



Ryan Dsouza

Posted on 2 Aug 2021

# Magic links ✨ with Cognito using the CDK

#cdk #lambda #cognito #typescript

This is a post on how to perform Passwordless authentication ✨ using Cognito. The resources will be created using the CDK (TypeScript).

The basic workflow in this would be:

1. User enters their email on the sign-in page.
2. We create a user in Cognito and store the secret/auth-challenge along with the creation timestamp.
3. We construct a link to our webapp with the user's email and secret and email it to them
4. Finally, when they open the sign-in link, we verify the secret + expiration and authenticate the user.

## Prerequisites

To follow along, I would suggest cloning the repo and installing the dependencies using `yarn`.

 [ryands17](#) / [passwordless-auth](#)

Allows a user to login directly via email without a need for entering passwords using Cognito

**Note:** This is a [monorepo](#) that has both the backend resources and simple UI to sign-in. This will be located in the `packages` folder under `backend` and `frontend` respectively.

## Constructs

Let's have a look at the constructs required to make this work.

### Cognito User Pool and App Client

We need to create a Cognito User Pool and App Client that will help us register the users and execute our sign in flow.

```
// packages/backend/lib/passwordless-login-stack.ts
```

```
import * as cg from '@aws-cdk/aws-cognito'
```

```
const userPool = new cg.UserPool(this, 'users', {
  standardAttributes: { email: { required: true, mutable: true } },
  customAttributes: {
    authChallenge: new cg.StringAttribute({ mutable: true }),
  },
  passwordPolicy: {
    requireDigits: false,
    requireUppercase: false,
    requireSymbols: false,
  },
  accountRecovery: cg.AccountRecovery.NONE,
  selfSignUpEnabled: true,
  signInAliases: { email: true },
  lambdaTriggers: {
    preSignUp: lambda(this, 'preSignup'),
    createAuthChallenge: lambda(this, 'createAuthChallenge'),
    defineAuthChallenge: lambda(this, 'defineAuthChallenge'),
    verifyAuthChallengeResponse: lambda(this, 'verifyAuthChallenge'),
    postAuthentication,
  },
  removalPolicy: cdk.RemovalPolicy.DESTROY,
})

const webClient = userPool.addClient('webAppClient', {
  authFlows: { custom: true },
})
```

First, we create a *UserPool*. This will have a standard attribute `email` that we will use for signing up and so is required. We have also specified a custom attribute that will be used to store the authentication challenge code along with the current timestamp.

We disable all password policies because we will not be using it. We also disable account recovery and set `email` as an alias so signing in can be done via the `email`. There are some `lambdaTriggers` that we will be discussing in the next section.

For sending emails, you need to [verify your email address in SES](#) as we are using SES in Sandbox mode.

We then create a `client` where we enable a custom auth flow. This is needed because we won't be using a username/password flow but sending a sign-in link which is technically a custom authentication flow in Cognito.

## Lambda triggers

Let's have a look at all the triggers used in the `lambdaTriggers` section:

- `preSignUp`

This function will be called when we signup the user after they enter their email.

```
// packages/backend/functions/preSignup.ts

export const handler: PreSignUpTriggerHandler = async (event) => {
  event.response.autoConfirmUser = true
  return event
}
```

What we do here is confirm the user so that we don't need to verify them on signup. Cognito's default flow needs a user to be verified on signup and that would defeat the purpose of a link based sign-in workflow.

- `createAuthChallenge`

This function will create the challenge and set the public and private challenge parameters respectively.

```
// packages/backend/functions/createAuthChallenge.ts

export const handler: CreateAuthChallengeTriggerHandler = async (event) => {
  // This is sent back to the client app
  event.response.publicChallengeParameters = {
    email: event.request.userAttributes.email,
  }

  // Add the secret login code to the private challenge parameters
  // so it can be verified by the "Verify Auth Challenge Response" trigger
  event.response.privateChallengeParameters = {
    challenge: event.request.userAttributes['custom:authChallenge'],
  }

  return event
}
```

The public challenge parameter i.e. the `email` will be sent to the user whereas the private challenge parameter i.e. our secret will be added to the event which will be used to verify from the sign-in link later on.

- `defineAuthChallenge`

This function checks if the parameters for our Passwordless auth (Cognito's custom challenge) are correct.

**Note:** I haven't included the functions used here to keep the code concise but you can easily view them [here](#) or in the file mentioned in the comments if you have cloned the repo :)

```
// packages/backend/functions/defineAuthChallenge.ts
```

```
export const handler: DefineAuthChallengeTriggerHandler = async (event) => {
  if (notCustomChallenge(event)) {
    // We only accept custom challenges; fail auth
    event.response.issueTokens = false
    event.response.failAuthentication = true
  } else if (tooManyFailedAttempts(event)) {
    // The user provided a wrong answer 3 times; fail auth
    event.response.issueTokens = false
    event.response.failAuthentication = true
  } else if (successfulAnswer(event)) {
    // The user provided the right answer; succeed auth
    event.response.issueTokens = true
    event.response.failAuthentication = false
  } else {
    // The user did not provide a correct answer yet; present challenge
    event.response.issueTokens = false
    event.response.failAuthentication = false
    event.response.challengeName = 'CUSTOM_CHALLENGE'
  }

  return event
}
```

This function checks for four things:

1. If the challenge was not a custom challenge, then fail the authentication process as our sign-in link only works with a custom challenge.
2. If the user has already made three attempts at the incorrect answer i.e. if the link was tampered with, fail the auth.
3. If the answer was successful, issue the tokens which means that the user will login successfully.
4. If all of the above conditions don't match, it means that the user hasn't got the link yet so set the challenge and wait for the user to click the sign-in link.

Think of this as the pre-validation function. It will check the parameters before we validate the link and return `event` with the appropriate values.

- `verifyAuthChallengeResponse`

This is the main function that verifies the link opened by the user. We call the `sendCustomChallengeAnswer` method from the [Amplify JS library](#) that would internally call this Lambda.

```
// packages/backend/functions/verifyAuthChallenge.ts
```

```
const MAGIC_LINK_TIMEOUT = 3 * 60 * 1000
```

```

export const handler: VerifyAuthChallengeResponseTriggerHandler = async (
  event
) => {
  const [authChallenge, timestamp] = (
    event.request.privateChallengeParameters.challenge || ''
  ).split(',')

  // fail if any one of the parameters is missing
  if (!authChallenge || !timestamp) {
    event.response.answerCorrect = false
    return event
  }

  // is the correct challenge and is not expired
  if (
    event.request.challengeAnswer === authChallenge &&
    Date.now() <= Number(timestamp) + MAGIC_LINK_TIMEOUT
  ) {
    event.response.answerCorrect = true
    return event
  }

  event.response.answerCorrect = false
  return event
}

```

Here, we set the validity of our link to be **3 minutes**. We then check if the `authChallenge` (secret code) and timestamp are available and fail if they are missing.

In the next step, we verify if the link has the correct code (not tampered in any way) and the user has opened the link in time. If that's valid, we set the `answerCorrect` property to `true` which means that the user can successfully sign-in. If not, we will show a nice error message in the UI stating that the sign-in link was invalid.

- `postAuthentication`

This is the final function which will be called after a user has been successfully authenticated.

```

// packages/backend/functions/postAuthentication.ts

import { CognitoIdentityServiceProvider } from 'aws-sdk'

const cisp = new CognitoIdentityServiceProvider()

export const handler: PostAuthenticationTriggerHandler = async (event) => {
  if (event.request.userAttributes?.email_verified !== 'true') {
    await cisp
      .adminUpdateUserAttributes({

```

```

    UserPoolId: event.userPoolId,
    UserAttributes: [
      {
        Name: 'email_verified',
        Value: 'true',
      },
    ],
    Username: event.userName,
  })
  .promise()
}
return event
}

```

We check if the user's email is verified and if it isn't, we verify it using the `adminUpdateUserAttributes` method from `CognitoIdentityServiceProvider`.

This was the entire flow of validating the link sent to the user and signing them in without a password.

You must be wondering that we have covered all this but where do we send the email to user with the sign-in link? For this, we will be creating a `signIn` API using API Gateway and Lambda proxy which will use `SES` to send an email to the user.

## API

Let's start with creating the resources for API Gateway and its method.

```

// packages/backend/lib/passwordless-login-stack.ts

const api = new apiGw.RestApi(this, 'authApi', {
  endpointConfiguration: { types: [apiGw.EndpointType.REGIONAL] },
  defaultCorsPreflightOptions: { allowOrigins: ['*'] },
  deployOptions: { stageName: 'dev' },
})

const signInMethod = new apiGw.LambdaIntegration(signIn)
api.root.addMethod('POST', signInMethod)

```

This snippet creates a REST API with Lambda Proxy integration and a function named `signIn`. This will be added as on the `POST` method of our endpoint. We will be passing the `email` in the body of this request.

The Lambda resource would look something like this:

```

// packages/backend/lib/passwordless-login-stack.ts

const signIn = lambda(this, 'signIn')

```

```

.addEnvironment('SES_FROM_ADDRESS', process.env.SES_FROM_ADDRESS)
.addEnvironment('USER_POOL_ID', userPool.userPoolId)

signIn.addToRolePolicy(
  new iam.PolicyStatement({
    effect: iam.Effect.ALLOW,
    actions: ['ses:SendEmail'],
    resources: ['*'],
  })
)

signIn.addToRolePolicy(
  new iam.PolicyStatement({
    effect: iam.Effect.ALLOW,
    actions: ['cognito-idp:AdminUpdateUserAttributes'],
    resources: [userPool.userPoolArn],
  })
)

```

Here, we are creating a function that needs two environment variables, one for sending the email and the other for updating the `authChallenge` (secret) in our user's custom attribute.

Due to these operations, we would also need to add permissions for SES and Cognito. The permissions `sendEmail` and `AdminUpdateUserAttributes` are for sending the email and updating the custom attribute respectively.

Now let's have a look at what the Lambda function contains:

```

// packages/backend/functions/signIn.ts

const cisp = new CognitoIdentityServiceProvider()
const ses = new SES({ region: process.env.AWS_REGION })

export const handler: APIGatewayProxyHandler = async (event) => {
  try {
    const { email } = JSON.parse(event.body || '{}')
    if (!email) throw Error()

    // set the code in custom attributes
    const authChallenge = randomDigits(6).join('')
    await cisp
      .adminUpdateUserAttributes({
        UserAttributes: [
          {
            Name: 'custom:authChallenge',
            Value: `${authChallenge},${Date.now()}`,
          },
        ],
        UserPoolId: process.env.USER_POOL_ID,
        Username: email,
      })
  }
}

```

```

    })
    .promise()

    await sendEmail(email, authChallenge)

    return {
      statusCode: 200,
      headers: {
        'Access-Control-Allow-Origin': '*',
      },
      body: JSON.stringify({
        message: `A link has been sent to ${email}`,
      }),
    }
  } catch (e) {
    console.error(e)
    return {
      statusCode: 400,
      headers: {
        'Access-Control-Allow-Origin': '*',
      },
      body: JSON.stringify({
        message: `Couldn't process the request. Please try after some time.`,
      }),
    }
  }
}

const BASE_URL = `http://localhost:3000/verify`

async function sendEmail(emailAddress: string, authChallenge: string) {
  const MAGIC_LINK = `${BASE_URL}?email=${emailAddress}&code=${authChallenge}`

  const html = `
    <html><body>
    <p>Here's your link:</p>
    <h3>
      <a target="_blank" rel="noopener noreferrer" href="${MAGIC_LINK}">Click to sign-in</a>
    </h3>
    </body></html>
  `.trim()

  const params: SES.SendEmailRequest = {
    Destination: { ToAddresses: [emailAddress] },
    Message: {
      Body: {
        Html: {
          Charset: 'UTF-8',
          Data: html,
        },
      },
      Text: {

```



```

    Charset: 'UTF-8',
    Data: `Here's your link (copy and paste in the browser): ${MAGIC_LINK}`,
  },
},
Subject: {
  Charset: 'UTF-8',
  Data: 'Login link',
},
},
Source: process.env.SES_FROM_ADDRESS,
}
await ses.sendEmail(params).promise()
}

```

First, we create a cryptographically random secret that will be inserted in the sign-up link. We set the secret and the creation timestamp in the user's custom attribute. Finally, we call the `sendEmail` method that accepts the secret and email of the user trying to sign in.

We created a `BASE_URL` constant that will actually be the link to our webapp set to `localhost` currently for development. We then construct the magic link as follows:

```
const MAGIC_LINK = `${BASE_URL}?email=${emailAddress}&code=${authChallenge}`
```

This will append the `email` and `code` as querystring params which we be fetching in our webapp to validate the sign-in request. The final part is just constructing the email to be sent where we embed the link and the user can directly click the link to open it in the browser.

## Deploying the stack

The final step to add in our stack is outputs that our webapp will use to perform authentication against the User Pool. This is done using the [Amplify JS library](#).

```

new cdk.CfnOutput(this, 'userPoolId', {
  value: userPool.userPoolId,
})

new cdk.CfnOutput(this, 'clientId', {
  value: webClient.userPoolClientId,
})

```

We set the `userPoolId` and `clientId` to be used by Amplify. CDK **automatically outputs** the API Gateway endpoint URL so we don't need to add it explicitly. Sweet!

Now let's deploy the stack! As we are using workspaces, we need to run the `deploy` command in the backend workspace.

```
yarn workspace backend cdk deploy
```

As an alternative, you can also move to the `backend` directory and simply run `yarn cdk deploy`.

This will deploy all our resources and create a JSON file with the outputs in our `frontend/src` folder. We generate the outputs file in the `frontend/src` folder on purpose as we need to use this in our React app. This app is created using [create-react-app](#).

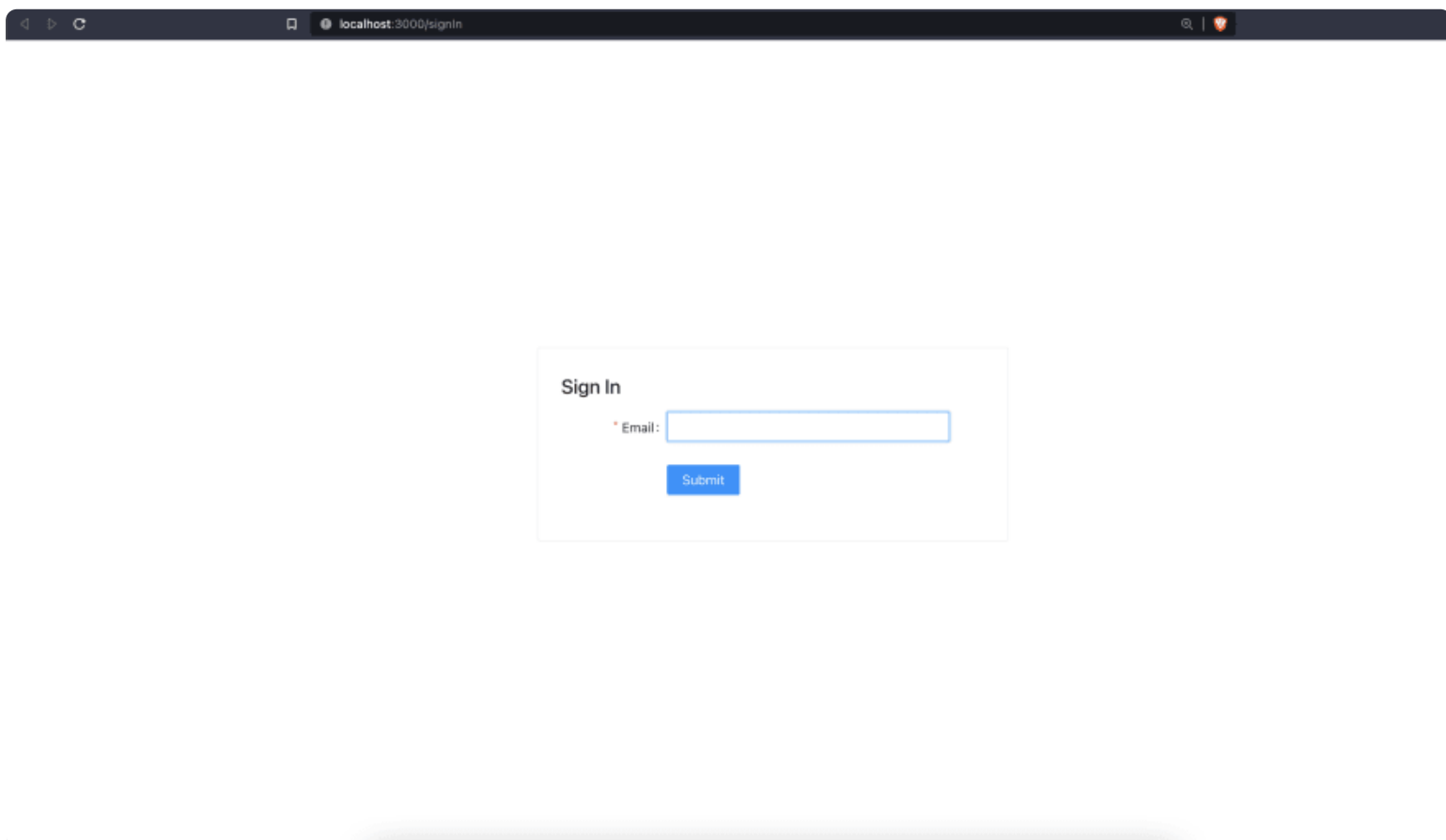
**Note:** You will need to edit the URL key the `api` file before proceeding further as CDK will generate a new resource name for your stack.

## Testing the login flow

After successful deployment, we can run the webapp locally to test the login flow using the following command:

```
yarn workspace frontend dev
```

This will fire up the app on <http://localhost:3000/> and will redirect to the `signIn` page as we aren't currently authenticated.



On adding our email address here, we will get a link on our email address.

✓ A link has been sent to [REDACTED]@gmail.com

## Sign In

\* Email : [REDACTED]@gmail.com

Submit

Here's the link I received that includes the email and authChallenge (secret).

Here's your link:

[Click to sign-in](#)

On clicking the link, a verify page is opened that will perform the sign-in and call the `sendCustomChallengeAnswer` method which will successfully verify the user and be redirected to the home page :)

## Conclusion

So that was it! This is how we can perform a truly Passwordless sign-in with Cognito. Here's the repo again for those who would like to experiment:

 [ryands17 / passwordless-auth](#)

Allows a user to login directly via email without a need for entering passwords using Cognito

Also don't forget to **destroy this stack** so that you do not incur unnecessary charges using the following command:

```
yarn workspace backend cdk destroy
```


The resources I used to help create this are:

<https://aws.amazon.com/blogs/mobile/implementing-passwordless-email-authentication-with-amazon-cognito/>

<https://schof.co/cognito-magic-links/>

Thanks for reading this and I would love to hear your thoughts on this in the comments! If you liked this post, do give it a like and share, and follow me on [Twitter](#). Until next time!

### Top comments (5)

 Vince Fulco (It / It's) • 28 Mar

...