with commercially reasonable methods throughout the data lifecycle, including creating, handling, transporting, and destruction.

- If you suspect a security event, incident, or breach related to this project, contact Segment Security for assistance with your investigation and communications.
- Practice modern and common-sense security for any scenario that is not explicitly stated.

Configure your development environment

You don't need to access a Segment dev environment to build an integration. You'll test it locally on your machine. Destinations are written in TypeScript. For more information about TypeScript, see TypeScript's documentation.

To work with Segment's actions repository, download and install the following:

- node
- nvm
- yarn

Fork the repository

Fork the `segmentio/action-destinations` repository, connect to NPM and Yarn, and ensure a compatible version of Node is installed.

> Action-based destinations run several workflows on pull requests, which requires that GitHub actions be enabled in the repository. To prevent workflow failure, you must enable GitHub Actions on the Actions tab of the forked repository.

Run the test suite to ensure the environment is properly configured.

```
git clone https://github.com/<your_gh_org>/action-destinations.git
cd action-destinations
npm login
yarn login
# Requires node 14.17, optionally: nvm use 14.17
yarn --ignore-engines --ignore-optional
yarn bootstrap
yarn build
yarn install
yarn test
```

Create a destination

Once you've configured your environment, you're ready to begin building your first destination. All commands, unless noted otherwise, should run from the root of the project folder. For example, `./action-destinations` Run `./bin/run --help` at any time or visit the CLI README to see a list of available commands.

Scaffold the new destination

To begin, run `./bin/run init` to scaffold the project's directory structure, and create a minimal implementation of the new destination. The initialization sets the following information:

- Integration name
- Integration slug
- Authentication template (choose one of Custom Auth, Browser Destination (experimental), Basic Auth, OAuth2 Auth, or Minimal)

After completion, the directory structure of the new destination is created at `packages/destination-actions/src/destinations/<slug>`. The `init` command does not register or deploy the integration.

Cloud Mode Destination

The `index.ts` file in this folder contains the beginnings of an Actions-based Destination. For example, a destination named `Test` using `Basic Auth` contains the following:

```
import type { DestinationDefinition } from '@segment/actions-core'
import type { Settings } from './generated-types'

const destination: DestinationDefinition<Settings> = {
  name: 'Test',
  slug: 'actions-test',
  mode: 'cloud',

  authentication: {
    scheme: 'basic',
    fields: {
      username: {
        label: 'Username',
        description: 'Your Test username',
        type: 'string',
        required: true
      },
      password: {
        label: 'password',
        description: 'Your Test password.',
        type: 'string',
        required: true
      }
    },
    testAuthentication: (request) => {
      // Return a request that tests/validates the user's credentials.
      // If you do not have a way to validate the authentication fields safely,
      // you can remove the `testAuthentication` function, though discouraged.
    }
  },

  extendRequest({ settings }) {
    return {
      username: settings.username,
      password: settings.password
    }
  },

  onDelete: async (request, { settings, payload }) => {
    // Return a request that performs a GDPR delete for the provided Segment userId or anonymousId
    // provided in the payload. If your destination does not support GDPR deletion you should not
    // implement this function and should remove it completely.
  },

  actions: {}
}

export default destination
```

Notice the `name` and `slug` properties, the `authentication` object, an `extendRequest` function that returns the username and password from settings, and an empty `actions` object.

With this minimal configuration, the destination can connect to the Segment App's user interface, and collect authentication fields. The destination does not do anything at this point, because no Actions are defined.
The `testAuthentication` function verifies the user's credentials against a service. For testing, enter `return true` in this function to continue development.
The `onDelete` function performs a GDPR delete against a service. For testing, enter `return true` in this function to continue development.

Browser (Device Mode) Destination

```
import type { Settings } from './generated-types'
import type { BrowserDestinationDefinition } from '../../lib/browser-destinations'
import { browserDestination } from '../../runtime/shim'

// Declare global to access your client
declare global {
  interface Window {
    sdkName: typeof sdkName
  }
}

// Switch from unknown to the partner SDK client types
export const destination: BrowserDestinationDefinition<Settings, unknown> = {
  name: 'BrowserExample',
  slug: 'actions-browserexample',
  mode: 'device',

  settings: {
    // Add any Segment destination settings required here
  },

  initialize: async ({ settings, analytics }, deps) => {
    await deps.loadScript('<path_to_partner_script>')
    // initialize client code here

    return window.yourSDKName
  },

  actions: {}
}

export default browserDestination(destination)
```

In Browser Destinations, no authentication is required. Instead, you must initialize your SDK with the required settings needed.

When importing your SDK, Segment recommends loading from a CDN when possible. This keeps the bundle size lower rather than directly including the SDK in Segment’s package.

Make sure to add a global declaration where you specify your SDK as a field of a Window interface so you can reference and return it in your initialize function.

Actions

Actions define what the destination can do. They instruct Segment how to send data to your destination API. For example, consider this “Post to Channel” action from a Slack destination:

```
const destination = {
  // ...other properties
  actions: {
    postToChannel: {
      // the human-friendly display name of the action
      title: 'Post to Channel',

      // the human-friendly description of the action. supports markdown
      description: 'Post a message to a Slack channel',

      // fql query to use for the subscription initially
      defaultSubscription: 'type = "track"'

      // the set of fields that are specific to this action
      fields: {
        webhookUrl: {
          label: 'Webhook URL',
          description: 'Slack webhook URL.',
          type: 'string',
          format: 'uri',
          required: true
        },
        text: {
          label: 'Message',
          description: "The text message to post to Slack. You can use [Slack's formatting syntax.](https://api.slack
.com/reference/surfaces/formatting)",
          type: 'string',
          required: true
        }
      },

      // the final logic and request to send data to the destination's API
      perform: (request, { settings, payload }) => {
        return request.post(payload.webhookUrl, {
          responseType: 'text',
          json: {
            text: payload.text
          }
        })
      }
    }
  }
}
```

Actions best practices

Actions should map to a feature in your platform. Try to keep the action atomic. The action should perform a single operation in the downstream platform.

Define and scaffold an Action

As mentioned above, actions contain the behavior and logic necessary for sending data to your platform's API.

To create the **Post to Channel** action above, begin by creating the scaffold on top of which you'll build the action. Run `./bin/run generate:action postToChannel server` to create the scaffold.

The `generate:action` command takes two arguments:

- The name of the action
- The type of action

When you create a scaffold, the CLI also imports the action to the definition of the destination, and generates empty types based on the action's fields.

Add functionality to the Action

After you've created the scaffold for the action, add logic that defines what the action does. Here, you'll define the fields that the action expects to receive, and write the code that performs the action.

Action fields

For each action or authentication scheme, you define a collection of inputs and `fields`. Input fields define what the user sees in the Action Editor within the Segment App. In an action, these fields accept input from the incoming Segment event.

The Segment CLI introspects field definitions when you run `./bin/run generate:types` to generate their TypeScript declarations. This ensures the `perform` function is strongly-typed.

Define fields following the field schema. If your editor or IDE provides good Intellisense and autocompletion, you should see the allowed properties.

As mentioned above, the `perform` function contains the code that defines what the action does.

Segment recommends that you start with a simple task, and evolve it. Get the basics working first. Add one or two fields to start, then run `./bin/run generate:types` when you change the definition of a field. Run this step manually after changes, or run `yarn types --watch` to regenerate types when a change is detected.

Write tests

Testing ensures that your destination functions the way you expect. For information on testing, see Test your destination.

This page was last modified: 12 Jan 2023