# Article Summaries & Keywords

Below are the summaries and keywords for every article submitted for the contract with Unique Software Development to deliver tutorials that explain software development to readers of the site at https://www.uniquesoftwaredev.com

## Part 1. Setup Local Serverless Development

The aim of this tutorial is to deploy a simple application on AWS to share the joys of developing applications on the Serverless Framework with AWS. The application we will be walking you through includes a backend API service to handle basic CRUD operations built on something we call the DARN Technology Stack that includes the following tool set:

* DynamoDB
* AWS Serverless Lambda
* React.js
* Node.js

The article discusses the idea of a JavaScript Toolkit to more easily onboard new engineers onto a fast-moving team of developers to better and more effectively contribute to en enterprise level project in JavaScript. The tutorial reviews the proper installation of tools like nvm, eslint, code formatting, and the proper configuration of standard tools on your local machine and your IDE.

One of the more difficult activities facing junior developers is understanding how to properly configure your machine locally and this tutorial will show the reader exactly how to start any project with ease and confidence.

### Keywords

Serverless, AWS Lambda, Local Development, JavaScript Toolkit, Node.js, ESLint, SublimeText3, NVM, IDE Configuration

## Part 2. GoServerless on AWS

Serverless programming and computing is a software architecture that enables an execution paradigm where the cloud service provider (AWS, GoogleCloud, Azure) is the entity responsible for running a piece of backend logic that you write in the form of a stateless function. In our case we are using AWS Lambda and the cloud provider you choose to run your stateless function, is responsible for the execution of your code in the cloud, and will dynamically allocate the resources needed to run your backend logic, by abstracting the deployment infrastructure for you, so that you can focus on developing your product instead of auto-scaling servers.

Since the serverless paradigm abstracts away the need for an engineer to configure the underlying physical infrastructure typical to the deployment of a modern day application, in what is known as the new Functions As A Service (FAAS) reality, the following are a few considerations that should be kept in mind while we proceed through the development of our Single Page Application:

* Stateless Computing
* Serverless + Microservices
* Cold Starts

To deploy our demo application with a serverless backend to handle our business logic with independent functions deployed to AWS Lambda, we will need to configure Lambda and APIGateway to use the ServerlessFramework. The ServerlessFramework handles the configuration of our Lambda functions to use our code to respond to http requests triggered by APIGateway. The ServerlessFramework lets us use easy template files to programmatically describe the resources and infrastructure that we need AWS to provision for us, and on deployment, AWS CloudFormation does the job of instantiating the cloud based infrastructure that we call the serverless architecture on AWS. The serverless.yml file is the file that executes the explicit resources that we declare from within the ServerlessFramework, to tell AWS CloudFormation what we need from AWS to run our application.

### Keywords

Serverless, AWS Lambda, MicroServices, Serverless Framework, FAAS, AWS, AWS CLI, APIGateway, CloudFormation

## Part 3. Configure Infrastructure As Code

The ServerlessFramework lets you describe the infrastructure that you want configured for your serverless + microservice based application logic. You can use template files in .yml or .json format to tell AWS CloudFormation what exact resources you need deployed on AWS to correctly run your application. The YAML or JSON-formatted files are the blueprints you design and architect to build your services with AWS resources. We can see that by using the AWS Template Anatomy to describe our infrastructure, that the templates on AWS CloudFormation will include a few major sections described in the template fragments shown below:

* Format Version (optional)
* Description (optional)
* Metadata (optional)
* Parameters (optional)
* Mappings (optional)
* Conditions (optional)
* Transform (optional)
* Resources (REQUIRED)
* Outputs (optional)

We also show you how to mock, or fake the input parameters for a specific event needed by our Lambda’s with a .json file to be stored in a directory within the serverless + microservice project that we will use by executing the ServerlessFramework’s invoke command. The invoke command will run your serverless + microservice code locally by emulating the AWS Lambda environment.

Furthermore, the tutorial discusses how to implement Automated Testing with typical unit tests that will execute individual software modules or functions, in our case our unit tests will execute our Lambda functions on the AWS Cloud. When implementing your tests, you really want to try to make them useful, if not at least relevant to the goal of your application’s business logic. You really want to take some time to think of any edge cases that your users may be inputting into your application to ensure that your application’s user experience meets your user’s needs and expectations. If you are working as part of a team, you really should collaborate with them on the different test cases that you should implement to mitigate any potential errors, that your users may confront.

### Keywords

API Gateway, CloudFormation, DynamoDB, Serverless, AWS Lambda, MicroServices, S3, Infrastructure As Code

## Part 4. Configure Backend CICD Pipelines

By using an Agile development environment, we just want a process that we can use to iterate over a predefined Code Review workflow that will help us implement and merge new updates to our source code efficiently, transparently, and with close to zero downtime in the Wild. When a team member writes and implements a set of features, there should be someone, again, in my case Wilson, who will review the code you have implemented on a topic-branch in git after you create a Pull Request for your project lead to review your code.

The Code Review process is an important part of your team workflow because it allows you to share the knowledge you gained from the implementation of the logic and functionality that defines the feature you will deploy. It also gives you a layer of quality assurance that enables your peers to contribute and provide insight into the feature you will deploy, and it allows new team members to learn from others on the team by taking ownership of a feature and implementing the logic the new feature needs so that it can be accepted by the project owner.

The tutorial also discusses few recent updates to the AWS Lambda service that you should be aware of. The Node.js AWS Lambda runtime environment now supports Node.js v.10.14.1. In your serverless.yml file you should declare the runtime you will use as runtime: nodejs10.x to make sure that you can deploy your Lambda function correctly and with the latest supported features.

Throughout the remaining articles in this tutorial series we will indicate the newest updates that we have discovered and have had to evolve with using a tag like this:

* UPDATE: This will discuss the update in question

Moving forward with our project, and this tutorial, we can now take some time to discuss and understand the principles, practices, and benefits of adopting a DevOps mentality. We will also study and review concepts in Continuous Integration and Continuous Delivery and we will really start getting comfortable deploying enterprise ready software to the AWS Cloud. Just to make sure you are ready, we will review and get you comfortable with commiting your code to a Version Control repository on something like GitHub, and I’ll show you how to setup a continuous integration server and integrate it with AWS DevOps tools like CodeBuild and CodePipeline.

The idea is to apply software development practices like quality control, testing, and code reviews to infrastructure and feature deployment that can be rolled into production with little intervention and minimal risk. Transparency is prioritized so that every team member has a clear view at every stage of the development and deployment process from its implemetation by the dev team, all the way to the operations team that monitors and measures your application’s resources and infrastructure deployed in production.

### Keywords

CodeBuild, CodePipeline, DevOps, CICD, AWS, Git, Agile, Code Review, Deployment, Automation, App Development, IT Operations

## Part 5. Building Serverless + MicroServices

The article starts by defining the NoSQL tables that we will need to implement in AWS DynamoDB within an isolated environment, so that we can be sure to develop the appropriate data model needed for this specific serverless + microservice. Furthermore, in a microservice environment, each service will implement its own database. In the case of NoSQL, its own table (more on this to come).

The idea behind a microservice based architecture, for those of you not already in the know, is to be able to more easily maintain your code, and extend an infinite amount of features that you believe will save the world, in a decoupled environment that will allow you to build out new functionality without impacting the work of your team implementing their own versions of this application. Simply put, we just want to write code and develop services that only deal with one specific thing or task.

We will use a loosely coupled architecture that makes it easy to develop, test, and deploy new features independently of each other, and to maintain more control over the stability of the system. No two services should rely on data from each other or any other source, nor should they know anything about the other’s state. In a serverless + microservice environment, each microservice is going to have its own database, that will deal with the specific attributes that the service in question needs, to provide the correct response to any given request, while using the data it persists to its own data store.

Key Types determine how your application can access the data it collects later on. There are two Key types you can use to define for your table, furthermore all Key Type Attributes MUST be decided upon in advance. We can use either SimpleKey or CompositeKey types. To take advantage of the Distributed Hash Map Architecture that enables DynamoDB’s high performance as a Key:Value Document Storage database, we will use a CompositeKey.

DynamDB also helps us balance our costs against the availability of our database by letting us autoscale our database to meet the needs of our users. Using the settings in this section will need careful consideration on your part because these can lead an unexpected surge in your AWS Costs if your application goes viral. These settings will allow your database to scale to meet the increasing demand of users on your application. Keeping these settings static would prevent your database from responding to queries that exceed an arbitrary threshold. Instead, we will let it grow with the needs of our users.

Using AWS Cognito, we will be able to easily implement user registration, authentication, authorization, and management for our application. AWS Cognito is flexible enough to let us implement SSO using Federated Identities with third party Identity Providers (IdP) like Facebook, LinkedIn, or Twitter also. Our user-pool on Amazon Cognito will manage and handle the load of responses that include the authorization tokens returned from each of these social media sign-in federations and SAML IdPs.

Amazon Cognito is both PCI DSS and HIPAA compliant and is ready to deploy to production. The idea is to create a directory of users with your user-pool that will allow users to sign in to our application. AWS-Amplify is the Amazon SDK that lets us access the user profiles that are created for each of our users whether they sign up with our implementation of Cognito, or using a federated identity through a thrid-party IdP that we enable on our application.

Infrastructure As Code is a great way to implement Cognito and the tool set you will use on your application to deploy registration, authentication, and authorization of a user. You can complete the implementation of a user-pool on the AWS Console, but we will focus on completing what we need programatically so that we can commit all of our changes to source so that our CI/CD pipeline can deploy our changes automatically.

Here we create an object storage directory on the cloud with Simple Storage Service or S3 on AWS. We want to deal with the attachments that our users may add to every invoice or transaction that they add to each entry in our new GeneralLedger table that we implemented above.

Later on, when we walk you through building the React.js UI in the next tutorial, you will take advantage of AWS Amplify and the web development SDK provided by AWS to use an easy to use ReST API that will let us store all of our user’s attachments to this new S3 bucket we are implementing as Infrastructure As Code to deploy on CloudFormation.

### Keywords

DynamoDB, Cognito, Authentication, Data Modeling, Federated Identities, Simple Storage Service, s3, AWS, Infrastructure As Code, CloudFormation

## Part 6. Building A ReactJS FrontEnd

Now that we have a bunch of fancy backend logic on our AWS Cloud, we need to develop an engaging user interface that we can use to attract real humans to our latest get rich quick idea, we are going to continue to call it a digital Wallet to make mom, and our resumes, proud. Rather than get into the long history that led to React.js’ role as the ‘V’ or the View in the Model-View-Controller design paradigm, we will spend some time implementing React.js so that you can learn by doing.

Since the Facebook Mafia is much smarter than us, we are just going to go ahead and use something called Create React App, or create-react-app for those who are more programatically inclined. All sarcasm and Dilber Humor aside, React.js is a practical approach to another one of those esoteric software engineering questions that exist in an abundance of threads on the internet that scroll of into perpetuity. Implementing frontend component in React.js is effective and more productive approach that allows you to reuse code that you have written throughout your code. React takes advantage of a Virtual DOM that ituses to check against state changes to render dynamic UI elements faster than traditional MVC implementations.

The most important part of any web development project is making sure to have a good set of tools that will let you create a beautiful user interface that your user can use to interact with your serverless backend. You can definitely spend your days implementing your own elements in JavaScript, HTML, and CSS and maybe create your own UIKit that you can market to the world to become famous, but for the purposes of this tutorial we are just going to go ahead and use the Bootstrap UI toolset via the React-Bootstrap library that we will import into each of our components as needed.

We now need to consider a way to route the different requests that we make available to the users of our Single Page Application. We will take advantage of the nature of how components work within React.js and we will use something called React Router to use its collection of navigational components that will let us easily declare the routes that our applications needs to handle for our users and their requests.

Traditional web-architectures using a client-server approach lead to a poor user experience because of the need to re-render dynamic pages for every click of a button that needs to wait for an asynchronous response from a server. On the contrary, Modern Web Applications use a Single Page Application approach to package all of the application components into a single static file to offer a native-like feel to the user using brebuilt layout and JavaScript files to execute backend logic without having to re-render the page content.

AWS-Amplify just provides everything that we need to implement and deploy scalable cloud based apps that uses a CLI and library that really simplifies the development of web and mobile applications. In the next article we will even go over and show you how to deploy your app onto an infinite amount of stages by using AWS-Amplify.

When the state of the application is ready to submit data to our services on the cloud, we then use Amplify to call our resources in Cognito to authenticate a user. Take a look at the asynchronous promise returned by the handleSubmit() function shown above. We are using our state sttributes to call Amplify’s Auth.signIn() method to validate our credentials and authenticate our user. Using the await feature to return a promise, we execute Auth.signIn() to securely sign our user into the app.

### Keywords

React.js, AWS-Amplify, Bootstrap4, Modern Web Applications, Single Page Applications, JavaScript Promise, Asynchronous Requests

## Part 7. Serverless & React Integration with Multi-Step Registration

Now that this application of ours has a login interface that our users can use to authenticate themselves, we are going to have to make sure that there is a mechanism in place to enable the permissions our users need to create and manage their transactions within our app. To properly complete these configuration steps we will need to be able to load our application’s current state from our user’s login session managed by Cognito, we’ll need to implement a few redirects for proper login and logout functionality, and more importantly we always have to make sure that we are giving our users good feedback when logging in so that they can be sure that they are submitting the correct credentials.

Having declared an object that will hold the value of the authentication state that you need to pass into each rendered component used by your application, you now need to pass the object and its values into the component that is declared at the end of the render() function in your src/App.js file.

Let’s take a moment to talk about an advanced technique used in React.js as an emergent design pattern, because of its ability to allow developers to implement composite functions. Yes a function as an input to another function, or in the case of Higher Order Components in React.js, a function that accepts a Component as an argument to return a new Component that we canleverage to reuse Component logic. We need to use this concept to implement an HOC that will create a new Route for us that renders a new Child Component that will have the property values that have to be stored for use in the UI to deal with the appropriate state changes triggered by our users.

We are now ready to start adding a few more interactive UI elements into the layout of the app. We need to apply the authenticate state functionality so that we can let our users interact with our app on Sign Out. When a user is authenticated, they will need to see a button that will let them sign out of our application also. We will accomplish this in React.js using a component. A Fragment is great for creating groups of child elements without cluttering the DOM full of extra nodes.

Typically we can make use of Cookies or LocalStorage to store the user’s sign-in data that we can load from the session stored in the browser. In the case of a Progressive Web Application we can use these tools to persist offline data to allow our users to work with our app in a native-like environment. With AWS-Amplify we can store our session information automatically and use Amplify to load the session information we need when a user signs in, into the application’s state.

Using asynchronous methods to get a backend response from our serverless + microservices does introduce a bit of latency into the application that we will have to address so that our users will know that the app is working and has not crashed while waiting for the data requested. We will need to create a state attribute that we can use as a flag to determine when the application is awaiting an asynchronous response filled with data requested by the user from our UI. In our Signin component we will use an isLoading flag as a state attribute for the purposes we have just discussed.

In JavaScript, this binding really depends on how our functions are invoked; because of the way that React.js works with ES6 and Class Components, our event handlers and controlled components will point to an undefined value when our functions are used in strict mode. They will all fall back to their default bindings after losing their context. These functions must be implemented in our application’s constructor explicitly with a hard bind to the this value, while using the bind() method to persist the application context to the component rendered to the UI.

You have implemented a multi-step registration process while integrating a few of your services with React. In the next article we will discuss how to implement another feature that creates a database entry and we deploy all of these new features that take advantage of your GeneralLedger in the cloud on AWS Amplify.

### Keywords

Multi-Step Forms, Authentication, Higher Order COmpionents, Session, JWT, LocalStorage, Cognito