aws

Developer Guide

AWS Blocks

AWS Blocks: Developer Guide

Table of Contents

What is AWS Blocks?

AWS Blocks is a backend toolkit for building full-stack applications on AWS. Each Block is a self-contained backend capability. It bundles your application code, a local development setup, and the infrastructure to run it. Every Block works with every other Block, so any combination produces a functioning backend. Pick the ones you need, combine them, and AWS Blocks automatically defines the infrastructure for you following AWS best practices.

Your entire application runs locally without an AWS account. When you're ready, deploy the same code to AWS without changing anything.

Why AWS Blocks

Start building in seconds

Run one command to get a working application on your machine. You don't need an AWS account until you're ready to deploy.

End-to-end type safety

Define your backend once. Your frontend types update on their own. No code generation, no type mismatches. TypeScript carries types from your database to your UI.

AI agents get it right on first attempt

Steering files ship in the npm package. They guide AI coding agents to build correct code from the start. No plugins needed.

No ceiling on what you can build

Every Block uses production AWS services. When you need more control, drop into CDK and configure resources directly. You're never stuck in an abstraction.

Compose with what you already use

Add AWS features to your existing backend. Adopt one Block at a time. A built-in migration path lets you move at your own pace.

Blocks

A Block is a module that gives you a complete feature: cloud resources, a runtime API, and a local implementation. Each Block is an npm package. You import it and use it in your backend code.

The following table lists all available Blocks.

Data and storage

Block	Description	AWS service
`KVStore`	Key-value storage with conditional writes	Amazon DynamoDB
`DistributedTable`	Structured data with indexes and queries	Amazon DynamoDB
`Database`	SQL with Kysely query builder (managed Postgres)	Amazon Aurora Serverless v2
`DistributedDatabase`	Serverless SQL with zero idle cost	Amazon Aurora DSQL
`FileBucket`	File storage with presigned URLs	Amazon S3

Authentication

Block	Description	AWS service
`AuthBasic`	Username/password with state machine API	Amazon DynamoDB + JWT
`AuthCognito`	Managed auth with MFA, groups, passkeys	Amazon Cognito
`AuthOIDC`	OIDC sign-in (Google, GitHub, Okta)	OAuth redirect flow

Compute and background

Block	Description	AWS service
`AsyncJob`	Fire-and-forget background work	Amazon SQS + AWS Lambda
`CronJob`	Scheduled task execution	Amazon EventBridge + AWS Lambda

AI

Block	Description	AWS service
`Agent`	AI agent with tools, HITL approval, conversation persistence	Amazon Bedrock
`KnowledgeBase`	Semantic document retrieval and RAG	Amazon Bedrock Knowledge Bases

Communication

Block	Description	AWS service
`Realtime`	WebSocket pub/sub channels	Amazon API Gateway WebSocket
`EmailClient`	Transactional email	Amazon SES

Configuration

Block	Description	AWS service
`AppSetting`	Configuration values and secrets	AWS Systems Manager Parameter Store

Observability

Block	Description	AWS service
`Logger`	Structured logging with correlation IDs	Amazon CloudWatch Logs
`Metrics`	Custom application metrics	Amazon CloudWatch
`Tracer`	Distributed request tracing	AWS X-Ray
`Dashboard`	Auto-generated observability dashboard	Amazon CloudWatch

Hosting and deployment

Component	Description	AWS service
`Hosting`	Frontend deployment with SSR support (CDK layer, import from `@aws-blocks/blocks/cdk`)	Amazon CloudFront + Amazon S3

For detailed API references for each Block, see *Blocks reference*.

How AWS Blocks works

AWS Blocks uses Node.js conditional exports to load different code for each context:

- **Local development**: Blocks use in-memory and filesystem storage. Your app runs on your machine.
- **CDK synthesis**: Blocks produce CDK constructs. AWS Blocks creates a CloudFormation template.
- **AWS Lambda runtime**: Blocks call AWS services through the SDK.

The same `new KVStore(scope, 'todos')` line becomes a local store in development, a DynamoDB table at deploy time, and SDK calls in production. You don't change any code.

Supported platforms

AWS Blocks supports web frameworks (Next.js, Nuxt, Astro, React, Vue, Svelte, Angular), native mobile (Swift, Kotlin, Dart/Flutter), and desktop applications. Type safety extends from your backend all the way to your client, regardless of platform.

For the full list of supported frameworks and versions, see *Supported platforms*.

Related services and tools

The following are some of the AWS services and tools that AWS Blocks works with. This is not an exhaustive list.

AWS Cloud Development Kit (AWS CDK)

AWS Blocks applications are CDK applications. The AWS CDK is an open-source framework that lets you define cloud infrastructure as code. You can use any CDK construct alongside Blocks, and you can embed AWS Blocks into an existing CDK stack. For more information, see the AWS CDK Developer Guide.

AWS Amplify

AWS Amplify is a set of tools and services for building full-stack applications. AWS Blocks and Amplify are complementary. Amplify provides hosting, CI/CD, and a managed backend experience, while AWS Blocks focuses on type-safe infrastructure-from-code with local-first development. For more information, see the Amplify documentation.

AWS Lambda

AWS Blocks deploys your backend code to AWS Lambda. For more information, see the AWS Lambda Developer Guide.

Amazon API Gateway

AWS Blocks uses API Gateway to expose your backend APIs and WebSocket channels. For more information, see the API Gateway Developer Guide.

Amazon DynamoDB

Several Blocks use DynamoDB for data persistence. For more information, see the Amazon DynamoDB Developer Guide.

Amazon Aurora

The Database and DistributedDatabase Blocks use Aurora for managed PostgreSQL. For more information, see the Amazon Aurora User Guide.

Amazon Bedrock

The Agent and KnowledgeBase Blocks use Amazon Bedrock for AI capabilities. For more information, see the Amazon Bedrock User Guide.

Next steps

- *Getting started*: Set up your environment and build your first application.
- *Concepts*: Learn about Blocks, scopes, and the IFC layer.
- *Blocks reference*: Browse all available Blocks and their APIs.

Getting started with AWS Blocks

In this tutorial, you set up your development environment, create a todo application with user authentication, data persistence, and a type-safe API. The application runs locally and can optionally deploy to AWS.

Prerequisites

To develop with AWS Blocks, you need the following on your local machine:

- **Node.js** version 22 or later. Download from https://nodejs.org/.
- **npm** version 10 or later (included with Node.js).
- A code editor with TypeScript support, such as Visual Studio Code or Kiro.

To verify your Node.js and npm versions:

```
node --version
npm --version
```

For the optional deployment step, you also need:

- AWS CLI configured with credentials
- AWS CDK bootstrapped in your account

For deployment details, see *Deploy to AWS*.

Create your AWS Blocks project

Create a new AWS Blocks application:

```
npm create @aws-blocks/blocks-app@latest my-todo-app
cd my-todo-app
npm install
```

This creates a project with the following structure:

```
my-todo-app/
```

```
### aws-blocks/
#   ### index.ts
### src/                2
#   ### app.tsx
### package.json
```

The backend: Block instantiations and API definitions.

2 The frontend: imports and calls the backend API directly.

Run the application locally

Start the development server:

```
npm run dev
```

In your web browser, navigate to `http://localhost:3000`. You see a todo application with authentication, CRUD operations, and sorting.

All Blocks are running with local implementations:

- `DistributedTable` uses in-memory storage for structured data
- `AuthBasic` uses local JWT tokens for authentication
- `ApiNamespace` routes calls through a local HTTP server

No AWS account is needed. Changes to your code are reflected immediately through hot reload.

Explore the backend code

The `aws-blocks/index.ts` file defines your backend and API in a single place:

```
import { ApiNamespace, Scope, DistributedTable, AuthBasic } from '@aws-blocks/blocks';

const scope = new Scope('todo-app');

const auth = new AuthBasic(scope, 'auth');
const todos = new DistributedTable(scope, 'todos', {
  schema: { id: 'string', title: 'string', completed: 'boolean', userId: 'string' },
  key: { partition: 'userId', sort: 'id' },
});
```

This code creates two Blocks:

- `new AuthBasic(scope, 'auth')` creates an authentication system. Locally, this uses JWT tokens. On AWS, this provisions a DynamoDB table for user records.
- `new DistributedTable(scope, 'todos', {…})` creates structured data storage. Locally, this is in-memory. On AWS, this provisions a DynamoDB table with indexes.

The API methods use these Blocks:

```
export const api = new ApiNamespace(scope, 'api', (context) => ({
  async createTodo(title: string) {
    const user = await auth.getCurrentUser(context);
    const id = crypto.randomUUID();
    await todos.put({ id, title, completed: false, userId: user.userId });
    return { id, title, completed: false };
  },

  async listTodos() {
    const user = await auth.getCurrentUser(context);
    const results = [];
    for await (const item of todos.query({ where: { userId: user.userId } })) {
      results.push(item);
    }
    return results;
  },

  async toggleTodo(id: string) {
    const user = await auth.getCurrentUser(context);
    const todo = await todos.get({ userId: user.userId, id });
    if (!todo) throw new Error('Todo not found');
    await todos.put({ ...todo, completed: !todo.completed });
    return { ...todo, completed: !todo.completed };
  },
}));

export { auth };
```

Explore the frontend code

The frontend in `src/app.tsx` imports the backend API directly:

```
import { api, auth } from '../aws-blocks/index.js';
```

There is no client generation step, no API URL configuration, and no SDK initialization. TypeScript provides full type safety. If you change a method signature in the backend, the frontend shows a compile error immediately.

Make a change

Add a new API method to `aws-blocks/index.ts` inside the `ApiNamespace` definition:

```
  async deleteTodo(id: string) {
    const user = await auth.getCurrentUser(context);
    await todos.delete({ userId: user.userId, id });
  },
```

The development server hot-reloads. You can immediately call `api.deleteTodo(id)` from the frontend with full type safety.

Available Blocks

The following table lists the Blocks used in this tutorial and other commonly used blocks:

Block	Purpose
`DistributedTable`	Structured data with indexes and queries
`AuthBasic`	Username/password authentication with JWT sessions
`ApiNamespace`	Type-safe RPC from browser to backend
`Database`	Full PostgreSQL with Kysely query builder
`FileBucket`	File uploads and downloads
`Realtime`	WebSocket pub/sub channels
`AsyncJob`	Background job processing
`Agent`	AI agent with tool calling

For the complete list of all available Blocks, see *Blocks reference*.

Optional: Deploy your application to AWS

When you're ready to deploy to a real AWS environment, see *Deploy to AWS* for the full setup and deployment steps.

Clean up

To stop the local development server, press `Ctrl+C` in your terminal.

If you deployed to AWS and want to remove all resources:

```
# Remove production deployment
npm run destroy

# Remove sandbox environment
npm run sandbox -- --destroy
```

Next steps

- *Concepts*: Learn about Blocks, scopes, the IFC layer, and how code maps to resources.
- *Blocks reference*: Explore all available Blocks and their APIs.
- *Examples*: See common application patterns and sample code.

AWS Blocks concepts

This topic explains the key concepts you need to understand when building applications with AWS Blocks.

Blocks

A **Block** is an npm package that bundles everything for a single feature: cloud resources, runtime code, and a local implementation. When you create a Block in your code, AWS Blocks sets up the matching AWS resources, wires permissions, and gives you a local implementation for development.

For example, `KVStore` is a Block for key-value storage. A single instantiation gives you:

- A DynamoDB table (provisioned automatically during deployment)
- AWS SDK integration (used at runtime in Lambda)
- A local store (used during local development)

```
import { KVStore, Scope } from '@aws-blocks/blocks';

const scope = new Scope('my-app');
const cache = new KVStore(scope, 'cache', {});

// Same API works locally and in production
await cache.set('user:123', { name: 'Alice' });
const user = await cache.get('user:123');
```

Scope

A **Scope** is a namespace container for Blocks. Every Block must be instantiated within a scope. The scope provides identity and grouping. Each Block's full identifier is derived from its scope and the ID you provide.

```
const scope = new Scope('my-app');
const users = new KVStore(scope, 'users', {});   // Full ID: my-app/users
const sessions = new KVStore(scope, 'sessions', {}); // Full ID: my-app/sessions
```

Scopes ensure that resource names are unique and predictable across your application.

> **⚠ Warning**
>
> Renaming a Block ID (the second argument to the constructor) causes the corresponding AWS resource to be deleted and recreated on the next deployment. This results in **permanent data loss** for stateful Blocks such as `KVStore`, `DistributedTable`, `Database`, and `FileBucket`. Always treat Block IDs as immutable once deployed.

The IFC layer

The **IFC (Infrastructure from Code) layer** is your backend entry point, the `aws-blocks/index.ts` file. This is where you instantiate Blocks and define your API. AWS Blocks derives your infrastructure directly from this code. You don't need separate infrastructure-as-code files.

```
// aws-blocks/index.ts - the IFC layer
import { ApiNamespace, Scope, KVStore, AuthBasic } from '@aws-blocks/blocks';

const scope = new Scope('my-app');

// Infrastructure is derived from these instantiations
const auth = new AuthBasic(scope, 'auth');
const todos = new KVStore(scope, 'todos', {});

// API methods are callable from the frontend
export const api = new ApiNamespace(scope, 'api', (context) => ({
  async createTodo(title: string) {
    const user = await auth.getCurrentUser(context);
    await todos.set(`${user.userId}:${title}`, { title, done: false });
  },
}));

export { auth };
```

The IFC layer serves three purposes simultaneously:

- **Local development**: Blocks resolve to local implementations
- **Deployment**: Blocks resolve to CDK constructs that define your infrastructure
- **Production runtime**: Blocks resolve to AWS SDK integrations running in Lambda

Conditional exports

AWS Blocks uses **Node.js conditional exports** to route the same `import` statement to different implementations depending on the execution context. This is the mechanism that makes a single codebase work across local development, deployment, and production.

When you write `import { KVStore } from '@aws-blocks/blocks'`, the resolved file depends on context:

Context	Resolved implementation	What happens
Local development	In-memory and filesystem	Your app runs entirely on localhost
CDK synthesis	CDK construct (DynamoDB table)	Infrastructure is defined for CloudFormation
Lambda runtime	AWS SDK (DynamoDB client)	Real AWS service calls in production
TypeScript/IDE	Type definitions	Full IntelliSense and type checking

You never need to configure conditional exports manually. AWS Blocks sets up the build system to select the correct implementation for each context automatically.

ApiNamespace

An **ApiNamespace** defines type-safe backend methods that your frontend can call directly. It's the bridge between your backend logic and your client code.

```
// Backend: define the API
export const api = new ApiNamespace(scope, 'api', (context) => ({
  async greet(name: string) {
    return { message: `Hello, ${name}!` };
  },
}));
```

```
// Frontend: call the API directly
import { api } from '../aws-blocks/index.js';

const result = await api.greet('World');
// result.message === 'Hello, World!'
// TypeScript knows the return type
```

There is no code generation, no API client initialization, and no URL configuration. The frontend import is type-safe. If you change the backend method signature, TypeScript reports errors in the frontend immediately.

Locally, `ApiNamespace` routes calls through a local HTTP server. In production, calls go through API Gateway to Lambda.

BlocksContext

The **BlocksContext** is the request/response object provided to `ApiNamespace` handlers. Blocks that need HTTP-level access (such as authentication blocks reading cookies or setting headers) accept `BlocksContext` as a parameter.

```
export const api = new ApiNamespace(scope, 'api', (context) => ({
  async protectedAction() {
    // Auth blocks use context to read headers/cookies
    const user = await auth.getCurrentUser(context);
    return { userId: user.userId };
  },
}));
```

You don't construct `BlocksContext` yourself. It's provided automatically by the framework for each incoming request.

The CDK layer

The **CDK layer** is an optional file (`aws-blocks/index.cdk.ts`) that gives you direct access to CDK constructs. Use it when you need to:

- Add AWS resources that don't have a Block (such as SQS queues or SNS topics)
- Configure custom domains or other environment-specific settings

- Integrate AWS Blocks into an existing CDK application

```
// aws-blocks/index.cdk.ts
import * as cdk from 'aws-cdk-lib';
import { BlocksStack } from '@aws-blocks/blocks/cdk';

const app = new cdk.App();
const stack = await BlocksStack.create(app, 'my-stack', {
  backendHandlerPath: './index.handler.ts',
  backendCDKPath: './index.ts',
});

// Add any CDK construct alongside your Blocks
const queue = new sqs.Queue(stack, 'my-queue');
queue.grantSendMessages(stack.handler);
stack.handler.addEnvironment('QUEUE_URL', queue.queueUrl);
```

The CDK layer is optional. If you don't create one, AWS Blocks generates a default CDK application from your IFC layer automatically.

Local development

When you run `npm run dev`, AWS Blocks starts your entire application locally:

- Blocks use local implementations (in-memory stores, local JWT tokens, embedded databases)
- The application runs on `http://localhost:3000` with hot reload
- No AWS account, no internet connection, and no cloud costs required

Local data persists in a `.bb-data/` directory at your project root. Each Block gets its own subdirectory.

Sandbox deployments

A **sandbox** is a fast, ephemeral deployment to AWS for testing against real services. When you run `npm run sandbox`:

- AWS Blocks deploys your backend to Lambda with hot-swapping (seconds, not minutes)
- Blocks resolve to real AWS services (DynamoDB, API Gateway, etc.)

- Each developer gets an isolated sandbox identified by a unique ID
- Remove it with `npm run sandbox:destroy`

Sandboxes are useful when you need to test behavior that differs between local implementations and real services, such as DynamoDB query performance or IAM permission boundaries.

Terminology reference

Term	Definition
Block	A self-contained npm package that bundles infrastructure, runtime, and local development code for a single capability.
Scope	A namespace container that provides identity to Blocks.
IFC layer	The backend entry point (`aws-blocks/index.ts`) where Blocks are instantiated and APIs are defined. Infrastructure is derived from this code.
CDK layer	An optional file (`aws-blocks/index.cdk.ts`) for direct CDK access and custom infrastructure.
ApiNamespace	A Block that defines type-safe RPC methods callable from the frontend.
BlocksContext	The request/response context object provided to API handlers.
Conditional exports	Node.js mechanism that routes imports to different files based on execution context.
Sandbox	A fast, ephemeral AWS deployment for testing against real services.
Local implementation	An in-memory or filesystem-based version of a Block that runs without AWS.

Supported platforms

AWS Blocks supports web, desktop, and mobile platforms. Your backend API is defined once in TypeScript, and type-safe clients are available for every supported platform.

Web frameworks

AWS Blocks auto-detects your web framework and configures the development server and deployment pipeline accordingly. The following frameworks are supported:

Framework	Minimum version	Notes
Next.js	15.x	Full SSR support with automatic route detection and middleware handling
Nuxt	4.x	SSR via Nitro adapter with automatic server routes
Astro	4.x	Static and SSR modes supported
SolidStart	1.x	Via Nitro adapter
TanStack Start	1.x	Via Nitro adapter
React (SPA)	18.x	Static single-page application, any build tool (Vite, Create React App)
Vue (SPA)	3.x	Static single-page application via Vite
Svelte (SPA)	4.x	Static single-page application via Vite or SvelteKit (static adapter)
Angular (SPA)	17.x	Static single-page application

For SPA frameworks, AWS Blocks serves static assets from CloudFront with S3 origin. For SSR frameworks (Next.js, Nuxt, Astro SSR), AWS Blocks deploys server-side rendering via Lambda compute.

Framework detection is automatic based on your `package.json` dependencies. You can override it explicitly in your Hosting configuration.

Native and mobile platforms

AWS Blocks generates typed native clients from your backend API via OpenRPC codegen. You define your API once in TypeScript, and native platform SDKs are generated automatically.

Platform	Language	Use case
iOS	Swift	Native iOS applications via Swift Package Manager
Android	Kotlin	Native Android applications via Maven/Gradle
Cross-platform mobile	Dart (Flutter)	iOS, Android, web, and desktop from a single codebase
Desktop	Kotlin, Dart	Desktop applications with type-safe backend access

How native codegen works

The build pipeline generates an OpenRPC specification (`aws-blocks/blocks.spec.json`) from your `ApiNamespace` methods. Native codegen tools read this specification to produce typed SDKs for each platform.

The codegen pipeline has two layers:

Platform SDK (hand-written, built once per platform)

Handles transport (HTTP JSON-RPC), authentication (cookie management, token injection), Realtime hydration (WebSocket connections), and FileBucket hydration (presigned URL operations).

Generated typed facade (app-specific, produced by codegen)

One typed method per `ApiNamespace` method. Calls into the platform SDK's transport layer. Contains no transport logic.

To generate the specification:

```
npx blocks-generate-spec
```

This reads `aws-blocks/index.ts` and writes `aws-blocks/blocks.spec.json`. Add it to your build script for continuous generation:

```
{
  "scripts": {
    "build:spec": "blocks-generate-spec",
    "build": "npm run build:spec && tsc && vite build"
  }
}
```

Writing APIs for native compatibility

TypeScript method signatures produce correct native types automatically. For richer types (format constraints, enums), use Zod schemas:

- Plain TypeScript types (tier 2): correct base types inferred from method signatures
- Zod schemas (tier 1): full JSON Schema with constraints (`format: uuid`, `maxLength`, `minLength`)
- If neither is detected, parameters emit as `unknown` (generates `Any` in native clients)

For best results, use explicit return types and Zod validation on complex inputs.

Hosting and deployment

AWS Blocks provides `Hosting` (import from `@aws-blocks/blocks/cdk`) for deploying your frontend alongside your backend. Hosting provisions:

- **Amazon CloudFront** distribution with global edge caching
- **Amazon S3** origin for static assets
- **Lambda compute** for server-side rendering (SSR frameworks only)
- **Optional WAF** integration for security rules
- **Custom domains** with automatic DNS and TLS

- **Skew protection** for zero-downtime deployments

Locally, your frontend runs on `http://localhost:3000` with hot reload. Hosting is deployed when you run `npm run deploy` (full deployment). Sandbox deployments (`npm run sandbox`) do not include hosting because they focus on backend hot-swapping.

Framework auto-detection

Hosting detects your framework from `package.json` dependencies:

- `next` in dependencies → Next.js adapter
- `nuxt`, `nitropack`, `@solidjs/start`, `@tanstack/start` → Nitro adapter
- `astro` → Astro adapter
- None of the above → SPA/static adapter

Override detection by setting `framework` explicitly in your Hosting configuration.

Deploy your application to AWS

After developing and testing your application locally, you can deploy it to AWS. This page covers the one-time AWS account setup and the deployment commands.

Setting up AWS credentials

If you already have AWS credentials configured, skip to the section called “Bootstrap the AWS CDK”.

Sign up for an AWS account

If you don’t already have an AWS account, complete the following steps to create one.

1. Open https://portal.aws.amazon.com/billing/signup.
2. Follow the online instructions.

   Part of the sign-up procedure involves receiving a phone call and entering a verification code on the phone keypad.

   When you sign up for an AWS account, an *AWS account root user* is created. The root user has access to all AWS services and resources in the account. As a security best practice, assign administrative access to a user, and use only the root user to perform tasks that require root user access.

Create a user with administrative access

1. Sign in to the IAM console as the account owner by choosing **Root user** and entering your AWS account email address. On the next page, enter your password.
2. Enable IAM Identity Center.

   For more information about enabling IAM Identity Center, see Enabling AWS IAM Identity Center in the *AWS IAM Identity Center User Guide*.
3. In IAM Identity Center, grant administrative access to a user.

For more information about using the IAM Identity Center directory as your identity source, see Configure user access with the default IAM Identity Center directory in the *AWS IAM Identity Center User Guide*.

> **Note**
>
> Administrative access is recommended for getting started and local development. For production environments, use a least-privilege IAM policy scoped to only the resources that AWS Blocks creates.

Install and configure the AWS CLI

1. Install the AWS CLI version 2. For more information about installing the AWS CLI, see Installing the AWS CLI.
2. Configure the AWS CLI with your credentials. For more information about configuring the AWS CLI, see Configuring the AWS CLI to use IAM Identity Center.
3. Verify your configuration by running the following command:

   ```
   aws sts get-caller-identity
   ```

   This command should return your account ID, user ID, and ARN.

Bootstrap the AWS CDK

AWS Blocks uses the AWS CDK to deploy infrastructure. You must bootstrap your AWS account before your first deployment.

Run the following command, replacing `ACCOUNT_ID` and `REGION` with your values:

```
npx cdk bootstrap aws://ACCOUNT_ID/REGION
```

For example:

```
npx cdk bootstrap aws://123456789012/us-east-1
```

You need to bootstrap only one time per account and Region combination.

Deploy to a sandbox

A sandbox is a fast, ephemeral deployment for testing against real AWS services. To deploy your application to a sandbox, run the following command:

```
npm run sandbox
```

The sandbox:

- Deploys in seconds using Lambda hot-swapping
- Uses real AWS services (DynamoDB, API Gateway, Lambda)
- Supports rapid iteration without full CloudFormation deployments
- Gives each developer an isolated environment

The same code that ran locally now runs on AWS. Blocks automatically resolve to their AWS implementations. `KVStore` becomes a DynamoDB table, `AuthBasic` provisions user storage, and `ApiNamespace` creates an API Gateway endpoint backed by a Lambda function.

To remove sandbox resources, run the following command:

```
npm run sandbox:destroy
```

Full deployment

For a complete deployment including hosting, run the following command:

```
npm run deploy
```

This runs a full CDK deployment, creating a CloudFormation stack with all your resources. Use this for staging, production, or branch deployments.

To remove all deployed resources, run the following command:

```
npm run destroy
```

Next steps

- To configure custom domains, VPC settings, or other environment-specific infrastructure, see The CDK layer in the Concepts topic.
- To integrate AWS Blocks into an existing CDK application, see Integrating with existing infrastructure.

Blocks reference

AWS Blocks ships with built-in Blocks covering authentication, data storage, real-time messaging, AI, observability, and more. Each Block provides a typed API, automatic resource provisioning, and a local implementation for offline development.

For more information about source code and advanced configuration, see AWS Blocks on GitHub.

Authentication

Blocks for user identity and session management.

- **AuthBasic**: Username/password with JWT sessions. Use for prototypes and internal tools.
- **AuthOIDC**: OIDC sign-in with Google, GitHub, Okta, or any compliant provider. Use for social login.
- **AuthCognito**: Production-grade auth with MFA, social sign-in, SAML, and passkeys. Use for production applications.

For more information, see Authentication.

Data storage

Blocks for persisting and retrieving data.

- **KVStore**: Simple key-value storage backed by DynamoDB. Use for caches, session stores, and feature flags.
- **DistributedTable**: Structured data with schema validation, secondary indexes, and rich queries. Use for entities with multiple access patterns.
- **Database**: Full PostgreSQL with Kysely query builder, migrations, transactions, and Row Level Security. Use for relational data.
- **DistributedDatabase**: Serverless SQL with scale-to-zero and multi-region writes. Use for globally distributed relational data.
- **FileBucket**: File storage for uploads and downloads with presigned URLs. Use for user-generated content.

For more information, see Data storage.

Real-time and async

Blocks for real-time communication and background processing.

- **Realtime**: Typed WebSocket pub/sub channels. Use for live updates, chat, and notifications.
- **AsyncJob**: Fire-and-forget background work via SQS and Lambda. Use for tasks that don't need immediate results.
- **CronJob**: Scheduled task execution via EventBridge. Use for periodic maintenance and batch processing.

For more information, see Real-time and async.

AI

Blocks for AI-powered features.

- **Agent**: AI agent with streaming, tool calling, HITL approval, and conversation persistence. Locally, uses a canned keyword-based provider (no real model needed). On AWS, connects to Amazon Bedrock.
- **KnowledgeBase**: Semantic document retrieval via Amazon Bedrock Knowledge Bases. Use for RAG and contextual search.

For more information, see AI.

Communication

Blocks for sending messages to users.

- **EmailClient**: Transactional email sending via Amazon SES. Locally, captures emails for testing.

For more information, see Communication.

Configuration

Blocks for application settings and secrets.

- **AppSetting**: A single configuration value or secret backed by SSM Parameter Store.

For more information, see Configuration.

Observability

Blocks for monitoring, logging, and tracing.

- **Metrics**: Custom application metrics via CloudWatch Embedded Metric Format.
- **Logger**: Structured JSON logging with levels and contextual metadata.
- **Tracer**: Distributed tracing via AWS X-Ray.
- **Dashboard**: Auto-generated observability dashboard from your metrics definitions.

For more information, see Observability.

Hosting

Frontend deployment (CDK layer only, import from `@aws-blocks/blocks/cdk`).

- **Hosting**: Frontend deployment with SSR support. Auto-detects framework (Next.js, Nuxt, Astro, SPA).

For more information, see Hosting.

Authentication

This section covers Blocks for user identity and session management, from simple username/password to federated OIDC providers.

Choosing an authentication Block

Block	Best for	Avoid when
`AuthBasic`	Prototypes, internal tools, MVPs with simple username/password auth	You need social login, MFA, or enterprise federation

Block	Best for	Avoid when
`AuthCognito`	Production apps needing social sign-in, MFA, SAML, passkeys, OAuth/OIDC	You only need simple credential-based auth (higher setup complexity)
`AuthOIDC`	Social login with Google, GitHub, Okta, or any OIDC-compliant provider	You need full user management (signup, password reset) — use AuthCognito instead

AuthBasic

Username/password authentication with JWT sessions. Handles user signup, signin, signout, password hashing (bcrypt), HTTP-only cookie sessions, and optional email-confirmed signup. Provides a `createApi()` method that generates the auth API endpoints needed by the Authenticator UI component.

Locally, AuthBasic stores user records in memory. On AWS, it provisions a DynamoDB table for user records and issues JWTs for session management. Best for prototypes, internal tools, and MVPs where you need simple credential-based auth.

For more information, see bb-auth-basic on GitHub.

AuthOIDC

OIDC sign-in with Google, GitHub, Okta, Auth0, Microsoft Entra, or any OIDC-compliant provider. Configure one or more providers with their client credentials and AuthOIDC handles the OAuth redirect flow, token exchange, and session management automatically.

Locally, AuthOIDC simulates the OAuth flow for development. On AWS, it runs a real OAuth redirect flow with session cookies. Best for applications that need social login or enterprise SSO without the complexity of Cognito.

For more information, see bb-auth-oidc on GitHub.

AuthCognito

Production-grade authentication powered by Amazon Cognito. Supports social sign-in, MFA (SMS, TOTP, email OTP), user pool groups, SAML federation, passkeys, and account recovery. Same auth

interface as AuthBasic (`requireAuth`, `getCurrentUser`, `createApi`) so you can swap providers without changing application code.

Locally, AuthCognito simulates auth flows. On AWS, it provisions a Cognito User Pool with your configured options. Best for production applications that need enterprise-grade auth features.

For more information, see bb-auth-cognito on GitHub.

Data storage

This section covers Blocks for persisting and retrieving data.

Choosing a data Block

Block	Best for	Avoid when
`KVStore`	Caches, session stores, feature flags, config values	You need queries by multiple attributes or sorting
`DistributedTable`	Entities with multiple access patterns, structured data with indexes	You need JOINs, transactions across tables, or complex SQL
`Database`	Relational data, complex queries, JOINs, transactions, migrations	Your workload is mostly key-value lookups (higher latency and cost)
`DistributedDatabase`	Globally distributed relational data, scale-to-zero workloads	You need strong consistency (uses optimistic concurrency control)
`FileBucket`	File uploads, user-generated content, large binary objects	You need structured queries over file metadata (pair with `KVStore` for metadata)

KVStore

Simple key-value storage. Store and retrieve values by key with optional conditional writes. Supports listing by prefix and scanning all entries. Best for caches, session stores, feature flags, and configuration values where you always know the key.

Locally, KVStore persists data to the filesystem in the `.bb-data/` directory. On AWS, it provisions a DynamoDB table. You can also wrap an existing DynamoDB table with `KVStore.fromExisting(tableName)`.

For more information, see bb-kv-store on GitHub.

DistributedTable

Structured data with schema validation, secondary indexes, and rich queries. Define a schema and key structure, then query by partition key, sort key conditions, or secondary indexes. Supports batch operations for up to 25 items at a time.

Best for entities with multiple access patterns where you need to query data by different attributes. Locally, DistributedTable uses in-memory storage. On AWS, it provisions a DynamoDB table with Global Secondary Indexes.

For more information, see bb-distributed-table on GitHub.

Database

Full PostgreSQL with the Kysely query builder, migrations, transactions, foreign keys, and Row Level Security. Write SQL queries with full TypeScript type inference. Supports parameterized queries, single-row lookups, and ACID transactions.

Locally, Database runs PGlite (a WebAssembly Postgres engine) in-process for zero-latency development. On AWS, it provisions Aurora Serverless v2. Use the `db pull` command to import an existing database schema.

For more information, see bb-data on GitHub.

DistributedDatabase

Serverless SQL with instant provisioning, scale-to-zero, and optional multi-region active-active writes. Same query API as Database (Kysely query builder, transactions, migrations). Transactions use optimistic concurrency control and may fail at commit under contention.

Locally, DistributedDatabase uses PGlite. On AWS, it provisions Amazon Aurora DSQL. Best for globally distributed workloads or applications that need scale-to-zero SQL without managing connection pools.

For more information, see bb-distributed-data on GitHub.

FileBucket

File storage for uploads and downloads. Store files by path, generate presigned URLs for direct browser uploads and downloads, and list files by prefix. Supports versioning for point-in-time recovery.

Locally, FileBucket stores files in the `.bb-data/` directory on your filesystem, mirroring the S3 API behavior. On AWS, it provisions an S3 bucket. You can also wrap an existing S3 bucket with `FileBucket.fromExisting(bucketName)`.

For more information, see bb-file-bucket on GitHub.

Real-time and async

This section covers Blocks for real-time communication and background processing.

Choosing a real-time or async Block

Block	Best for	Avoid when
`Realtime`	Live updates, chat, notifications, collaborative features via WebSocket	Your use case is request/response only (no persistent connections needed)
`AsyncJob`	Long-running background tasks, event-driven processing, queue-based workloads	The task completes in under a few seconds (handle it synchronously)
`CronJob`	Scheduled recurring tasks, periodic cleanup, report generation	You need immediate execution triggered by an event (use AsyncJob)

Realtime

Typed WebSocket pub/sub channels. Define namespaces with Zod schemas for type-safe message broadcasting. Publish messages from the server and subscribe from the client. Each namespace can have multiple channels for topic-based routing.

Locally, Realtime uses an in-process EventEmitter with a local WebSocket server. On AWS, it provisions API Gateway WebSocket APIs with DynamoDB for connection management. Best for live updates, chat, notifications, and collaborative features.

For more information, see bb-realtime on GitHub.

AsyncJob

Background job processing. Submit work that runs asynchronously outside the request lifecycle. Define a handler function that processes each job payload. Jobs are fire-and-forget from the caller's perspective with automatic retries on failure.

Locally, AsyncJob executes handlers in-process. On AWS, it provisions an SQS queue with a Lambda consumer. Best for tasks that don't need an immediate response: sending emails, processing uploads, generating reports.

For more information, see bb-async-job on GitHub.

CronJob

Scheduled task execution. Define a handler function and a cron expression or rate schedule. The handler runs automatically at the specified interval.

Locally, CronJob executes on schedule using Node.js timers. On AWS, it provisions an EventBridge rule that triggers a Lambda function. Best for periodic maintenance, batch processing, cleanup jobs, and report generation.

For more information, see bb-cron-job on GitHub.

AI

This section covers Blocks for AI-powered features: conversational agents and semantic search.

Choosing an AI Block

Block	Best for	Avoid when
`Agent`	Conversational AI with tool calling, streaming, and human-in-the-loop approval	You only need document search without conversation (use KnowledgeBase)

Block	Best for	Avoid when
`KnowledgeBase`	Semantic search over documents, RAG pipelines, context retrieval	You need multi-turn conversation or tool calling (use Agent, optionally with KnowledgeBase as a tool)

Agent

AI agent with streaming responses, tool calling, human-in-the-loop approval, and conversation persistence. Powered by the Strands Agents SDK. Define a system prompt and a set of tools, then stream messages to the agent. The agent can call your tools, ask for human approval before executing sensitive actions, and persist conversations across sessions.

Locally, Agent uses a canned keyword-based provider that returns predictable responses without calling any real model. No API keys or cloud costs needed. You can alternatively configure an `openai-api` provider pointing to Ollama or any OpenAI-compatible endpoint for testing with real models. On AWS, it connects to Amazon Bedrock.

For more information, see bb-agent on GitHub.

KnowledgeBase

Semantic document retrieval. Point it at a folder of documents and query with natural language to get the most relevant chunks ranked by relevance. Use it to build RAG (Retrieval-Augmented Generation) pipelines or contextual search features.

Locally, KnowledgeBase uses in-memory vector search. On AWS, it provisions Amazon Bedrock Knowledge Bases with automatic document ingestion, chunking, and embedding.

For more information, see bb-knowledge-base on GitHub.

Communication

This section covers Blocks for sending messages to users.

EmailClient

Transactional email sending. Configure a from address and send emails with text or HTML bodies. Supports batch sending for up to 50 emails in one call.

Locally, EmailClient captures sent emails and logs them to the console for development testing. On AWS, it sends via Amazon SES. Best for signup confirmations, password resets, notifications, and any transactional communication.

For more information, see bb-email-client on GitHub.

Configuration

This section covers Blocks for application settings and secrets.

AppSetting

A single configuration value or secret. Read and update values at runtime. Mark a setting as `secret: true` to store it as a SecureString. Values are typed and can be read in your API handlers without environment variable boilerplate.

Locally, AppSetting stores values in memory. On AWS, it provisions an SSM Parameter Store parameter (or SecureString for secrets). Best for API keys, feature flags, and any configuration that might change without a redeploy.

For more information, see bb-app-setting on GitHub.

Observability

This section covers Blocks for monitoring, logging, and tracing.

Choosing an observability Block

Block	Best for	Avoid when
`Metrics`	Custom application metrics with dimensions for filtering and aggregation	You only need request-level tracing (use Tracer)
`Logger`	Structured JSON logging with correlation IDs and log levels	You need numeric aggregation or alerting (use Metrics)

Block	Best for	Avoid when
`Tracer`	Distributed tracing across services, latency analysis, dependency mapping	You only need simple log output (use Logger)
`Dashboard`	Real-time operational views combining metrics, logs, and alarms	You don't have Metrics or Logger blocks yet (add those first)

Metrics

Custom application metrics. Emit named metric data points with dimensions for filtering and aggregation. Supports child emitters with merged dimensions for scoped metric contexts. Uses CloudWatch Embedded Metric Format for zero-latency metric emission.

Locally, Metrics writes to the console. On AWS, metrics flow to Amazon CloudWatch. Best for tracking business KPIs, API latencies, error rates, and custom application counters.

For more information, see bb-metrics on GitHub.

Logger

Structured JSON logging with levels, contextual metadata, and correlation IDs. Supports debug, info, warn, and error levels. Attach arbitrary metadata to log entries for filtering and analysis.

Locally, Logger writes to the console. On AWS, structured JSON flows to CloudWatch Logs with automatic correlation to API Gateway request IDs. Best for application logging with searchable structured data.

For more information, see bb-logger on GitHub.

Tracer

Distributed request tracing. Wrap operations in traced segments to visualize request flow across your application. Add searchable annotations for debugging and add metadata for detailed inspection.

Locally, Tracer is a no-op. On AWS, it integrates with AWS X-Ray for distributed tracing across Lambda, API Gateway, and downstream services. Best for debugging latency issues and understanding request paths in production.

For more information, see bb-tracer on GitHub.

Dashboard

Auto-generated observability dashboard. Define your metrics and Dashboard generates a CloudWatch dashboard automatically from your Metrics definitions. No manual widget configuration needed.

Locally, Dashboard is a no-op. On AWS, it provisions a CloudWatch Dashboard with widgets for each metric you emit. Best for teams that want out-of-the-box monitoring without manual dashboard setup.

For more information, see bb-dashboard on GitHub.

Hosting

This section covers frontend hosting and deployment.

Hosting

Frontend deployment with server-side rendering (SSR) support. Hosting auto-detects your frontend framework (Next.js, Nuxt, Astro, or SPA) and provisions the appropriate AWS resources. Import from `@aws-blocks/blocks/cdk` and use in your CDK layer file (`aws-blocks/index.cdk.ts`).

Locally, your frontend is served by the development server at `http://localhost:3000` with hot reload. On AWS, Hosting provisions a CloudFront distribution with S3 origin, Lambda compute for SSR, optional WAF, monitoring dashboards, and DNS records. Supports custom domains and skew protection for zero-downtime deployments.

Hosting is used in the CDK layer, not in the IFC layer. It is included automatically when you run `npm run deploy`. For sandbox deployments (`npm run sandbox`), hosting is not deployed because sandboxes focus on backend hot-swapping.

For more information, see hosting on GitHub.

Best practices for AWS Blocks

This topic covers best practices for structuring, securing, testing, and scaling AWS Blocks applications.

Project structure

Organize your backend code for clarity and maintainability.

Keep the IFC layer thin. Your `aws-blocks/index.ts` should contain Block instantiations and API definitions. Move complex business logic into separate modules.

```
// aws-blocks/index.ts - thin orchestration layer
import { ApiNamespace, Scope, KVStore } from '@aws-blocks/blocks';
import { createOrder, listOrders } from './orders.js';

const scope = new Scope('my-app');
const store = new KVStore(scope, 'orders', {});

export const api = new ApiNamespace(scope, 'api', (context) => ({
  createOrder: (input) => createOrder(store, context, input),
  listOrders: () => listOrders(store, context),
}));
```

```
// aws-blocks/orders.ts - business logic, testable in isolation
export async function createOrder(store, context, input) {
  // validation, business rules, persistence
}
```

Use a single Scope per application. Multiple scopes in one backend create separate resource namespaces, which adds complexity without benefit for most applications.

Co-locate related Blocks. If your app has distinct domains (orders, users, notifications), group them logically but keep them in the same IFC layer file unless the file becomes unwieldy.

Error handling

AWS Blocks propagates errors from your API methods to the frontend with the error `name` preserved. Use named errors to enable structured error handling on the client.

```
// Backend
import { isBlocksError } from '@aws-blocks/blocks';

export const api = new ApiNamespace(scope, 'api', (context) => ({
  async getItem(id: string) {
    const item = await store.get(id);
    if (!item) {
      const err = new Error(`Item ${id} not found`);
      err.name = 'NotFoundError';
      throw err;
    }
    return item;
  },
}));
```

```
// Frontend
import { api } from '../aws-blocks/index.js';
import { isBlocksError } from '@aws-blocks/blocks';

try {
  await api.getItem('abc');
} catch (err) {
  if (isBlocksError(err, 'NotFoundError')) {
    // Handle not found
  }
}
```

Best practices for errors:

- Use descriptive error names (`NotFoundError`, `ValidationError`, `UnauthorizedError`). Avoid generic `Error`.
- Never expose internal details (stack traces, AWS resource ARNs) in error messages. The `Error.cause` property stays server-side and is never sent to the client.
- Let unhandled errors bubble up. AWS Blocks returns them as generic 500 responses without leaking internals.

Authentication and authorization

Always validate the user in every API method that accesses user data. Don't rely on frontend-only checks.

```
export const api = new ApiNamespace(scope, 'api', (context) => ({
  async getMyData() {
    // Always authenticate server-side
    const user = await auth.getCurrentUser(context);
    return store.get(`user:${user.userId}`);
  },
}));
```

Scope data access by user. Use the user ID as a key prefix to prevent users from accessing each other's data:

```
// Good: scoped to user
await store.set(`${user.userId}:${itemId}`, data);

// Bad: no user scoping - any authenticated user can access any item
await store.set(itemId, data);
```

Choose the right auth Block:

- `AuthBasic`: Username/password for prototypes and internal tools.
- `AuthOIDC`: Social login with Google, GitHub, Okta, or any OIDC-compliant provider.
- `AuthCognito`: Production-ready authentication with social sign-in, MFA, SAML, passkeys, and OAuth/OIDC.

Local development

Develop locally first. Use `npm run dev` as your primary development loop. Only deploy to a sandbox when you need to test behavior specific to real AWS services.

Understand local implementation limitations. Local implementations replicate the API surface but not all production behavior:

Block	Local behavior	Production difference
`KVStore`	Local filesystem store	DynamoDB has eventual consistency for some operations, item size limits (400 KB)
`DistributedTable`	In-memory store	DynamoDB query pagination, GSI propagation delays
`Database`	PGlite (embedded Postgres)	Aurora Serverless v2 has connection limits, cold start latency
`AuthBasic`	Local JWT tokens	Tokens are not portable between local and deployed environments
`Realtime`	Local WebSocket server	API Gateway WebSocket has connection limits and message size limits

Clear local data when needed. Delete the `.bb-data/` directory to reset all local state:

```
rm -rf .bb-data
```

Testing

Unit test your business logic independently. Extract logic from API handlers into pure functions that accept Block instances as parameters.

```
// orders.ts - testable without the full framework
export async function createOrder(store: KVStore, userId: string, input: OrderInput) {
  if (!input.title) throw new Error('Title required');
  const order = { id: crypto.randomUUID(), ...input, userId };
  await store.set(`${userId}:${order.id}`, order);
  return order;
}
```

```
// orders.test.ts
import { createOrder } from './orders.js';

it('creates an order', async () => {
```

```
  const mockStore = { set: vi.fn(), get: vi.fn() };
  const result = await createOrder(mockStore, 'user-1', { title: 'Test' });
  expect(result.title).toBe('Test');
  expect(mockStore.set).toHaveBeenCalled();
});
```

Integration test locally. Run your full application with `npm run dev` and test the API endpoints. Local implementations are deterministic and fast.

End-to-end test with sandbox. For critical paths, deploy to a sandbox and run tests against real AWS services to catch behavior differences.

Performance

Minimize Block instantiations. Each Block maps to AWS resources. Don't create a new `KVStore` for every data type. Use key prefixes to partition data within a single store.

```
// Good: one store, partitioned by prefix
const store = new KVStore(scope, 'data', {});
await store.set(`users:${id}`, userData);
await store.set(`orders:${id}`, orderData);

// Avoid: separate stores for each entity (more DynamoDB tables, more cost)
const userStore = new KVStore(scope, 'users', {});
const orderStore = new KVStore(scope, 'orders', {});
```

Use `DistributedTable` for query-heavy workloads. If you need to query by multiple attributes or sort data, `DistributedTable` with indexes is more efficient than scanning a `KVStore`.

Keep API methods focused. Each `ApiNamespace` method becomes a Lambda invocation path. Avoid methods that do too many things. Split them into focused operations.

Deployment

Use sandbox for development, full deploy for production. Sandboxes use Lambda hot-swapping for speed but aren't suitable for production traffic.

Set up separate AWS accounts for development, staging, and production. AWS Blocks deploys the same code to any account. The resources are derived from your code, not from environment-specific configuration.

Configure environment-specific settings in the CDK layer. Custom domains, VPC configuration, and other environment differences belong in `aws-blocks/index.cdk.ts`, not in your runtime code.

```
// aws-blocks/index.cdk.ts
const isProd = process.env.DEPLOY_ENV === 'production';

const stack = await BlocksStack.create(app, stackName, { /* ... */ });

if (isProd) {
  // Production-only: custom domain, WAF, etc.
  new Hosting(stack, 'Hosting', {
    customDomain: 'app.example.com',
  });
}
```

Working with AI agents

AWS Blocks is designed to work well with AI coding assistants. Steering files ship in the npm package, guiding agents to produce correct architecture from the start.

To get the best results from AI coding agents:

- **Keep your IFC layer clean and readable.** AI agents use it as context to understand your application.
- **Use descriptive Block IDs.** `new KVStore(scope, 'user-sessions', {})` gives agents more context than `new KVStore(scope, 's1', {})`.
- **Write JSDoc comments on API methods.** Agents use these to understand intent when generating frontend code.
- **Install Block packages.** Their `README.md` and type definitions in `node_modules` provide agents with API documentation and usage examples.
- **Use the Agent Block for AI features.** The Agent block provides tool calling, HITL approval, and conversation persistence. Locally, it uses a canned provider for predictable testing without API keys.

Example prompt for AI agents

The following prompt demonstrates how an AI agent can add multiple capabilities in a single interaction:

```
Add a FileBucket for user avatars, an AsyncJob that generates thumbnails on upload, and
 a Realtime channel that notifies the UI when processing completes.
```

Creating custom Blocks

When the built-in Blocks don't cover your use case, you can create your own. A custom Block packages infrastructure, runtime logic, and a local implementation into a single reusable module, usable across projects or shared with your team.

Use cases for custom Blocks

Create a custom Block when:

- You're repeating the same AWS resource + SDK boilerplate across multiple API methods
- You want local development support for a resource that doesn't have a built-in Block
- You want to share a reusable capability across projects or teams

Don't create a custom Block when:

- You only need the resource in one place. Use the The CDK layer directly instead
- A built-in Block already does what you need

Block structure

A Block is an npm package with up to four exports, each targeting a different execution context:

```
my-building-block/
### src/
#   ### index.ts
#   ### index.mock.ts     2
#   ### index.cdk.ts      3
#   ### client-hook.ts    4
#   ### types.ts          5
### package.json
### tsconfig.json
### README.md
```

Runtime: AWS SDK integration, runs in Lambda.

2 **Local**: In-memory/filesystem implementation, runs during `npm run dev`.

3. **Infrastructure**: CDK constructs, runs during `cdk synth`.
4. **Client hook**: Browser-side protocol extensions (optional).
5. **Shared types**: Type definitions shared across all exports.

Configure package.json exports

The `package.json` uses conditional exports to route imports to the correct file:

```
{
  "name": "@my-org/bb-notifications",
  "version": "1.0.0",
  "type": "module",
  "exports": {
    ".": {
      "types": "./dist/types.d.ts",
      "cdk": "./dist/index.cdk.js",
      "aws-runtime": "./dist/index.js",
      "default": "./dist/index.mock.js"
    }
  },
  "files": ["dist"]
}
```

Export condition	File	When it's used
`types`	`types.d.ts`	IDE IntelliSense and TypeScript compilation
`cdk`	`index.cdk.js`	CDK synthesis (infrastructure provisioning)
`aws-runtime`	`index.js`	Lambda execution (production)
`default`	`index.mock.js`	Local development (`npm run dev`)

Define shared types

Start by defining the interface your Block exposes. Both the runtime and local implementations must conform to this interface.

```
// src/types.ts
```

```
import { Scope } from '@aws-blocks/blocks';

export interface NotificationOptions {
  /** The sender email address (must be verified in SES) */
  fromAddress: string;
}

export interface SendEmailInput {
  to: string;
  subject: string;
  body: string;
}

export declare class Notifications {
  constructor(scope: Scope, id: string, options: NotificationOptions);
  sendEmail(input: SendEmailInput): Promise<{ messageId: string }>;
}
```

Implement the runtime

The runtime implementation uses the AWS SDK to interact with real services. It runs inside Lambda in production.

```
// src/index.ts
import { SESv2Client, SendEmailCommand } from '@aws-sdk/client-sesv2';
import { Scope } from '@aws-blocks/blocks';
import type { NotificationOptions, SendEmailInput } from './types.js';

export class Notifications {
  private client: SESv2Client;
  private fromAddress: string;

  constructor(scope: Scope, id: string, options: NotificationOptions) {
    this.client = new SESv2Client({});
    this.fromAddress = options.fromAddress;
  }

  async sendEmail(input: SendEmailInput): Promise<{ messageId: string }> {
    const result = await this.client.send(new SendEmailCommand({
      FromEmailAddress: this.fromAddress,
      Destination: { ToAddresses: [input.to] },
      Content: {
```

```
        Simple: {
          Subject: { Data: input.subject },
          Body: { Text: { Data: input.body } },
        },
      },
    }));
    return { messageId: result.MessageId ?? 'unknown' };
  }
}
```

Implement the local version

The local implementation provides the same API but runs without AWS. Use in-memory data structures, the filesystem, or embedded databases.

```
// src/index.mock.ts
import { Scope, getMockDataDir } from '@aws-blocks/blocks';
import { writeFileSync, mkdirSync } from 'fs';
import { join } from 'path';
import type { NotificationOptions, SendEmailInput } from './types.js';

export class Notifications {
  private logDir: string;

  constructor(scope: Scope, id: string, options: NotificationOptions) {
    this.logDir = getMockDataDir(scope);
    mkdirSync(this.logDir, { recursive: true });
  }

  async sendEmail(input: SendEmailInput): Promise<{ messageId: string }> {
    const messageId = crypto.randomUUID();
    const logFile = join(this.logDir, `${messageId}.json`);
    writeFileSync(logFile, JSON.stringify({ ...input, messageId, sentAt: new
 Date().toISOString() }));
    console.log(`[mock] Email sent to ${input.to}: "${input.subject}"`);
    return { messageId };
  }
}
```

During local development, emails are logged to the console and saved as JSON files in `.bb-data/` instead of being sent through SES.

Implement the infrastructure

The infrastructure export defines CDK constructs that are synthesized during deployment. Grant the Lambda handler the permissions it needs.

```
// src/index.cdk.ts
import { Scope } from '@aws-blocks/blocks';
import type { NotificationOptions } from './types.js';
import * as iam from 'aws-cdk-lib/aws-iam';

export class Notifications {
  constructor(scope: Scope, id: string, options: NotificationOptions) {
    // Grant the Lambda handler permission to send emails via SES
    scope.handler.addToRolePolicy(new iam.PolicyStatement({
      actions: ['ses:SendEmail'],
      resources: ['*'],
      conditions: {
        StringEquals: { 'ses:FromAddress': options.fromAddress },
      },
    }));
  }

  // No-op methods. Infrastructure doesn't execute runtime logic
  async sendEmail() { return { messageId: '' }; }
}
```

Client hook (optional)

Most Blocks don't need a client hook. Add one only if your Block requires a custom client-server protocol (such as WebSockets or streaming).

```
// src/client-hook.ts

// No client-side protocol needed for email notifications
export {};
```

If your Block does need a client hook (for example, a real-time messaging block), export a class with lifecycle methods:

```
// src/client-hook.ts (for a real-time Block)
```

```
export class RealtimeClientHook {
  private ws?: WebSocket;

  async onInit(config: { websocketUrl: string }) {
    this.ws = new WebSocket(config.websocketUrl);
  }

  subscribe(channel: string, handler: (msg: any) => void) {
    this.ws?.addEventListener('message', (event) => {
      const data = JSON.parse(event.data);
      if (data.channel === channel) handler(data.message);
    });
  }
}
```

Use your Block

After building your package, install it in your project and use it like any built-in Block:

```
// aws-blocks/index.ts
import { ApiNamespace, Scope } from '@aws-blocks/blocks';
import { Notifications } from '@my-org/bb-notifications';

const scope = new Scope('my-app');
const email = new Notifications(scope, 'email', {
  fromAddress: 'noreply@example.com',
});

export const api = new ApiNamespace(scope, 'api', (context) => ({
  async sendWelcomeEmail(userEmail: string) {
    return email.sendEmail({
      to: userEmail,
      subject: 'Welcome!',
      body: 'Thanks for signing up.',
    });
  },
}));
```

Testing your Block

Test both the mock and runtime implementations:

- **Mock tests**: Run directly with a test runner (Vitest, Jest). Verify that the mock behaves correctly and returns expected shapes.
- **Runtime tests**: Deploy to a sandbox and verify against real AWS services.
- **Interface parity**: Ensure both implementations accept the same inputs and return the same output shapes.

```
// tests/notifications.test.ts
import { describe, it, expect } from 'vitest';
import { Notifications } from '../src/index.mock.js';

describe('Notifications (mock)', () => {
  it('returns a messageId', async () => {
    const scope = new MockScope('test');
    const notif = new Notifications(scope, 'test-email', {
      fromAddress: 'test@example.com',
    });

    const result = await notif.sendEmail({
      to: 'user@example.com',
      subject: 'Test',
      body: 'Hello',
    });

    expect(result.messageId).toBeDefined();
  });
});
```

Publishing

To share your Block:

- **Within a monorepo**: Reference it as a workspace dependency. No publishing needed.
- **Within your organization**: Publish to a private npm registry (such as AWS CodeArtifact).
- **Publicly**: Publish to the npm public registry.

Include a `README.md` with usage examples, API documentation, and information about the AWS resources provisioned. AI coding agents use this documentation to help developers integrate your Block.

Best practices for custom Blocks

- **Keep the interface identical** between runtime and mock. Consumers shouldn't need to know which implementation is active.
- **Document mock limitations**: If the mock doesn't replicate all production behavior (such as DynamoDB query limits), document the differences.
- **Use `getMockDataDir(scope)`** for persistent mock data. This follows the `.bb-data/{id}/` convention.
- **Minimize client hooks**: Only use them for non-HTTP protocols. Most Blocks don't need one.
- **Export `{}` for unused layers**: Don't omit files. An explicit empty export signals intent.
- **Share types in a separate file**: This ensures type consistency across all implementations.

Integrating with existing infrastructure

AWS Blocks is designed to work with existing AWS infrastructure. Whether you have an existing CDK application, pre-deployed resources, or a mix of both, you can adopt AWS Blocks incrementally without rewriting your infrastructure.

Choosing a pattern

AWS Blocks provides four patterns for integrating with existing infrastructure. Choose based on your situation:

Pattern	Effort	Use when	Local support
CDK in AWS Blocks	Low	No Block exists for the resource you need	# Manual
`fromExisting`	Lowest	A Block exists and you have a pre-deployed resource	# Automatic
Custom Block	Medium	You're repeating the same resource pattern in multiple places	# Automatic
Vendorize	Low	A first-party Block is almost right but needs CDK changes	# Automatic

Pattern 1: CDK in AWS Blocks

Use raw CDK constructs alongside Blocks. AWS Blocks gives you two shapes for this:

`BlocksStack` : A standalone CloudFormation stack containing the AWS Blocks Lambda and API Gateway. Use for new projects or when you want Blocks isolated in its own deployment unit.

`BlocksBackend` : A CDK Construct you drop into an existing stack. Use when you already have stacks and want Blocks to live alongside your other resources.

Both expose a `.handler` (the Lambda function) that you can grant permissions to and inject environment variables into.

Example: BlocksStack with an SQS queue

```
// aws-blocks/index.cdk.ts
import * as cdk from 'aws-cdk-lib';
import * as sqs from 'aws-cdk-lib/aws-sqs';
import { BlocksStack } from '@aws-blocks/blocks/cdk';

const app = new cdk.App();
const stack = await BlocksStack.create(app, 'my-app', {
  backendHandlerPath: './index.handler.ts',
  backendCDKPath: './index.ts',
});

const queue = new sqs.Queue(stack, 'work-queue');
queue.grantSendMessages(stack.handler);
stack.handler.addEnvironment('QUEUE_URL', queue.queueUrl);
```

```
// aws-blocks/index.ts
import { Scope, ApiNamespace } from '@aws-blocks/blocks';
import { SQSClient, SendMessageCommand } from '@aws-sdk/client-sqs';

const scope = new Scope('my-app');
const sqs = new SQSClient({});

export const api = new ApiNamespace(scope, 'api', (context) => ({
  async enqueue(payload: Record<string, unknown>) {
    await sqs.send(new SendMessageCommand({
      QueueUrl: process.env.QUEUE_URL!,
      MessageBody: JSON.stringify(payload),
    }));
    return { ok: true };
  },
}));
```

Example: BlocksBackend inside an existing stack

Use a static factory method because `BlocksBackend.create()` is asynchronous and constructors cannot use `await`:

```
// my-existing-stack.ts
import * as cdk from 'aws-cdk-lib';
```

```
import * as sqs from 'aws-cdk-lib/aws-sqs';
import { BlocksBackend } from '@aws-blocks/blocks/cdk';

export class MyApiStack extends cdk.Stack {
  static async create(scope: cdk.App, id: string, props?: cdk.StackProps) {
    const stack = new MyApiStack(scope, id, props);

    // Your existing resources
    const queue = new sqs.Queue(stack, 'work-queue');

    // Drop Blocks in as a Construct
    const blocks = await BlocksBackend.create(stack, 'BlocksApi', {
      backendHandlerPath: './aws-blocks/index.handler.ts',
      backendCDKPath: './aws-blocks/index.ts',
    });

    queue.grantSendMessages(blocks.handler);
    blocks.handler.addEnvironment('QUEUE_URL', queue.queueUrl);
    return stack;
  }
}
```

Tradeoffs

- # Full access to any AWS resource with plain CDK and SDK
- # No local implementation. `npm run dev` will call real AWS unless you stub the SDK yourself
- # No type-level guarantee that environment variables exist

If you find yourself repeating this pattern, consider creating a custom Block instead.

Pattern 2: fromExisting

Some Blocks can wrap a pre-deployed AWS resource. The Block provides its typed API and local implementation, but skips provisioning. You keep ownership of the resource.

Supported Blocks:

Block	Factory method	Wraps
`KVStore`	`KVStore.fromExisting(tableName)`	Existing DynamoDB table
`DistributedTable`	`DistributedTable.fromExisting(tableName)`	Existing DynamoDB table
`FileBucket`	`FileBucket.fromExisting(bucketName)`	Existing S3 bucket
`Database`	`Database.fromExisting({ … })`	Existing RDS instance
`AuthCognito`	`AuthCognito.fromExisting(userPoolId)`	Existing Cognito User Pool

Example

```
// aws-blocks/index.ts
import { Scope, KVStore, ApiNamespace } from '@aws-blocks/blocks';

const scope = new Scope('my-app');

const sessions = new KVStore(scope, 'sessions', {
  table: KVStore.fromExisting('my-legacy-sessions-table'),
});

export const api = new ApiNamespace(scope, 'api', (context) => ({
  async getSession(token: string) {
    return sessions.get(token);
  },
}));
```

Tradeoffs

- # Local development still works. `npm run dev` uses in-memory storage, not the real table

- # IAM permissions are granted automatically to the AWS Blocks Lambda
- # Same typed API as a AWS Blocks-managed resource
- # Limited to the Block's API surface. If you need features the BB doesn't expose, use Pattern 1
- # Cross-account resources require manual IAM (fall back to Pattern 1)

Pattern 3: Custom Block

When you're repeating Pattern 1 for the same resource type, wrap it in a custom Block. This gives you a typed API, a local implementation, and reusability.

See *Custom Blocks* for a complete guide to authoring Blocks.

When to use

- You have ≥2 callers using the same resource pattern
- You want `npm run dev` to work without AWS for this resource
- You want type-safe call sites instead of raw `process.env` + SDK

Pattern 4: Vendorize

If a first-party Block is *almost* right but needs CDK changes you can't get upstream quickly, you can eject its source into your project:

```
npx @aws-blocks/vendorize bb-kv-store ./packages/bb-kv-store
```

This copies the Block source into your monorepo. You now own it. Modify the CDK, runtime, or local implementation as needed.

Tradeoffs

- # Full control over the Block implementation
- # Local implementations, types, and API surface all still work
- # You're responsible for maintenance. Upstream updates require manual re-sync
- # Only use when a custom wrapping BB (Pattern 3) or upstream PR won't solve the problem

CLI reference

AWS Blocks uses standard npm scripts for all development and deployment workflows. After scaffolding a project, all commands run from your project root.

For more information about CLI commands and options, see AWS Blocks CLI documentation on GitHub.

Create a project

```
npm create @aws-blocks/blocks-app@latest <project-name> [-- --template <template>]
```

Scaffolds a new AWS Blocks project.

Option	Description
`--template default`	Minimal starter with a single API endpoint (default)
`--template demo`	Todo app with auth, KVStore, DistributedTable, and CRUD operations

Local development

```
npm run dev
```

Starts the full-stack development server with hot reload. Blocks use local in-memory and filesystem storage. No AWS account required.

- Frontend: `http://localhost:3000`
- Backend API: `http://localhost:3001`

Sandbox deployment

```
npm run sandbox
```

Deploys a fast, ephemeral backend to AWS using Lambda hot-swapping. Uses real AWS services for testing. Each developer gets an isolated sandbox.

```
npm run sandbox:destroy
```

Tears down all sandbox resources.

Production deployment

```
npm run deploy
```

Deploys the full application to AWS via CDK (CloudFormation). Use for staging and production environments.

```
npm run destroy
```

Tears down all deployed resources.

Related resources

For more information about advanced CLI options, environment variables, and CI/CD configuration, see AWS Blocks CLI reference on GitHub.

AWS Blocks CLI telemetry

The AWS Blocks CLI collects anonymized usage data to help improve the developer experience. This topic explains what data is collected, how it's used, and how to opt out of telemetry collection.

You can review all collected data, understand privacy protections, and choose the opt-out method that works best for your workflow. AWS Blocks CLI telemetry is enabled by default. You can opt out at any time using any of the methods described on this page.

AWS Blocks CLI telemetry overview

The AWS Blocks CLI collects anonymized usage data each time you run a command. This data cannot identify you or your organization. No personal information, source code, resource names, or AWS account identifiers are ever collected.

The data helps the AWS Blocks team understand which commands are used most frequently and identify commands that fail often. It also helps prioritize fixes, track adoption of new features and Blocks, optimize performance for common workflows, and understand which environments and tools developers use.

You can review an exhaustive list of the collected data in the following table. This list may change over time as new metrics are added. Any new data collection will follow the same privacy principles.

Data collected

The following table describes all telemetry data that the AWS Blocks CLI collects. It includes the metric name, description, reason for collection, and example values.

Metric	Description	Reason	Example
AWS Blocks Version	CLI version (x.y.z)	Collecting this helps us bisect our data by version, which can help us identify problematic versions.	`"0.21.3"`
Installation ID	Random UUID stored on local machines executing CLI commands, at `~/.blocks/telemetry/installation-id`	This helps us estimate the distinct number of users who are affected by an error. Each unique machine has its own Installation ID. The ID is generated randomly using UUID V4.	`"87c785a7-983c-41de-a106-614d5f1d7ac0"`
Project ID	Random UUID generated per project, stored in `.blocks/config.json`	This helps us understand usage patterns across different projects without identifying the project itself.	`"a5e82e34-c12f-47ee-a507-8244d8f51893"`

Metric	Description	Reason	Example
Event ID	Random UUID generated for each telemetry event	This creates a unique identifier for each event, helping us deduplicate data.	`"f76343c7-f665-4b83-ad79-01071935f935"`
Timestamp	The time (generated by the client) the data is sent to our telemetry endpoint	This helps us generate time series graphs and identify when issues occur.	`"2026-06-10T08:08:04.877Z"`
Command Name	The CLI command that was run	Along with command state, this informati on helps us understan d feature usage and prioritize improvements.	`"deploy"`, `"dev"`, `"sandbox"`
Command State	The result of the command. Possible values are SUCCESS, FAIL, ABORTED	This helps us track error rates on each CLI command and alert us to potential regressions before they are reported by customers.	`"SUCCESS"`
Duration	The length of the command execution in milliseconds	This helps us notice performance patterns and identify bottlenecks in specific commands.	`4500`
Error Code	A categorized error code, if an error occurs	We use this to identify recurring error types and prioritize fixes.	`"CDK_DEPLOY_FAILED"`
Error Phase	Which phase of execution the error occurred in	This helps us narrow down where failures happen in the command lifecycle.	`"deploy"`

Metric	Description	Reason	Example
Operating System	The OS platform being used	This helps us debug issues that only affect specific operating systems.	`"darwin"`, `"linux"`, `"win32"`
Node Version	The Node.js runtime version used in the environment	This helps us debug any Node.js-specific issues that arise and ensure compatibility.	`"22.12.0"`
CI/CD	Whether or not the CLI command is being invoked from a CI/CD environment	This helps us identify issues specific to CI/CD environments.	`false`
Package Manager	The package manager user agent string	This helps us understand the developer toolchain and debug package-manager-specific issues.	`"npm/10.8.2 node/v22.12.0"`
Agent	Our best guess at whether an AI agent is executing the command, and which one	This helps us debug issues that affect AI agent workflows.	`"amazon-q"`
AWS CDK Version	The version of the AWS CDK used under the hood	This helps us ensure CDK compatibility and bisect CDK-specific issues.	`"2.190.0"`
Framework	The frontend framework in use	This helps us prioritize framework-specific improvements.	`"nextjs"`, `"spa"`, `"static"`

Metric	Description	Reason	Example
Template	The project template name and version used with `create-blocks-app`	This helps us understand which starter templates are popular and maintain them accordingly.	`{"name": "nextjs", "version": "1.0.0"}`
Blocks	The names and versions of AWS Blocks-provided Blocks used in the project	This helps us understand Block adoption and prioritize development of popular blocks.	`[{"name": "KVStore", "version": "1.0.0"}, {"name": "AuthBasic", "version": "1.0.0"}]`
Block Count	The total number of Blocks in the project	This helps us understand project complexity and plan for scale.	3
Custom Blocks	The number of custom (user-authored) Blocks	This helps us track custom Block adoption and improve the authoring experience.	1

AWS CDK CLI telemetry

Some AWS Blocks commands (such as `sandbox`, `deploy`, and `destroy`) use the AWS CDK CLI under the hood. When this happens, the AWS CDK CLI also collects its own telemetry data on behalf of AWS CDK.

For information about AWS CDK CLI telemetry collection, see Configure AWS CDK CLI telemetry in the *AWS CDK Developer Guide*.

> **Note**
>
> When you disable AWS Blocks CLI telemetry using any of the methods described on this page, telemetry for AWS Blocks-spawned CDK CLI invocations is also disabled automatically. You do not need to separately configure the CDK CLI opt-out.

Data NOT collected

AWS Blocks does **not** collect:

- Your AWS account ID or credentials
- Source code, file contents, or resource names
- File names or directory paths
- IP addresses (not logged server-side)
- Environment variable values
- Any personally identifiable information (PII)

How to opt out of AWS Blocks CLI telemetry

You can disable telemetry using any of the following methods. If any method indicates telemetry is disabled, no data is collected.

Opt out using the CLI

You can use the AWS Blocks CLI to check and modify your telemetry settings.

To check whether telemetry is currently enabled or disabled, enter the following command:

```
npx blocks-telemetry --status
```

To disable telemetry for the current project, enter the following command:

```
npx blocks-telemetry --disable
```

To disable telemetry globally, enter the following command:

```
npx blocks-telemetry --disable --global
```

To re-enable telemetry, enter the following command:

```
npx blocks-telemetry --enable
```

Opt out using an environment variable

To disable telemetry using an environment variable, enter the following command to set `AWS_BLOCKS_DISABLE_TELEMETRY` to 1 as shown in the following example:

```
export AWS_BLOCKS_DISABLE_TELEMETRY=1
```

> **Tip**
>
> Add this environment variable to your shell startup file, such as `.bashrc` or `.zshrc`, to persist the setting across sessions.

This method works well for CI/CD pipelines. Use it to disable telemetry without modifying project files.

Opt out using a configuration file

To disable telemetry for a specific project, set `telemetry.enabled` to `false` in the project's `.blocks/config.json` file:

```
{
  "telemetry": {
    "enabled": false
  }
}
```

This disables telemetry for the current project only.

To disable telemetry globally, set the same value in `~/.blocks/config.json`:

```
{
  "telemetry": {
    "enabled": false
  }
}
```

This disables telemetry for all projects on this machine.

Send telemetry data to a local file

If you stay opted in to AWS Blocks CLI telemetry, you can send your telemetry data to a local file. You can then review the file to audit the telemetry data that we are collecting.

Use the `--telemetry-file` option on any AWS Blocks CLI command to send telemetry data to a local file:

```
--telemetry-file='path/to/local/file'
```

As an example, this deploy command sends telemetry data to a local file:

```
npm run deploy -- --telemetry-file='/tmp/blocks-telemetry.json'
```

Each line in the file is a complete JSON object representing one telemetry event.

Examples and patterns

This topic shows common application patterns you can build with AWS Blocks. Each example demonstrates how Blocks compose together to solve real-world problems.

For more information about runnable sample applications, see AWS Blocks example applications on GitHub.

CRUD API with authentication

A typical data-driven application with user authentication and scoped data access.

```
import { ApiNamespace, Scope, KVStore, AuthBasic } from '@aws-blocks/blocks';

const scope = new Scope('notes-app');
const auth = new AuthBasic(scope, 'auth');
const notes = new KVStore(scope, 'notes', {});

export const api = new ApiNamespace(scope, 'api', (context) => ({
  async create(title: string, body: string) {
    const user = await auth.getCurrentUser(context);
    const id = crypto.randomUUID();
    const note = { id, title, body, createdAt: Date.now() };
    await notes.set(`${user.userId}:${id}`, note);
    return note;
  },

  async list() {
    const user = await auth.getCurrentUser(context);
    return notes.list({ prefix: `${user.userId}:` });
  },

  async delete(id: string) {
    const user = await auth.getCurrentUser(context);
    await notes.delete(`${user.userId}:${id}`);
  },
}));

export { auth };
```

Blocks used: `AuthBasic`, `KVStore`, `ApiNamespace`

Real-time chat

A chat application with WebSocket pub/sub and message history.

```
import { ApiNamespace, Scope, KVStore, AuthBasic, Realtime } from '@aws-blocks/blocks';
import { z } from 'zod';

const scope = new Scope('chat-app');
const auth = new AuthBasic(scope, 'auth');
const messages = new KVStore(scope, 'messages', {});

const messageSchema = z.object({ id: z.string(), user: z.string(), text: z.string(),
 ts: z.number() });

const realtime = new Realtime(scope, 'realtime', {
  namespaces: { chat: Realtime.namespace(messageSchema) },
});

export const api = new ApiNamespace(scope, 'api', (context) => ({
  async sendMessage(text: string) {
    const user = await auth.getCurrentUser(context);
    const msg = { id: crypto.randomUUID(), user: user.userId, text, ts: Date.now() };
    await messages.set(`msg:${msg.id}`, msg);
    await realtime.publish('chat', 'general', msg);
    return msg;
  },

  async getHistory() {
    return messages.list({ prefix: 'msg:' });
  },
}));

export { auth, realtime };
```

Frontend integration

The frontend subscribes to real-time updates:

```
import { api, realtime, auth } from '../aws-blocks/index.js';

// Authenticate
await auth.signIn('user@example.com', 'password', context);
```

```
// Subscribe to messages
const channel = await realtime.getChannel('chat', 'general');
channel.subscribe((msg) => {
  console.log(`${msg.user}: ${msg.text}`);
});

// Send a message
await api.sendMessage('Hello, world!');

// Load history
const history = await api.getHistory();
```

Blocks used: `AuthBasic`, `KVStore`, `Realtime`, `ApiNamespace`

File uploads with metadata

An application that stores files with associated metadata.

```
import { ApiNamespace, Scope, KVStore, FileBucket, AuthBasic } from '@aws-blocks/
blocks';

const scope = new Scope('files-app');
const auth = new AuthBasic(scope, 'auth');
const files = new FileBucket(scope, 'uploads');
const metadata = new KVStore(scope, 'file-metadata', {});

export const api = new ApiNamespace(scope, 'api', (context) => ({
  async upload(name: string, data: Buffer) {
    const user = await auth.getCurrentUser(context);
    const key = `${user.userId}/${name}`;
    await files.put(key, data);
    await metadata.set(key, { name, size: data.length, uploadedAt: Date.now() });
    return { key };
  },

  async listFiles() {
    const user = await auth.getCurrentUser(context);
    return metadata.list({ prefix: `${user.userId}/` });
  },

  async download(key: string) {
    const user = await auth.getCurrentUser(context);
```

```
    if (!key.startsWith(`${user.userId}/`)) throw new Error('Forbidden');
    return files.get(key);
  },
}));

export { auth };
```

Blocks used: `AuthBasic`, `FileBucket`, `KVStore`, `ApiNamespace`

SQL-backed application

An application using relational data with joins and transactions.

```
import { ApiNamespace, Scope, Database, AuthBasic, sql } from '@aws-blocks/blocks';

const scope = new Scope('store-app');
const auth = new AuthBasic(scope, 'auth');
const db = new Database(scope, 'db');

export const api = new ApiNamespace(scope, 'api', (context) => ({
  async createOrder(items: Array<{ productId: string; qty: number }>) {
    const user = await auth.getCurrentUser(context);

    return db.transaction(async (trx) => {
      const [order] = await trx.query(sql`
        INSERT INTO orders (user_id, status, created_at)
        VALUES (${user.userId}, 'pending', NOW())
        RETURNING *
      `);

      for (const item of items) {
        await trx.execute(sql`
          INSERT INTO order_items (order_id, product_id, quantity)
          VALUES (${order.id}, ${item.productId}, ${item.qty})
        `);
      }

      return order;
    });
  },

  async getOrders() {
    const user = await auth.getCurrentUser(context);
```

```
    return db.query(sql`
      SELECT * FROM orders
      WHERE user_id = ${user.userId}
      ORDER BY created_at DESC
    `);
  },
}));

export { auth };
```

Blocks used: `AuthBasic`, `Database`, `ApiNamespace`

Common patterns

Pattern	Approach
User-scoped data	Prefix keys with `${user.userId}:` to isolate data per user
Optimistic UI	Return the created/updated object from API methods so the frontend can update immediately
Background work	Use the CDK layer to add SQS queues or EventBridge rules alongside Blocks
Multi-tenant	Use Scope IDs or key prefixes to partition data by tenant
Feature flags	Store flags in a `KVStore` and read them in API methods before executing logic

Related resources

For more information about runnable sample applications with frontend code, deployment instructions, and tests:

- AWS Blocks example applications on GitHub
- AWS Blocks project templates used by `npm create @aws-blocks/blocks-app@latest`

Security in AWS Blocks

Cloud security at AWS is the highest priority. As an AWS customer, you benefit from a data center and network architecture that is built to meet the requirements of the most security-sensitive organizations.

Security is a shared responsibility between AWS and you. The shared responsibility model describes this as security *of* the cloud and security *in* the cloud:

- **Security of the cloud** – AWS is responsible for protecting the infrastructure that runs AWS services in the AWS Cloud. AWS also provides you with services that you can use securely. Third-party auditors regularly test and verify the effectiveness of our security as part of the AWS Compliance Programs. To learn about the compliance programs that apply to AWS Blocks, see AWS Services in Scope by Compliance Program.
- **Security in the cloud** – Your responsibility is determined by the AWS service that you use. You are also responsible for other factors including the sensitivity of your data, your company's requirements, and applicable laws and regulations.

This documentation helps you understand how to apply the shared responsibility model when using AWS Blocks.

How AWS Blocks manages permissions

AWS Blocks automatically configures IAM permissions for the resources your application uses. When you instantiate a Block, the framework grants your Lambda function the minimum permissions required to interact with that resource.

For example:

- `KVStore` grants `dynamodb:GetItem`, `dynamodb:PutItem`, `dynamodb:DeleteItem`, and `dynamodb:Query` on the specific table.
- `FileBucket` grants `s3:GetObject` and `s3:PutObject` on the specific bucket.
- `Database` grants `rds-data:ExecuteStatement` on the specific Aurora cluster.

You don't need to write IAM policies manually for Block resources. However, if you add custom CDK resources in the CDK layer, you are responsible for configuring their permissions.

Authentication

AWS Blocks provides authentication Blocks that handle user identity:

AuthBasic

Stores user credentials in DynamoDB with bcrypt-hashed passwords. Issues JWT tokens for session management. Suitable for prototypes and internal tools.

AuthCognito

Integrates with Amazon Cognito for production-grade authentication. Supports social sign-in (Google, Apple, Facebook), SAML, OIDC federation, multi-factor authentication (MFA), and account recovery flows.

Best practices:

- Use `AuthCognito` for production applications.
- Always call `auth.getCurrentUser(context)` server-side in every API method that accesses user data. Never trust client-side authentication alone.
- Scope data access by user ID to prevent unauthorized cross-user access.

Data protection

AWS Blocks uses the following data protection mechanisms by default:

Encryption at rest

All data stored by Blocks is encrypted at rest using AWS-managed keys:

- `KVStore` and `DistributedTable`: DynamoDB encryption with AWS-owned keys
- `FileBucket`: S3 server-side encryption (SSE-S3)
- `Database`: Aurora storage encryption with AWS KMS

Encryption in transit

All communication between your application and AWS services uses TLS. API Gateway endpoints use HTTPS exclusively.

Local development

During local development, data is stored unencrypted in the `.bb-data/` directory on your machine. This directory should be added to `.gitignore` (which AWS Blocks does by default) and should not contain production data.

Network security

By default, AWS Blocks deploys your API behind Amazon API Gateway, which provides:

- HTTPS-only endpoints (HTTP is not supported)
- DDoS protection via AWS Shield Standard
- Request throttling and rate limiting

For additional network isolation, you can configure your Lambda function to run inside a VPC using the CDK layer:

```
// aws-blocks/index.cdk.ts
import * as ec2 from 'aws-cdk-lib/aws-ec2';

const vpc = new ec2.Vpc(stack, 'AppVpc');
stack.handler.connections.allowToAnyIpv4(ec2.Port.tcp(443));
```

API protection

Input validation

Always validate inputs in your API methods. AWS Blocks does not perform automatic input validation. Your API methods receive whatever the client sends.

```
export const api = new ApiNamespace(scope, 'api', (context) => ({
  async createItem(title: string) {
    if (!title || title.length > 200) {
      throw new Error('Title must be 1-200 characters');
    }
    // ...
  },
}));
```

Rate limiting

API Gateway provides default throttling (10,000 requests per second per account per Region). For stricter limits, configure throttling in the CDK layer.

CORS

AWS Blocks configures CORS headers automatically for your frontend origin. In production, restrict CORS to your specific domain.

Managing secrets

Don't hardcode secrets (API keys, database passwords, third-party tokens) in your source code. Use environment variables injected through the CDK layer:

```
// aws-blocks/index.cdk.ts
import * as ssm from 'aws-cdk-lib/aws-ssm';

const apiKey = ssm.StringParameter.valueForStringParameter(stack, '/my-app/api-key');
stack.handler.addEnvironment('EXTERNAL_API_KEY', apiKey);
```

```
// aws-blocks/index.ts - access at runtime
const apiKey = process.env.EXTERNAL_API_KEY;
```

For sensitive secrets, use AWS Secrets Manager instead of SSM Parameter Store.

Least privilege for deployment

The IAM user or role that deploys your AWS Blocks application needs permissions to create and manage the underlying resources (Lambda, DynamoDB, API Gateway, CloudFormation, etc.). We recommend:

- Use a dedicated deployment role with scoped permissions.
- Don't deploy with your personal administrator credentials in production.
- Use CI/CD pipelines with OIDC-based authentication (no long-lived access keys).

Troubleshooting

Local development issues

Port 3000 or 3001 already in use

Symptom: `npm run dev` fails with `EADDRINUSE` on port 3000 or 3001.

Solution: Use a custom port, or stop the process that occupies the default port:

```
# Option 1: Use a different port
npm run dev -- --port 3002

# Option 2: Find and stop the blocking process
lsof -i :3000 | grep LISTEN
kill <PID>
```

Local data is stale or corrupted

Symptom: Local development behaves unexpectedly, returns old data, or throws deserialization errors.

Solution: Delete the `.bb-data/` directory to reset all local state:

```
rm -rf .bb-data
npm run dev
```

TypeScript errors after changing the backend

Symptom: Frontend shows type errors after modifying API methods in `aws-blocks/index.ts`.

Solution: This is expected. This means type safety is working. Update your frontend code to match the new API signature. If your IDE doesn't pick up the changes, restart the TypeScript language server.

Deployment issues

CDK bootstrap required

Symptom: `npm run deploy` or `npm run sandbox` fails with "This stack uses assets, so the toolkit stack must be deployed."

Solution: Bootstrap CDK in your account and Region:

```
npx cdk bootstrap aws://ACCOUNT_ID/REGION
```

You need to do this only one time per account/Region combination.

AWS credentials expired or missing

Symptom: Deployment fails with `ExpiredTokenException` or "Unable to locate credentials."

Solution: Refresh your AWS credentials:

```
# If using IAM Identity Center
aws sso login

# Verify credentials work
aws sts get-caller-identity
```

Sandbox deployment hangs or times out

Symptom: `npm run sandbox` takes more than a few minutes or appears stuck.

Solution: This usually indicates a CloudFormation stack in a failed state. Destroy and recreate:

```
npm run sandbox:destroy
npm run sandbox
```

If `destroy` also fails, check the CloudFormation console for the stack status and manually delete if in `ROLLBACK_COMPLETE` state.

Permission denied during deployment

Symptom: Deployment fails with `AccessDenied` or "User is not authorized to perform" errors.

Solution: Your IAM user or role needs permissions to create Lambda functions, DynamoDB tables, API Gateway APIs, IAM roles, and CloudFormation stacks. Use an account with `AdministratorAccess` for initial setup, then scope down for CI/CD.

Runtime issues

Behavior differs between local and deployed

Symptom: Code works locally but fails in production, or vice versa.

Common causes:

Cause	Solution
DynamoDB item size limit (400 KB)	Reduce payload size or use `FileBucket` for large objects
DynamoDB query pagination	Handle paginated responses. Local implementations return all results in a single response
Aurora connection limits	Reduce concurrent connections or configure connection pooling
Lambda cold starts	First request after idle period is slower. Not a bug
Environment variables missing	Ensure CDK layer injects all required env vars via `addEnvironment`

API returns 500 Internal Server Error

Symptom: Frontend receives a generic 500 error without a useful message.

Solution: Unhandled exceptions in your API methods return as 500s without details (to avoid leaking internals). Check CloudWatch Logs for the Lambda function to see the full error and stack trace:

```
aws logs tail /aws/lambda/<stack-name>-handler --follow
```

Getting further help

If your issue isn't covered here:

- Check AWS Blocks GitHub Issues for known bugs and workarounds
- Search AWS Blocks GitHub Discussions for community solutions
- File a new issue with your error message, Node.js version, and steps to reproduce