」メカニズムを使用して、複数の受信者にタイミングが重要なメッセージを送信することができます。そのため更新を定期的に確認または「ポーリング」する必要性がなくなります。Amazon SQS は、分散型アプリケーションが使用するメッセージキューサービスであり、ポーリングモデルを通じてメッセージを交換し、コンポーネントの送受信を切り離すために使用することができます。Amazon SQS ではアプリケーションの分散型コンポーネントに対して柔軟性が提供され、同時に利用可能な各コンポーネントを必要とすることなくメッセージの送受信を行うことができます。

**全文検索**

全文検索エンジンに関してはElasticSearchが最も有名かつ一般的に使用されていると思いますし、私が今までお世話になった会社様でもElasticSearchを使用しているチームはかなり多かったので、こちらは個人的に勉強しておいて損はないツールだと思います。

(AWSだとフルマネージドのElasticSearchサービスが使用出来ます)

それ以外にもGroongaやSolrといった全文検索エンジンの名前もよく聞きますが、一般的なトレンドとしてはやはりElasticSearchなのかなというか、それ以外の全文検索エンジンを個人的に勉強してもあまりメリットが無いような気はします。

**Docker**

こちらも言わずもがなだと思いますが、現在業務でまだDockerを使用したことがないという方は強烈な危機感を持った方が良いと思いますし、最低限

* Dockerfileを書いて
* docker buildコマンドでDockerイメージをビルドして
* そのイメージを何らかのリポジトリ(DockerHub等)にdocker pushして
* そのイメージを何らかのVM上にdocker pullして
* そのイメージをdocker runしてコンテナを実行して
* そのコンテナの特定のポートに外部からアクセスする

といった一連の作業を自分一人で出来るようになっておくのは必須というか、上記の作業が出来ない方は**トレンドから相当遅れてしまっている**と考えた方が良いと思われます。

サービスのコンテナ化というのはここ数年のWeb業界の一大トレンドというか完全に一般化してきていますので、もしこの記事を読んでおられる皆さんの会社がまだ本番環境でDockerを運用していないのであれば、多分**その会社は早めに辞めたほうが良い**と思います。(そのくらい重要な技術になってきています)

ローカル環境でのテストや、CI上でのテストに関してもDockerを使用することが当然のような流れになってきています。

**Webサーバー**

記事ではnginxやapacheのことをWebサーバーとして括っているようですが、LAMP環境以外ではapacheはほとんど見かけなくなってきていますし、リバースプロキシとしての用途でもnginxが使われることが圧倒的に多いと思いますので、言語としてPHPを使うのでなければ、apacheを勉強する必要はほとんど無いと思います。

とりあえずnginxの設定ファイルの書き方等に関しては慣れておいた方が良いかなと。

**Webソケット**

ゲームやチャットサービスや、あるいはTrelloのようなサービスを開発する上では必須な技術だと思うのですが、残念ながら私はまだ業務でWebソケットを使ったサービスを作った経験がありません。(Elixir + Phoenixを使ったサービス開発を行っていた際にこれ系の機能も検証してみるべきだったなぁとちょっと後悔しておりますが)

将来的に上記のようなサービスを作りたいと考えている方は是非勉強しておいた方が良い技術だと思います。(私もいずれは業務で使ってみたいと考えております)

**GraphQL**

最近非常に話題になっていますので私も当然興味はあるのですが、一般的に広まっていくのはベストプラクティス的なものがある程度固まってきてからということになると思いますので、今のところは様子見で十分ではないかなと考えております。

[GraphQLを導入してみて得た知見と雑感。GraphQLはタイタニックの救命ボードになりえるかも - Qiita](https://qiita.com/jabba/items/0bf3d2436215394391a7)  
[GraphQLはRESTの置き換えではない｜こんぴゅ｜note](https://note.mu/konpyu/n/nc4fd122644a1)

GraphQLは単一エンドポイントなので、HTTPのCache-ControllのようなURLベースのキャッシュ機構やCDNでのキャッシュはそのままでは使えない。

RESTではエラーはHTTPのステータスコードで表現されるが、GraphQLではリクエストが通れば200で返し、何かエラーが発生した時はエラーオブジェクトに内容が吐かれる。RESTから移行する際はクライアントでのエラーハンドリングも修正が必要となるだろう。

GraphQLは単一のエンドポイントなので、エンドポイントごとにレスポンス性能を監視することはできない。Newrelicでの対応例を以下に挙げたが、なんらかの工夫が必要になってくる。

**Graphデータベース**

AWSでもNeptuneというグラフデータベースサービスがリリース(まだプレビュー版ですが)されましたし、私も興味はあるのですが、Web業界におけるユースケースとしてはSNS的なサービスを作るのでない限りはあまり用途が思いつかないなあという感じなので、こちらもまだまだ様子見で十分なのではないかなという印象です。

**ロードマップには記載されていないが現場において重要な技術**

**コンテナのオーケストレーション**

GCPにおいてはKubernetesとGKE、AWSではCloud FormationとECS(またはBeanstalk/Fargate)という組み合わせ(最近AWSでもKubernetesが使用可能出来るようになりました)になると思いますが、単体のDockerイメージやコンテナを管理するだけでなく、複数のコンテナの組み合わせと、それらのデプロイやオートスケールを管理するための仕組みに関する知見も非常に重要になってきていますので、Dockerの勉強と合わせてKubernetesの学習も必須だと思われます。

[Overview - Kubernetes](https://kubernetes.io/docs/tutorials/kubernetes-basics/)

Kubernetesに関しては、正直ちょっと学習コストが高めというか、PodやService等の概念やその設定方法を理解するのに中々時間が必要(特にネットワークの部分が難しい)だと思うのですが、コンテナのオーケストレーション用ツールとしては一応これがスタンダードなので、勉強しておいて損はないかなと思います。(将来的にはもう少しシンプルなツールが出てくるような気もしますが)

**サーバーレス**

AWSだとLambda、GCPだとCloud Functionsが中心的なサービスになりますが、適切に使用すれば（VMやコンテナを使うよりも)遥かに楽に色々な機能をサービスに組み込むことが出来ますので、こちらに関しても学習は必須だと思われます。

ただし、例えばLambdaの場合は使用言語の制限や同時実行数の制限や最大実行時間の制限、あるいはAWSのVPC環境で実行する場合の起動コストの問題があったりしますので、適切な使い所を見極める必要があります。

**インフラのコード化**

AWSやGCPのWebコンソール上でポチポチやってEC2やGCEインスタンスを立ち上げるという方式は、(開発途中や検証中の場合は別として)もはや完全にレガシーな手法になったと言い切ってしまって良いと思います。

インフラを手順書で管理する方式は、当然のことながら手順書の更新漏れによるオペレーションミスや作業の属人化を防げませんし、そのインフラが今実際どういう構成になっているかを把握することが非常に困難になる等、デメリットが非常に大きい手法です。

AWSであればCloudFormation、GCPであればDeploymentManager、またその両方に対応しているTerraform等を使用すれば、インフラをコードで管理出来ることになりますので、上記のような問題を一掃することが可能です。

もちろん、インフラの全てをコードで管理しようとすると、逆につらみが発生する箇所もありますので、適度な妥協は必要になるのですが、基本的にはコードで管理出来るインフラは全てコード化するという方針にしておいた方が色々な面で楽になりますし、今後おそらく各企業さんでもそういった方法がより一般的になっていくと思いますので、習得しておいて損はない技術だと思います。

**CI/CD**

2018年において、ソース管理用のリポジトリにソースがpushされた際にCIツール上でテストが実行されていないようなチームはまさか存在しないとは思うのですが、もし皆さんの会社がそういうご環境でしたら、(Dockerの節にも書きましたが)それはあまりにもレガシーな環境ですので**なるべく早めに転職された方がよい**と思います。

おそらくCircleCI 2.0を使ってらっしゃるチームが多いと思うのですが、依存しているサービス用のDockerイメージと、テスト実行用のスクリプトを作っておけば、ローカル環境でもCI上でも同じコマンドでテストが実行出来ますし、適切なデプロイスクリプトを書いておけばクラウドへのデプロイも一発で実行出来る(手順書やデプロイ職人が不要になる)ので、ソース管理サービスとCIツールを連携させる知見は、ワークフローを効率化する上では不可欠になってきていると思います。

**DBマイグレーションの自動化**

マイグレーションスクリプトを実行するためのサーバー(踏み台サーバー)等にSSHでログインしてマイグレーションを実行する手法がポピュラーだと思いますが、その処理をCIツール側のスクリプトが実行する場合、踏み台サーバー側でCIツール側のIPからのアクセスを許可しなければなりません。

CIツール側が固定IPなら問題ありませんが、CircleCI等のCIツールはアクセス元のIP範囲が不定なので、想定されるかなり広い範囲のIPレンジからのアクセスを許可する必要があり、これはセキュリティ上非常に好ましくない設定になります。データベースをPublicにしてしまうという方式も可能ですがこちらも当然危険です。

この問題に対しては、AWSの場合であればLambdaをうまいこと使用してマイグレーションを実行するのがベスト・プラクティスであると個人的には考えておりますが、いずれにしてもクラウド環境におけるDBマイグレーションの自動化に関する知見は非常に重要だと思います。

**CDN**

AWSであればCloudFront、GCPであればCloud CDNがこれに該当しますが、何らかの静的ファイル(HTML/CSS/JS/画像等)を配信するサービスを開発する場合は、CDNの知識が必須になります。

基本的には、例えばAWSであればCloudFrontとS3を連携させるだけなのですが、S3のメタデータの設定や、S3へのアクセス制限(CloudFrontのオリジンアクセスアイデンティティの設定)等のノウハウが必要になりますので、機会があれば何かサンプル的なWebサービスを作成して、ここら辺の挙動を確認してみると良いのではないかと思います。

**監視/モニタリング**

AWSであればCloudWatch、GCPであればStackDriverといったフルマネージドなサービスで監視やモニタリングを行うことが可能ですし、小規模なサービスであればそれらで十分だと思うのですが、見やすいダッシュボードを作りたいとかAWSやGCPにアクセス可能なアカウントを持っている人以外にもダッシュボードを見れるようにしたいという場合にはCloudWatch等では対応出来ませんし、細かな使い勝手等を考慮するとDatadog等のサードパーティの監視サービスを使用する必要が発生する場合もあります。

ZabbixやNagios等のアンマネージドなサービスを自分達でクラウドにホストしたりするというケースはかなりレアだとは思うのですが、Datadog,Mackerel,NewRelic等のマネージドなサードパーティサービスは一般的によく使われていますので、こちらに関しては機会があれば是非触ってみることをお薦めします。

[モニタリングサービス（Mackerel、Datadog、New Relic）入門！ - Qiita](https://qiita.com/uzresk/items/fa58720acbe7672607c9)

**ログ基盤/分析基盤**

TreasureDataとGCPのBigQueryが双璧だと思いますが、料金面において(使い方を誤らなければ)大抵のサービスにおいてはBigQueryの方がお得なので、こちらを使っている会社さんが多いという印象です。

バックエンドエンジニアの場合、分析基盤にクエリを投げて分析を行うというよりも、アプリケーションから分析基盤に対して安定してデータを送り込む手法の知見の方が重要ということになると思いますが、こちらに関してはfluentdを使う方式が一般的だと思いますし、fluentdは非常に色々な企業さんで使われているので、fluentdの使用方法や設定方法に関しては、なるべく早めに実務経験を積んでおいた方が良いと思います。

**機械学習**

こちらは人によって意見が分かれるところだとは思うのですが、クラウドの登場によって、サーバーサイドエンジニアがインフラの構成管理等を一人で行うことが可能になったように、機械学習の難解な部分がクラウドや各種ライブラリやツールによって抽象化されていくことによって、**優秀なバックエンドエンジニアであればサービスの機械学習の部分も全て一人で担当出来てしまう**という時代がすぐそこまで来ているというのが私の個人的な印象です。

(もちろん機械学習部分に対する要求が非常に高い場合は専門の機械学習エンジニアやコンサルタントの力をお借りすることが必要になると思いますが、ある程度の精度のものをクラウドのマネージドサービスを使って迅速にリリースしていきたいというようなケースにおいては、バックエンドエンジニアが全てを担当するというケースは今後どんどん増えていくのではないかと)

また、現在私は機械学習やデータサイエンスを専門にしているチームにジョインさせて頂いているのですが、機械学習部分のインフラやワークフローというのは、モダンな環境で過ごしてきたバックエンドエンジニアから見るとかなりレガシーと感じる場合が多く、それらをモダン化していきたいというかなり強いニーズが顕在化してきていると認識しております。

私の予想ですが、今後機械学習ワークフローやインフラのモダン化というのはWeb業界における非常に重要なトピックになっていくと思いますので、機械学習そのものの知見だけでなく、機械学習周辺の各種のテクノロジーに詳しくなっていくことは、バックエンドエンジニアのキャリアを進展させる上で大きなメリットがあるのではないかと考えております。

ちなみに以前下記のような記事を書きまして、最近やっとKaggleを始めたのですが、機械学習の勉強は確かに大変ですが楽しいです♪(^.^)

[文系エンジニアが機械学習に入門するために小学校の算数から高校数学までを一気に復習してみました。 - Qiita](https://qiita.com/poly_soft/items/8ea0fc85577f0726f098)  
[文系エンジニアがCourseraの機械学習コースを1ヶ月で修了したので振り返ってみました。 - Qiita](https://qiita.com/poly_soft/items/0f7c09470af4ad5dbd39)

**まとめ**

実際にはもうちょっと必要な技術が色々あるかなとは思うのですが、書いている途中で力尽きたので本日はこの程度にしておきたいと思います。(後日また改めて追加しておきたいと思います)

**こんなにやらなきゃいけないの？**と思う方もいらっしゃると思いますが、「やらなきゃいけない」というよりも、現在お勤めされている会社やチームで上記のような経験が積めないということであれば、その環境は「レガシー」であるという可能性が高いと思いますし、レガシーな環境に長く居続けるほど、モダンな環境に移行出来る可能性がどんどん小さくなっていってしまうということは認識しておいた方が良いと思います。

全てにおいてモダンな企業さんやチームというのは私も見たことが無いですが、少なくとも「レガシー環境で歯を食いしばって頑張る」というのは、旧来の日本的な価値観においては「美徳」ということになると思うのですが、最前線のバックエンドエンジニアを目指す上ではほとんどメリットがありませんので、レガシー環境を改善していくか、もしくは他のチームや会社さんに移るか、どちらかを考えた方がいいかなと。

SIerさんではなくWeb業界で働くことの醍醐味というのは**モダンな面白い技術にどんどん触れて毎日ウキウキした気持ちで働ける**ということにあると思いますので、レガシー環境で我慢するのであれば**Web業界で働く意味がないのでは**というのが私の意見です。

**おまけ**

Youtubeの方で、ITエンジニアやIT業界に興味のある方たち向けの[雑食系エンジニアTV](https://www.youtube.com/channel/UC_HLK-ksslL-Z_2wiIZDlMg" \t "_blank)というチャンネルを始めました。

最近はなかなか更新出来てないのですが、もしご興味ございましたらチャンネル登録してみて頂けると大変嬉しいです(^.^)

[**勝又 健太@poly\_soft**](https://qiita.com/poly_soft)

雑食系エンジニア/サービス開発アドバイザー/顧問エンジニア/IT系Youtuber/小説家。最近の使用言語はElixir・Go・Scala・Rust・Kotlin・TypeScriptなど。Youtubeのチャンネル「雑食系エンジニアTV」はこちら → https://goo.gl/7EnJkM