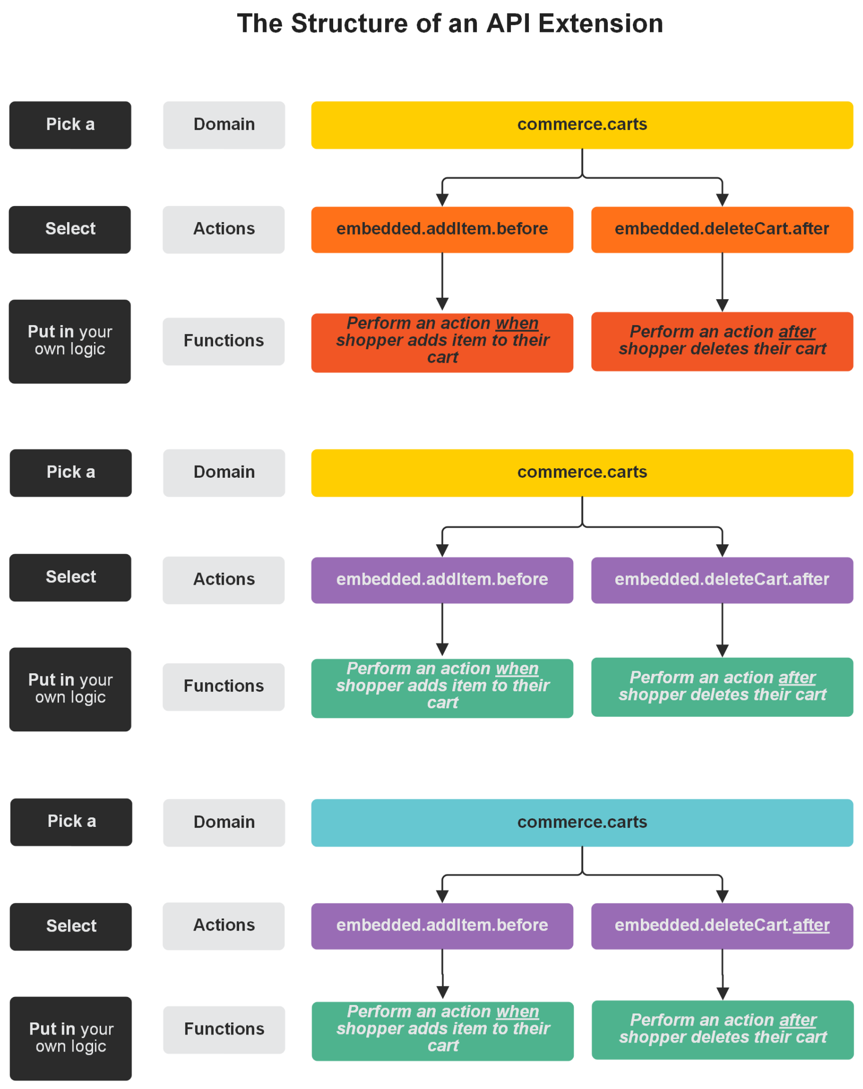
The Structure of an API Extension Application

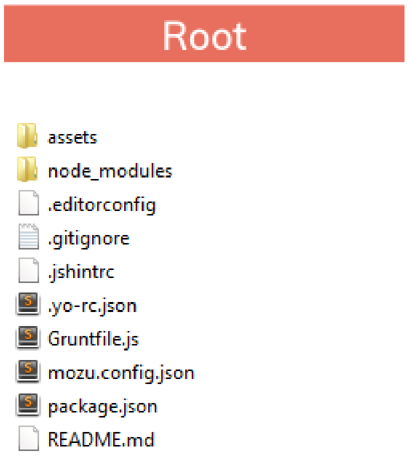
API Extension applications contain JavaScript files that run custom functions before or after a specific action occurs in Kibo. The available actions are organized into domains that relate to the component they interact with. The following diagram provides a visual demonstration of this structure. Note that while most actions run only one function, some actions can run multiple functions.



The Structure of an API Extension
Pick a
Domain
commerce.carts
Select
Actions
embedded.addItem.before
embedded.deleteCart.after
Put in your own logic
Functions
Perform an action when shopper adds item to their cart
Perform an action after shopper deletes their cart
Pick a
Domain
commerce.carts
Select
Actions
embedded.addItem.before
embedded.deleteCart.after
Put in your own logic
Functions
Perform an action when shopper adds item to their cart
Perform an action after shopper deletes their cart
Pick a
Domain
commerce.carts
Select
Actions
embedded.addItem.before
embedded.deleteCart.after
Put in your own logic
Functions
Perform an action when shopper adds item to their cart
Perform an action after shopper deletes their cart

API Extension Project Files

The project files in which you develop API Extension applications have the structure shown in the following image. You can use the Yeoman Generator to automatically scaffold these files.



Root

assets
node_modules
.editorconfig
.gitignore
.jshintrc
.yo-rc.json
Gruntfile.js
mozu.config.json
package.json
README.md

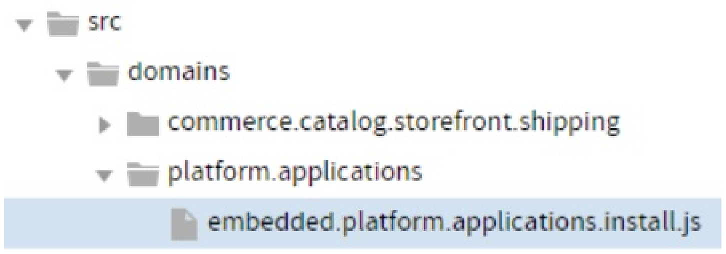
assets	This folder contains the subfolders where you code your custom functions and design your tests. When you run Grunt, all the files in the assets folder are uploaded to your application in Dev Center.

Gruntfile.js	This file defines Grunt task configurations and loads Grunt plugins. Grunt tasks automate the repetitive aspects of API Extensions development and are invoked through the command line. You can add custom Grunt tasks to this file if you wish. The default Grunt commands include: • `grunt`: Runs `grunt build`. In addition, asks for your developer account password in order to upload your assets to Dev Center. If your tests fail, you can append `--force` to the command to force the upload. • `grunt build`: Lints your code, bundles your dependencies, runs tests, and creates the `dist` folder and `functions.json` file, but does not upload the assets to Dev Center (making this command useful for development). If your tests fail, you can append `--force` to the command to force the build. • `grunt test`: Runs the tests associated with your action files (you can edit these tests in the `assets/test` directory). • `grunt reset`: Deletes all assets from your project folder and from Dev Center. • `grunt w`: Enables a watch that runs `grunt` every time you save a change to a project file. Unlike in theme development, where you can refresh a browser page to test most changes, setting a watch is normally not useful for API Extensions development, where you are more likely to test your changes across larger intervals. An exception when you might want to set a watch is if you are using API Extensions to alter view data for the theme.
mozu-config.json	This file stores authentication credentials for the application, sync app, and your Dev Center account.
package.json	This file stores npm module metadata (like the package name and version number) and includes a list of dependencies for the application (for example, Grunt dependencies, development tools, and any external Node.js libraries that the application leverages).

assets folder

dist	This folder contains built and optimized assets that result from the Grunt build task.
src	This folder contains a list of domain folders. Each domain folder contains JavaScript files that pertain to specific actions. The Grunt build task creates optimized versions of the files in the `src` folder and places them in the `dist` folder.
test	This folder contains JavaScript files where you can design simulations and tests for your application. Each file corresponds to a source code file in the `src/domains` folder.
functions.json	This file specifies how the custom functions in your JavaScript files (within the `src` folder) map to actions for your application to implement.

src folder

domains	This folder contains the JavaScript files where you code your custom functions. The files are named for the action they bind to and are organized by the domain to which the action pertains. For example, the `embedded.platform.applications.install.js` file, in the `platform.applications` domain folder, is where you code the custom function that binds to the action that occurs when the application is installed to a sandbox.  src domains commerce.catalog.storefront.shipping platform.applications embedded.platform.applications.install.js
manifest.js files	These files define the relationship between custom functions and the actions they bind to. The build task autogenerates these files based on the settings you specify in the Yeoman scaffolding tool, and then leverages the manifest files to generate the `functions.json` file.

The Structure of the Action Files

The action files are the Javascript files in which you code the custom functionality that you want to bind to actions. The naming and structure of these files follows the REST resource hierarchy.

Filename Syntax

`Type.Domain.Action.Occurrence`

Type	Identifies the type of action.
Domain	Identifies the domain of the action, which specifies what part of the API hierarchy the action interacts with. For example, `commerce.carts` or `commerce.customer`.
Action	Identifies the HTTP or action that runs the custom function. For example, `updateItem`.
Occurrence	Specifies whether the custom function runs before or after the HTTP or Kibo action occurs. For example, for an HTTP action, `before` runs the function when the HTTP request occurs in Kibo but before Kibo executes the request, and `after` runs the function after Kibo executes the request and is ready to send the HTTP response. Not every action includes this element.

Example

`embedded.commerce.orders.price.after`

This embedded action executes the custom functions associated with it after a price for an order is set.

To see the full list of actions available in API Extensions, refer to the reference documentation.