B2C Commerce Continuous Integration and Continuous Delivery Implementation

Introduction

Continuous Integration/Continuous Delivery (CI/CD) is an important aspect of the development process. This document provides solutions and guidelines for setting up a CI/CD process on Salesforce B2C Commerce and applying CI/CD to B2C Commerce and its target environments throughout the development lifecycle. We assume you are familiar with NodeJS and NPM and have a basic understanding of B2C Commerce.

What is Continuous Integration and Delivery?

CI/CD provides an automated release process for a software application. In most cases, the CI/CD process automatically pushes code into a repository. The specifics of how you implement CI/CD depends on your Git-Workflow process and your build system, but most of the CI/CD principles apply to all tools.

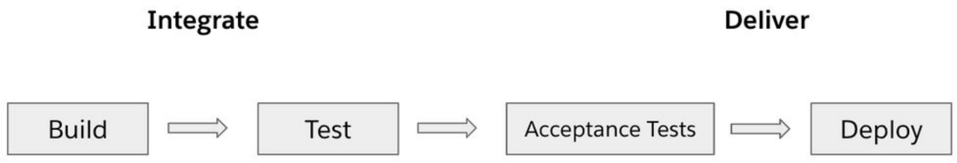
Continuous Integration

CI is the practice of managing your code in a central repository, such as Git, and using tooling and automation to provide feedback on the quality of changes. Using CI, your code is automatically compiled, built, and tested, and the results are published in a central repository where developers can review the status of each phase of the process.

Continuous Delivery

CD extends the integration process so that you can test and release your software more quickly.

Combined CI and CD Process



In this document, we combine CI/CD into one process that allows you to release software early, release software often, and release stable software.

B2C Commerce Environments

B2C Commerce provides four environments.

Sandboxes

Sandboxes are small environments where you can work without affecting other environments. Typically, developers use sandboxes to develop features independently.

Currently B2C Commerce offers two types of sandboxes:

- POD Sandboxes
    - Part of the production hardware
    - Statically provisioned
- On-Demand Sandboxes
    - Provisioned as required
    - Hosted in a public cloud environment

Development

The development environment is part of the primary instance group (PIG). The development environment is capable of receiving the full data set of the production system. Unlike sandboxes, development environments can execute scheduled jobs. For most B2C Commerce customers, the development environment serves as the main environment for integration and front-end automated tests as well as tests by QA engineers, and for UAT and acceptance tests. The development environment is an important quality gate within the CI process. We recommend that you set up continuous delivery to development to ensure you always have a deployable code base.

Staging

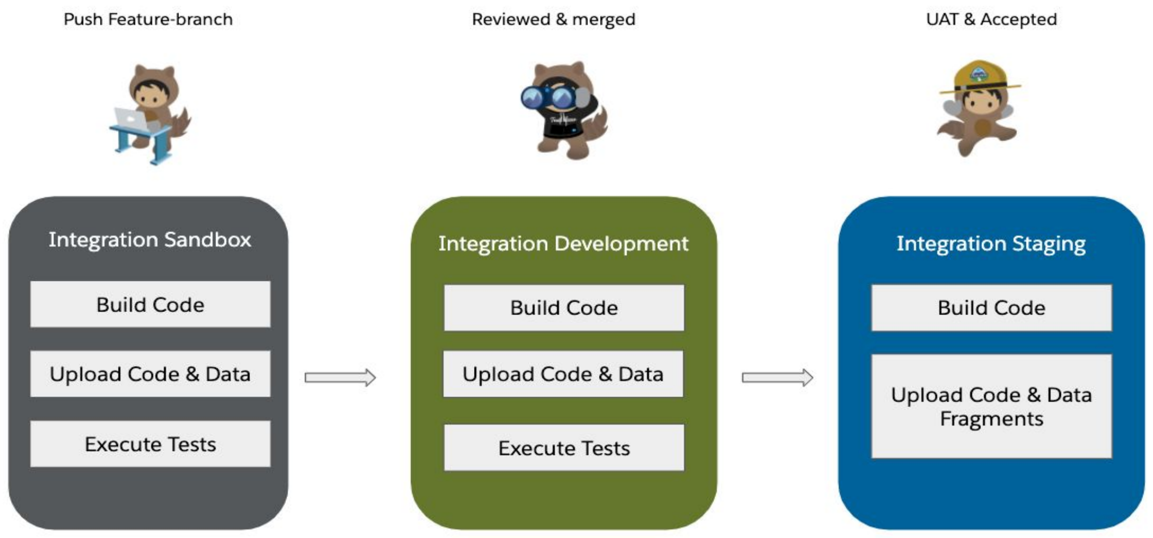
The staging environment is the master record for many kinds of data, including product, prices, content, and campaigns. Merchandisers work in the staging environment and expect it to be bug-free. Developers should ensure that the staging environment is stable at all times, since staging is the preproduction environment. Replications are allowed only from the staging environment to the development or production environments.

Production

In B2C Commerce, all data (with a few exceptions, for example, inventory) and code are replicated from staging to the production instance. To keep the environments in sync and keep the metadata on all instances the same, use a CI/CD process to avoid manual and therefore error-prone tasks.

CI/CD in B2C Commerce Environments

We have divided the B2C process into three steps that reflect the state of the software development process and the environments involved.



1. **Sandbox:** Developers work in their personal sandboxes and push a feature branch that triggers and executes the sandbox CI process. While working on features, a developer must ensure that the code is working and passes all development quality gates. This approach ensures that only code that is functional is reviewed. In large enterprise projects, where time is critical, this process saves time and pull requests discussions.

2. **Development**: After the code passes review, it's merged into the main development branch, often called the release branch. If the code passes defined quality gates, it's ready to be accepted. Quality gates in development are focused on acceptance and functional testing that ensure the code works well from an end-user perspective in the overall application.

3. **Staging:** To release code, the code must be deployed to staging, where it's replicated to production. The CI process running against the staging instance is a bit different. It must provide one-time data uploads that don't overwrite existing objects on the instance, such as a content library. Replication to production should be a manual process in nearly all cases.

Using NPM with CI/CD

NodeJS and its package manager NPM allow developers to manage all tasks through the Package-Json-File in a single place. You can use NPM as a task manager, which allows you to create new tasks easily by adding them into the package.json file

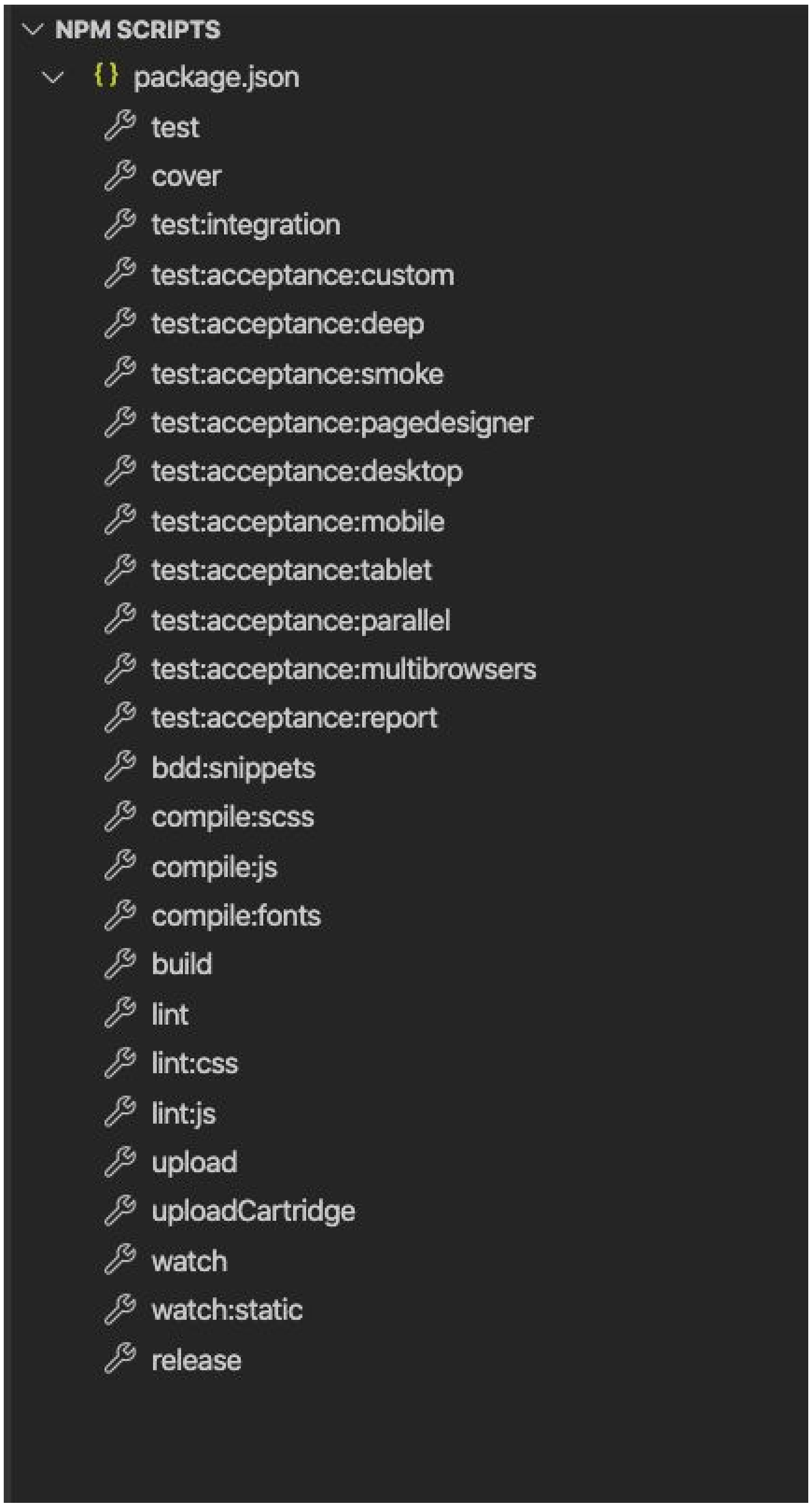
Defining tasks in the package.json is straightforward. Add the task under “scripts,” as in this example:

```
"scripts": {
   "build": "npm run compile:js;npm run compile:fonts;npm run
compile:scss;",
 },
```

After the tasks have been defined, you can run them from the command-line using the npm-run command

```
> npm run build
```

In addition, some IDEs, such as Visual Studio Code provide a simple visualisation of your NPM tasks.



NPM SCRIPTS

{ } package.json

- test
- cover
- test:integration
- test:acceptance:custom
- test:acceptance:deep
- test:acceptance:smoke
- test:acceptance:pagedesigner
- test:acceptance:desktop
- test:acceptance:mobile
- test:acceptance:tablet
- test:acceptance:parallel
- test:acceptance:multibrowsers
- test:acceptance:report
- bdd:snippets
- compile:scss
- compile:js
- compile:fonts
- build
- lint
- lint:css
- lint:js
- upload
- uploadCartridge
- watch
- watch:static
- release

Building Code for a CI/CD Process

Building your code on B2C commerce involves linting and bundling static files, such as javascript, CSS, and images.

ESlint helps to create consistent code that adheres to predefined styling rules and identifies errors during the build process. The StyleLint tool can accomplish the same tasks for SCSS and CSS.

You can do client-side bundling with module bundlers like Webpack or ParcelJS.
Those tools help to structure your javascript and CSS code into modules and optimize the modules during build time.

Setting Up SFRA

If you are extending the Storefront Reference Architecture (SFRA) for your own implementation, you can define SFRA as a NodeJS dependency in package.json and also include scripts in the package.json file.

When you add SFRA as a NodeJS dependency, the application is stored inside the node_modules folder. Since many build systems provide capabilities to cache node-dependencies, this approach allows you to reduce your build-time and avoid cloning SFRA within a longer process on each CI-process. Bundling all scripts into a single command to setup your projects eliminates some repetitive tasks in your process.

The following example shows a package.json file modified to include SFRA as a dependency and use scripts.

```
{
 "name": "your application",
 "version": "1.0.0",

 ....
 "scripts": {
     ....
    "build:sfra" : "cd salesforce-storefront-reference-architecture
&& npm run build"
    "init:sfra": "rsync -a ./node _modules/sfra/
salesforce-storefront-reference-architecture",
    "setup:project": "npm install && npm run init:sfra && npm
build:sfra"
 },
 "dependency": {
   "sfra":
"https://github.com/SalesforceCommerceCloud/storefront-reference-arch
itecture",

 },
 ....
}
```

Executing npm run setup:project downloads, installs, and builds SFRA**.**

To decrease build time, you could also run your build tasks in parallel, for example using Concurrently.

```
"parallel:test+lint": "concurrently \"npm run test\"  \"npm run
lint\" "
```

Importing Data

You can update B2C Commerce instances using our standard site import process. The site import feature allows you to populate different environments with the same data. Site imports provide an easy way to get sandboxes quickly up and running.

The Site Import/Export documentation describes how to prepare the site data for import.

A best practice is to put the site import files into a common area such as Git, thereby allowing the whole team to synchronise data changes. You can specify different site import folders for each build environment.

- **common** / site_template
  - Can be uploaded safely to either sandbox, development, or staging
  - Includes metadata (like system-object-definitions)
- **testdata** / site_template
  - Should be uploaded only to sandbox or development environments
  - Includes data used for automated integration and acceptance tests

You can build a NodeJS script, for example, dataUpload.js**,** that selects the correct site import folder for your build process, as in this example:

```
// dataUpload.js
const environment = process.env.TARGET_ENVIRONMENT;
let folder;

if (environment === 'common') {
   folder = 'common/site_template';
   siteTemplate = 'site_template';
} else if (environment === 'test') {
   folder = 'testdata/site_template';
}
...
```

Creating Code and Uploading Data

You can upload code and metadata using the Open Commerce API (**OCAPI**). The SFCC-CI tool provides a “wrapper” around OCAPI that allows you to easily accomplish these tasks. The SFCC-CI tool is open sourced. Install it using `npm install sfcc-ci`. The README file in the SFCC-CI GitHub repository provides information about how to set up and use SFCC-CI:

https://github.com/SalesforceCommerceCloud/sfcc-ci/blob/master/README.md

You can execute SFCC-CI commands either by using the command-line or the NodeJs-API. Which approach you use depends on your configuration. The NodeJS-API provides better response handling, while the command-line approach is faster to set up.

For CI/CD, the following SFCC-CI commands are most relevant:

- **client:auth**
    - For unattended deployment, you can authenticate using Account Manager credentials with clientId & clientSecret.
        - The clientSecret needs to be stored securely, for example, as a secured environment variable in your build system like in **Github Secrets**
- **code:upload & code:activate**
    - Provides a mechanism to upload a zipped code version to B2C commerce WebDAV and activate that code.
- **instance:upload & instance:import**
    - Uploads zipped site import data to a B2C commerce instances and runs the import job.
- **job:run** *<jobName>*
    - Runs a job in B2C Commerce to process data on the instance.

Uploading Code

The following example illustrates how to upload and activate a code version file using SFCC-CI from the command line and using NodeJS-API.

CodeUpload: Command-Line API Approach

```
$> sfcc-ci client:auth $client_ID $client_SECRET
package.json
{
 "name": "your application",
 "version": "1.0.0",

 ....
 "scripts": {
     ....
     "auth:unattended": "sfcc-ci client:auth $client_ID
$client_SECRET",
     "code:upload": "sfcc-ci code:deploy code_version.zip",
     "code:activate": "sfcc-ci code:activate code_version",
     "code:deploy": "npm run code:upload && npm run code:activate"
 },
 "dependency": {
   "sfra":
"https://github.com/SalesforceCommerceCloud/storefront-reference-arch
itecture",

 },
 ....
}
```

CodeUpload: NodeJS-API Approach

```
/**
  * Upload to webDav using sfccCi
  * @param {string} codeVersionToUpload - path of code version
  * @param {string} codeVersionName  - name of code version
  */
 uploadCode(codeVersionToUpload, codeVersionName) {
   return new Promise((resolve, reject) => {
     console.info('Start code upload');
     sfcc.auth.auth(this.clientId, this.clientSecret, (err, token) =>
{
       sfcc.code.deploy(
         this.instance,
         `${codeVersionToUpload}.zip`,
         token,
         {},
         err => {
           if (err) {
             console.error('Code deploy error: %s', err);
             reject(err);
             return;
           }
           console.info('Finished code deploy');
           /**
           * Active the code version
           */
           sfcc.code.activate(this.instance, codeVersionName, token,
err => {
             if (err) {
               reject(err);
               return;
             }
             resolve();
           });
         },
       );
     });
   });
 }
```

Uploading Data

Depending on what kind of data was uploaded, sometimes you have to trigger addiional steps on the B2C Commerce instance in order to have the data fully functional. An example would be assigning products to a given category that involves triggering site-import-processes on the instance.

The following example illustrates how to upload, process, and post-process (meta) data.

```
# Upload the metadata file in B2C Commerce
> sfcc-ci instance:upload yourmetadata.zip
# Import the uploaded metadata file
> sfcc-ci instance:import yourmetadata.zip
/**
* Upload to webDav using sfccCi
* @param {string} zipPath - path of zip file
* @param {string} zipName  - name of zip file
*/
uploadAndImportMetaData(zipPath, zipName) {
   return new Promise((resolve, reject) => {
       sfcc.auth.auth(this.clientId, this.clientSecret, (err, token)
=> {
           if (err) {
               reject(err);
               return;
           }
           sfcc.instance.upload(this.instance, zipPath, token, {},
err => {
               if (err) {
               reject(err);
               return;
               }
               sfcc.instance.import(this.instance, zipName, token,
err => {
               if (err) {
                   reject(err);
                   return;
               }
               resolve();
               });
           });
       });
   });
}
```

After the data has been uploaded, a job process is started that triggers, for example, a Search-Index rebuild.

```
# Reindex is the name of the job on B2C Commerce
> sfcc-ci job:run Reindex
/**
* Triggers a job on B2C commerce
*/
uploadAndImportMetaData(packageId, jobId, jobParams) {
   return new Promise((resolve, reject) => {
       sfcc.auth.auth(this.clientId, this.clientSecret, (err, token)
=> {
           if (err) {
               reject(err);
               return;
           }
           sfcc.job.run(this.instance, jobId, jobParams, token, {},
err => {
               if (err) {
                   reject(err);
                   return;
               }
               resolve();
           });
       });
   });
}
```

Executing the CI Process

Execute the CI process using a separate build system. Common build systems are bitbucket-pipelines, Github Actions, CircleCi, AWS-CodeBuild, and Jenkins.

Since the storefront-reference-architecture (SFRA) repository provides a bitbucket build pipeline, we focus on bitbucket in this document, but the same information applies to other build systems. When choosing a build system, consider whether you want to maintain your own build system, like Jenkins, or prefer to use an out-of-the-box solution such as Bitbucket pipelines.

You can modify the standard bitbucket-pipeline.yaml configuration to support the CI approach. The following sections show how to use npm-tasks inside a bitbucket-pipeline in combination with SFCC-CI.

Standard Approach

```
# This is a sample build configuration for Javascript.
# Check our guides at https://confluence.atlassian.com/x/VYk8Lw for
more examples.
# Only use spaces to indent your .yml configuration.
# -----
# You can specify a custom docker image from Docker Hub as your build
environment.
image: node:6.9.2
pipelines:
 default:
   - step:
       script: # Modify the commands below to build your repository.
         - npm install
         - npm run lint
         - npm test
         - npm run compile:js
         - npm run compile:scss
         - npm run compile:fonts
```

Extended CI Approach

```
# You can specify a custom docker image from Docker Hub as your build
environment.
image: node:10.16.4
pipelines:
 pull-requests:
   '**':
    ###
    # Target environment sandbox
    ###
    - step:
         script:
           ##
           # Summed up SFRA installment
           ##
           - npm run setup:project
           - npm run auth:interactive $client_ID $client_SECRET
           ##
           # assuming we created data + code upload
           ##
```

```
          - npm run dataupload 'common'
          - npm run dataupload 'testdata'
          - npm run code:deploy
          ##
          # execute further instance specific build steps
          ##
          - npm run test:acceptance:smoke
branches:
   ###
   # Target environment development
   ###
   release/v*:
     - step:
        script:
          ##
          # Summed up SFRA installment
          ##
          - npm run setup:project
          - npm run auth:interactive $client_ID $client_SECRET
          ##
          # assuming we created data + code upload
          ##
          - npm run dataupload 'common'
          - npm run dataupload 'testdata'
          - npm run code:deploy
          ##
          # execute further instance specific build steps
          ##
          - npm run test:acceptance:deep
tags:
  ###
  # Target environment staging
  ###
    '**':
       - step:
            script:
              ##
              # Summed up SFRA installment
              ##
              - npm run setup:project
              - npm run auth:interactive $client_ID $client_SECRET
              ##
              # assuming we created data + code upload
              ##
              - npm run dataupload 'common'
```

GitHub Example

```
name: Development CI

on:
 push:
   branches: [ develop ]

jobs:
 build:

   name: Deploy - Integration instance

   runs-on: ubuntu-latest

   steps:
   # Check out the repository
   - uses: actions/checkout@v2

   # Install Node.js
   - uses: actions/setup-node@v1
     with:
       node-version: 10
   - run: npm run setup:project
   - run: npm run lint
   - run: npm run auth:unattended ${{ secrets.CLIENT_ID }} ${{
secrets.CLIENT_SECRET }}
   - run: npm run config:generate:environment ${{
secrets.INTEGRATION_ENV }}
   - run: npm run dataupload 'common'
   - run: npm run dataupload 'testdata'
   - run: npm run code:deploy



name: Run ESLint on Pull Requests

on:
 - pull_request

jobs:
 build:
   name: Run ESLint
```

```
    runs-on: ubuntu-latest
    steps:

      # Check out the repository
      - uses: actions/checkout@v1

      # Install Node.js
      - uses: actions/setup-node@v1
        with:
          node-version: 10

      # Install your dependencies
      - run: npm ci
      - run: npm run lint
```

Integration Process: Sandbox

Sandboxes are not intended to be a 1:1 copy of a development or staging environment. A sandbox is designed for development and not for a large amounts of data. As a general rule, having 500-1000 products, including prices and inventory, is more than sufficient for all development tasks. If other products or categories are needed, they can easily be added using the site import process.

For CI Processes on a sandbox, a larger set of products or other data means longer build times and therefore a longer period before a code change can be approved through test automation. Our recommendation is to select dedicated products for individual test cases, so you have only, for example, a single preorder product that your test automation uses to verify the preorder functionality.

B2C Commerce on-demand sandboxes are a great fit for the CI/CD process. The on-demand sandboxes can be created and populated during the CI/DC process for integration and testing purposes. Refer to the on-demand sandbox documentation for more information.

Integration Process: Development

Integration from the development instance integration process is very similar to the process for sandboxes. Integration from development serves as the last quality gate before pushing code to the staging instance. There should be a complete test suite running to ensure the code not only works as expected for integration but also from an end-user perspective. In most projects, the QA engineers are testing on the development instance. The automated test suites provide even more complete regression testing.

As the data is synchronised with live data coming from staging, make sure that your test suite can handle changes in the data set. For example, if you are testing the on-site search and want to find out if the word *blue* returns a result, make sure that actual products are shown for the search. Don't just check to see if the search result count is 12.

Integration Process: Staging

Creating a meaningful CI process for staging is the most difficult part of the setup. Staging requires its own process for code and data upload.

Code Upload on Staging

Two-factor authentication using a client certificate as the second factor is required when uploading code on staging. The two-factor authentication adds an extra layer of security to the platform. Adding two-factor authentication to the code deployment process is easy using SFCC-CI.
Before running any code, you must create a **.p12** certificate file, as in the following example.

Important: Do not store the certificate and the password together. This blog post provides good information about security best practices for CI/CD.

```
###
# Example of creating a .p12 file using openssl
###
> openssl req -new -newkey rsa:1024 -nodes -out {CustomName}.req
-keyout {CustomName}.key
> openssl x509 -CA
cert.staging.web.{YourInstance}.demandware.net_01.crt -CAkey
cert.staging.web.{YourInstance}.demandware.net_01.key -CAserial
cert.staging.web.{YourInstance}.demandware.net.srl -req -in
{CustomName}.req -out {CustomName}.pem -days 7300
> openssl pkcs12 -export -in {CustomName}.pem -inkey {CustomName}.key
-certfile cert.staging.web.{YourInstance}.demandware.net_01.crt -name
"{CustomName}" -out{CustomName}.p12
```

SFCC-CI allows you to pass the .p12 file using either the command line or the NodeJS-API.

Example Using Command Line API

```
###
# SFCC-CI command-line example
###
> sfcc-ci code:deploy code_version.zip -i my-instance.demandware.net
-c path/to/my/certificate.p12 -p "myPassphraseForTheCertificate"
```

Using NodeJS-API

```
###
# SFCC-CI JS
###
/**
* Upload to webDav using sfccCi
* @param {string} codeVersionToUpload - path of code version
* @param {string} codeVersionName  - name of code version
*/
uploadCode(codeVersionToUpload, codeVersionName) {
   return new Promise((resolve, reject) => {
     console.info('Start code upload');
     sfcc.auth.auth(this.clientId, this.clientSecret,
     ###
     # Allows to pass the certificate for the upload process
     ###
     { pfx: certificatePath, passphrase: certPassPhrase }, (err,
token) => {
       sfcc.code.deploy(
         this.instance,
         `${codeVersionToUpload}.zip`,
         token,
         {},
         err => {
           if (err) {
             console.error('Code deploy error: %s', err);
             reject(err);
             return;
           }
           console.info('Finished code deploy');
           sfcc.code.activate(this.instance, codeVersionName, token,
err => {
             if (err) {
               reject(err);
               return;
             }
             resolve();
           });
         },
       );
     });
   });
}
```

Data Upload on Staging

Data uploading requires a robust process, which guarantees, to not overwrite data on the instance where other people are working on. Let’s have a look at an example which demonstrates the challenge

Developer A is building a new content slider for the homepage. Therefore content assets are used, in order to provide capabilities to change the images within that slider without having to touch the code. Further requirements are, that the link of the images are interchangeable and the developer decides to add a new custom attribute to the content asset, which allows to provide links.
The developer needs to push two kind of different data with the deployment to the staging instance, system-object-extensions and the library which includes the content. If the library remains in the version control system, it will overwrite the content on the staging instance every time again, when a deployment happened. To avoid a manual overhead, like having the developer adding the library to each instance a **one time upload** can be build.

One-time Upload and Data Seeding

In general this is **custom code** which can recognise, if a certain set of data has already been pushed to the given instance.
The structure in the repository might look like this:

```
###
# Illustrates how to organize data for data seeding inside a repository
###
data_{TimeStamp1}
|
|--- site_template
      |
      |--- sites
             |
             |--- {SiteID}
                    |
                    |---library.xml
##### another folder ####
data_{TimeStamp1}
|
|--- site_template
      |
      |--- sites
             |
             |--- {SiteID}
                    |
                    |---library.xml
```

This allows to store the latest upload timestamp either in cloud or local database, which can read during the deployment. By comparing the timestamps, the logic within the CI process is able to decide, wether or not the data should be added to the instance.

Which Tests to Run on Which environment

Testing your software application is a key part of the CI/CD process. We can provide only an overview in this document, but using test-driven development processes and providing accurate metrics for testing reduces time and cost significantly.,

Unit Tests

Unit Tests ensure that a code unit is robust on its own and can run on the CI machine with a mocked B2C Commerce API. No B2C Commerce servers are involved. You can write unit tests using tools such as ChaiJs, MochaJS, and other state-of the art testing frameworks for nodeJS. SFRA includes unit tests out-of-the-box.

Integration Tests

Use integration tests to make sure your code modules can interact with each other. Write integration tests for any custom code that affects business critical processes, such as basket and order calculation, or customer operations. If your code interacts with web services, the B2C Commerce platform allows you to simulate service calls as mock calls.

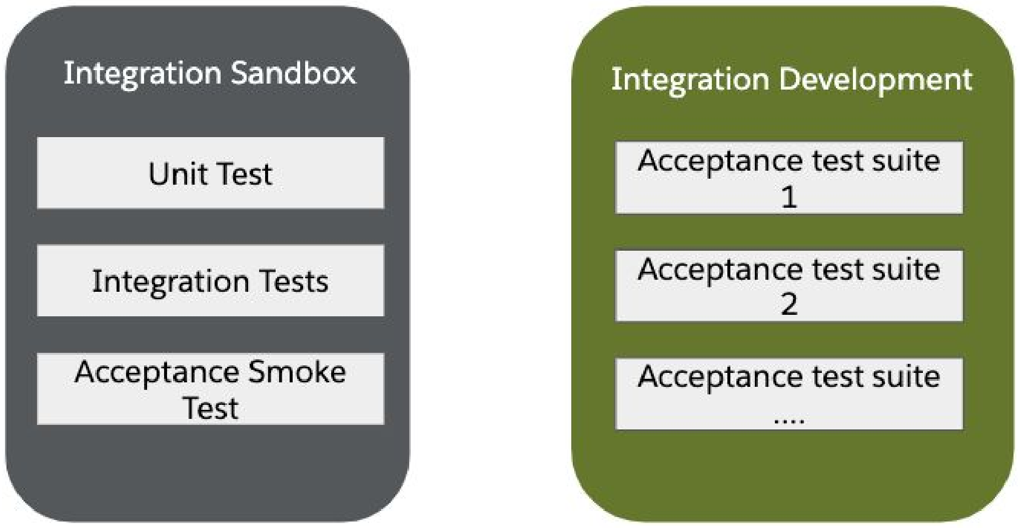
To expose data through REST during an integration test, you can use a separate integration_test cartridge that is not deployed to the staging instance. With this approach, you can ensure that no REST API endpoint exposes data. An environment guard, serving as middleware inside your controller, can provide an additional layer of security.

```
function ensureTestEnvironment(req, res, next) {
   if (System.getInstanceType() === System.PRODUCTION_SYSTEM) {
       throw new Error('Request forbidden on this system');
   }
   return next();
}
```

Acceptance Tests

Acceptances test evaluate if the features you are building work as specified in the definition of done and from an end-user perspective. SFRA provides out-of-the-box acceptance test capabilities using **codeceptJS.** The SFRA tests allow you to quickly get started with test-driven development. CodeceptJS also supports headless scenarios, making is a great fit for CI/CD processes.

The following illustratration shows how different test suites can be executed on different environments.



Conclusion

The CI/CD process allows you to deliver high quality software quickly while reducing manual setup and testing steps. A carefully implemented CI/CD process allows your development team to work as efficiently as possible, with test suites that uncover errors throughout the development cycle.

You can also apply the principles discussed in this document to implement Continuous Delivery for automated deployments onto production. Once you have a dependable test automation system in place, you can trigger a replication of data code using the build system and the job framework.