**Eventique**

**Authors:** Aditya Sharma, Parth Mishra  
**Mentor:** Dr. Naween Kumar  
**Institution:** Bennett University  
**Date:** 29/04/15

**Abstract**

Eventique is a serverless web application designed for managing and registering events. It enables users to securely sign in, browse available events, register for selected events, and view their personal event registrations. The application uses a React-based single-page frontend hosted on AWS S3 and delivered via CloudFront, and a backend composed of multiple AWS Lambda functions behind API Gateway. User authentication is handled by Amazon Cognito, which issues JWT tokens for secure access. Application data (events and registrations) is stored in Amazon DynamoDB. The entire infrastructure (Cognito, API Gateway endpoints, Lambda functions, DynamoDB tables, S3 bucket, CloudFront distribution, etc.) is defined using Terraform for reproducible deployment. This report details the system architecture, technologies, implementation, and results of the Eventique project, highlighting how serverless components and infrastructure-as-code were used to create a scalable, reliable event management platform.

**Table of Contents**

* [Abstract](https://chatgpt.com/c/67caf5e0-2a28-800d-9875-4e28882bb3ae#abstract)
* [Introduction](https://chatgpt.com/c/67caf5e0-2a28-800d-9875-4e28882bb3ae#introduction)
* [System Architecture](https://chatgpt.com/c/67caf5e0-2a28-800d-9875-4e28882bb3ae#system-architecture)
* [Technologies and Tools Used](https://chatgpt.com/c/67caf5e0-2a28-800d-9875-4e28882bb3ae#technologies-and-tools-used)
* [Implementation Details](https://chatgpt.com/c/67caf5e0-2a28-800d-9875-4e28882bb3ae#implementation-details)
* [Infrastructure as Code using Terraform](https://chatgpt.com/c/67caf5e0-2a28-800d-9875-4e28882bb3ae#infrastructure-as-code-using-terraform)
* [Results and Screenshots](https://chatgpt.com/c/67caf5e0-2a28-800d-9875-4e28882bb3ae#results-and-screenshots)
* [Challenges Faced](https://chatgpt.com/c/67caf5e0-2a28-800d-9875-4e28882bb3ae#challenges-faced)
* [Conclusion](https://chatgpt.com/c/67caf5e0-2a28-800d-9875-4e28882bb3ae#conclusion)
* [References](https://chatgpt.com/c/67caf5e0-2a28-800d-9875-4e28882bb3ae#references)

**Introduction**

Eventique addresses the common need for an online event management system that is easy to use and scales automatically. In Eventique, users can create accounts, log in securely, browse a list of upcoming events, and register for events of interest. The platform also allows users to view and manage the events for which they have registered. By adopting a serverless architecture on AWS, Eventique minimizes operational overhead while scaling to handle many users and events. React is used on the frontend to create an interactive single-page application, while the backend logic (event queries, registrations, etc.) is handled by AWS Lambda functions exposed through Amazon API Gateway. User identities and sessions are managed by Amazon Cognito, providing secure sign-up/sign-in and token-based access. This combination of technologies ensures a responsive user experience and secure, reliable handling of data. The following sections discuss the system design and technologies in more detail, explaining how the components work together to implement the required features.

**System Architecture**

Eventique follows a typical serverless web architecture. The static frontend assets (HTML, CSS, JavaScript) of the React single-page application are stored in an Amazon S3 bucket and delivered globally through an Amazon CloudFront CDN for low-latency access. When the user opens the web app in their browser, the React application runs entirely client-side. Any dynamic operations (such as loading event data or registering for an event) are done by calling RESTful API endpoints. These endpoints are implemented using Amazon API Gateway, which routes incoming HTTP requests to AWS Lambda functions. For example, a GET request to /events invokes a Lambda function that queries a DynamoDB table of events and returns the list. A POST request to /register invokes another Lambda function that records the user’s registration in DynamoDB.

User authentication and authorization are managed by Amazon Cognito user pools. When a user signs in (through a login form on the React app), the credentials are sent to Cognito, which validates them and returns a JSON Web Token (JWT) if successful. This JWT is stored in the client (e.g. in local storage) and attached to subsequent API requests. API Gateway is configured to validate the token, so only authenticated requests reach the backend Lambdas. This ensures that only logged-in users can perform actions like registering for events.

Each Lambda function operates with its own minimal IAM role, granting it access only to the specific DynamoDB table or other resources it needs. In this way, the data tier is a set of DynamoDB tables: one table holds the event listings (event details, dates, etc.) and another table tracks user registrations (e.g. mapping user IDs to event IDs). By using this pattern, the architecture cleanly separates concerns: the *front end* is a static React app (scaled by S3 and CloudFront), the *API layer* is API Gateway + Lambda, and the *data layer* is DynamoDB. AWS automatically handles scaling and resource management for each service, allowing the application to handle large numbers of users and events without manual provisioning.

**Technologies and Tools Used**

* **React (Frontend)** – A popular JavaScript library for building interactive user interfaces. The Eventique frontend is a single-page application built with React, which dynamically renders event lists and forms in the browser and communicates with the backend via REST APIs.
* **Amazon S3 & CloudFront (Static Hosting)** – AWS S3 provides object storage for the React app’s compiled assets. CloudFront (a CDN) is configured to cache and deliver these assets globally, reducing load times for users around the world.
* **Amazon Cognito (Authentication)** – A user identity service that provides sign-up, sign-in, and access control. Cognito user pools store the user accounts; upon login, Cognito issues JSON Web Tokens (JWTs) to the client, enabling secure API calls.
* **Amazon API Gateway (API Layer)** – A fully managed service to create and publish APIs. API Gateway routes HTTP requests (from the frontend) to the appropriate Lambda functions. It also handles authorization (verifying Cognito JWTs) and other features like throttling.
* **AWS Lambda (Backend Logic)** – A serverless compute service that runs code in response to events or API calls. In Eventique, Lambda functions implement the business logic for operations such as listing events, registering a user for an event, and retrieving a user’s registrations.
* **Amazon DynamoDB (Database)** – A serverless, NoSQL, key-value database provided by AWS. DynamoDB tables are used to store event details and user registrations. DynamoDB automatically scales and provides fast performance (single-digit millisecond latency) for read/write operations.
* **Terraform (Infrastructure as Code)** – An open-source tool by HashiCorp that enables defining cloud resources in declarative configuration files. All AWS infrastructure (Cognito user pool, API Gateway setup, Lambda functions, DynamoDB tables, S3 buckets, CloudFront distribution, IAM roles, etc.) is defined in Terraform so that the stack can be consistently deployed or updated.
* **Additional Tools:** AWS CLI/Console for managing services; Node.js runtime for writing Lambda function code; Git for version control; etc.

**Implementation Details**

**Frontend Application**

The frontend is a React single-page application. It consists of components and routes for: **Login/Signup**, **Event List**, **Event Details**, and **My Registrations**. Upon loading, the app checks if the user is authenticated by seeing if a valid Cognito JWT is stored. If not, the user is directed to a login page (or signup if they are new). The login form collects username and password and sends them to Cognito via the AWS Amplify library or AWS Cognito SDK. On successful login, Cognito returns an ID token (JWT), which the app stores locally.

Once logged in, the app fetches the list of events by calling the API Gateway endpoint (e.g., GET /events). The JWT is included in the Authorization header of each API request. The response (a JSON array of event objects) is rendered in a table or card layout. The user can click an event to see more details, and can click a “Register” button to register for that event. When registering, the frontend sends a request to the backend API (e.g., POST /events/{id}/register) with the event ID and the user token. The app also has a **My Events** page that requests GET /my-registrations from the API and displays all events the user has registered for. React state and props are used to manage UI updates; React Router is used for navigation between views.

**Backend (Serverless Logic)**

The backend consists of several AWS Lambda functions, each with a specific responsibility. Examples include:

* **GetEventsFunction:** Triggered by GET /events, this Lambda queries the DynamoDB "Events" table (using the AWS SDK) and returns a list of event records.
* **RegisterEventFunction:** Triggered by POST /events/{eventId}/register, this function reads the authenticated user’s ID (from the JWT claims) and the eventId path parameter. It then writes a new item into the "Registrations" table in DynamoDB, linking the user to that event. It might also check if the user is already registered or if the event exists.
* **GetMyEventsFunction:** Triggered by GET /my-registrations, this function reads the user ID from the token and queries the "Registrations" table for all events the user has registered, possibly performing a second lookup on the "Events" table to get event details.
* **AdminFunctions (optional):** If event creation or administration is needed, additional Lambdas (e.g., CreateEventFunction) could be implemented to add events to the DynamoDB table.

Each Lambda is written (for example) in Node.js or Python and uses the AWS SDK to interact with DynamoDB. Lambdas are configured with minimal IAM roles: for example, GetEventsFunction has read-only access to the Events table, and RegisterEventFunction has write access to the Registrations table. API Gateway is set up with routes and methods (GET, POST) and integrations that invoke these Lambdas as proxies.

All requests between API Gateway and Lambdas are stateless JSON calls, and responses are returned as JSON to the frontend. Error handling is included in the Lambdas to return appropriate HTTP status codes (e.g., 400/500). CloudWatch logs are automatically generated for each Lambda invocation to aid debugging.

**Authentication Flow**

User authentication is handled via an Amazon Cognito **User Pool**. During development, a Cognito user pool and an app client were created (via Terraform). The React app uses the AWS Amplify Auth library (or Cognito JS SDK) to communicate with Cognito. When a user signs up, Cognito saves their account in the user pool. When the user logs in, Cognito validates credentials and issues an ID token (JWT). This token includes user information (username, email, etc.) and is cryptographically signed. The frontend stores the JWT and sends it in the Authorization header (Bearer <token>) of each API call. API Gateway is configured with a Cognito authorizer, so it automatically checks the token’s validity. Only if the token is valid does the request proceed to the Lambda. This provides secure, token-based access control. As AWS documentation notes, authenticated users receive a JWT directly from Cognito, which can then be used with backend APIs. This removes the need to implement custom authentication logic and leverages AWS’s managed security features.

**Infrastructure as Code using Terraform**

All cloud resources for Eventique are defined and managed using Terraform. This ensures that the entire infrastructure can be version-controlled, reviewed, and recreated reliably. Key Terraform configurations include:

* **Provider and Backend:** The AWS provider is configured with the target region and credentials. A Terraform remote backend (e.g., S3 state bucket) can be used to store state files.
* **Cognito Resources:** Terraform resources like aws\_cognito\_user\_pool and aws\_cognito\_user\_pool\_client are used to create the user pool and app client. This defines username/password settings and callback URLs for the login process.
* **API Gateway Setup:** Resources aws\_apigatewayv2\_api, aws\_apigatewayv2\_route, and aws\_apigatewayv2\_integration (for HTTP APIs) or the equivalent REST API resources are used to define the API. For each endpoint (e.g., /events, /events/{id}/register), a route and integration to the corresponding Lambda are declared. A Cognito authorizer is attached to secure the routes.
* **Lambda Functions:** Each function is defined with aws\_lambda\_function, specifying the runtime (Node.js 14.x or Python 3.9, etc.), handler code (from a zip artifact or inline), and environment variables. An aws\_lambda\_permission is added to allow API Gateway to invoke the function. IAM roles and policies (aws\_iam\_role, aws\_iam\_policy) are created to grant the Lambda only necessary permissions (e.g., read/write on specific DynamoDB tables).
* **DynamoDB Tables:** The event and registration tables are created with aws\_dynamodb\_table. For example:
* resource "aws\_dynamodb\_table" "events" {
* name = "Events"
* billing\_mode = "PAY\_PER\_REQUEST"
* hash\_key = "EventID"
* attribute {
* name = "EventID"
* type = "S"
* }
* }

This creates a DynamoDB table named “Events” with a string primary key. A similar table is defined for registrations (e.g., with a composite key of UserID and EventID).

* **S3 Bucket and CloudFront:** The static website bucket is defined with aws\_s3\_bucket, and CloudFront distribution with aws\_cloudfront\_distribution is configured to serve the bucket content over HTTPS. Bucket policies allow CloudFront to access the objects.
* **Outputs:** Terraform outputs (optional) provide URLs or ARNs, such as the API Gateway endpoint URL and CloudFront distribution domain, making it easy to retrieve endpoints.

Using Terraform provides several benefits: infrastructure changes (e.g. adding a new table or adjusting an IAM policy) can be made in code and applied via terraform apply. The declarative configuration ensures that resources remain in the desired state, and it makes collaboration and deployment reproducible. For example, if the team adds a new Lambda or database index, they simply update the .tf files and re-apply; Terraform handles creating or updating the actual AWS resources.

**Results and Screenshots**

The deployed Eventique application successfully meets the project requirements. The frontend allows users to sign up and log in via Cognito, after which they can view the event listing and register. Below are descriptions of expected results and example UI states (no real user data is shown here):

* **User Login Page:** The React app displays a login form with fields for username and password. Upon submitting valid credentials, the user is authenticated via Cognito and redirected to the main event list. (Figure 2: *Login Screen* – shows text fields and a “Login” button.)
* **Event Listing:** After logging in, the user sees a table or list of events. Each event shows the title, date, location, and a “Register” button. This list is fetched from DynamoDB via a Lambda function. (Figure 3: *Event List* – an example list of upcoming events with register options.)
* **Registration Confirmation:** When the user clicks “Register” for an event, the frontend sends a request to the backend, and upon success, shows a confirmation message (e.g., “Registration successful”). The Lambda function adds an entry to the Registrations table.
* **My Registered Events:** The user’s personal page (“My Events”) lists all events they have registered for, by querying the registrations table. This verifies that the backend correctly linked the user to the events.

A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.

**Challenges Faced**

* **Cognito Configuration:** Setting up Amazon Cognito user pools and integrating them with API Gateway required careful attention. We had to correctly configure callback URLs for the hosted UI and ensure API Gateway had the proper Cognito authorizer. Working through Cognito’s configuration panels and IAM roles took time.
* **CORS and Authentication Tokens:** When calling the API from the React frontend, we encountered CORS (Cross-Origin Resource Sharing) issues. We had to configure the API Gateway to allow the web application’s domain and include appropriate headers. Passing the JWT token from the frontend in the Authorization header also required ensuring the frontend code attached the token correctly.
* **Terraform Dependency Ordering:** Defining resources in Terraform meant managing dependencies (for example, the Lambda function needs the API Gateway after the API is created). We learned to use depends\_on and Terraform’s implicit dependency resolution (e.g., passing resource IDs between modules) to ensure resources were created in the correct order.
* **IAM Permissions:** Granting least-privilege IAM roles to Lambdas was tricky at first. Initial tests failed due to missing permissions (e.g., a Lambda not allowed to write to DynamoDB). It required checking CloudWatch logs to identify permission errors and updating the Terraform IAM policy accordingly.
* **Local Development vs Cloud Environment:** Testing Lambdas locally versus in the AWS cloud had differences (environment variables, IAM roles). We used mocks and AWS SAM/Serverless framework for local testing, which helped validate logic before deployment.
* **Version Control of Infrastructure:** We learned the importance of managing the Terraform state file securely (e.g., in an S3 bucket with locking) so that multiple collaborators could work safely on the infrastructure code.

Overall, these challenges were resolved through documentation, AWS forums, and iterative testing. The experience of troubleshooting real cloud services deepened our understanding of serverless deployments.

**Conclusion**

In this project, we built **Eventique**, a full-stack event management web application using modern serverless technologies. By leveraging AWS services (Lambda, API Gateway, Cognito, DynamoDB, S3, CloudFront) we achieved an architecture that is highly scalable, cost-efficient, and requires minimal operational maintenance. The React frontend provides a responsive user interface, while the backend Lambdas and API Gateway deliver robust RESTful services. Terraform enabled us to define the entire infrastructure as code, ensuring consistency and ease of deployment.

The result is a working prototype where users can seamlessly authenticate, browse events, register for them, and view their registrations. Through this project, we gained practical experience in cloud architecture, user authentication flows, and infrastructure automation. Future enhancements could include adding event creation by administrators, search/filter features for events, analytics of user registrations, and improving UI design. The modular design of Eventique makes such extensions straightforward.

In summary, Eventique demonstrates how a serverless, cloud-native approach can effectively address common web application needs, and the project serves as a strong example of applying infrastructure-as-code principles in a student project.

**References**

* Amazon Web Services. *Web Application – AWS Serverless Multi-Tier Architectures (API Gateway & Lambda)* (serverless whitepaper).
* Subash Kumar. *“Build a Serverless Web Application with AWS Lambda, API Gateway, DynamoDB and Cognito”*, DEV Community blog.
* React. *“The library for web and native user interfaces.”* (GitHub README).
* Amazon Cognito Developer Guide. *What is Amazon Cognito?*.
* Amazon API Gateway. *Overview* (product page).
* Amazon DynamoDB. *Product Overview – Fast NoSQL Key-Value Database* (AWS homepage).
* HashiCorp Terraform. *What is Infrastructure as Code with Terraform?* (HashiCorp documentation).
* AWS Prescriptive Guidance. *Deploy a React-based SPA to Amazon S3 and CloudFront*.