Builder Documentation

Release 0.0.1

Dan Nolan

Dec 07, 2022

Contents

What is ChainShot?

ChainShot is an interactive learning website for Ethereum developers.

What is ChainShot Builder?

The ChainShot Builder is an application that allows Content Creators to build their own interactive learning content.

CHAPTER 1

Glossary

1.1 Quick Start

For an easy Setup, we commend you use the :

Simply install `chainshot-builder` globally using `npm`, then run it from your new content repository!

```
npm install -g chainshot-builder
mkdir myContentRepository
cd myContentRepository
chainshot-builder init
```

And that's it!

The Builder CLI will spin up the IDE for you. Any changes you make within the UI will be written to your file system.

Simply commit any changes up to Github (*Creating a git repository*) and then link ChainShot to your Github account (*Linking your Github Account*)!

If you'd like to develop on the Builder itself, check out *Developing on Builder*.

1.2 Build New Content

Here are the steps to start from scratch and build your own content.

1.2.1 Setting Up

First, if you haven't already, follow the instructions on the *Quick Start*.

1.2.2 Start Editing

Once you have Builder up and running you'll be able to create your own content. You'll be given the choice between starting from any of the stage container types (see *Container Types*).

Once you're inside the editor you'll be able to configure your stage container and start adding stages. The Builder will automatically create a folder called `Content` (unless overridden by *Server Configuration*). Any changes made within the editor will automatically save to your file system.

1.2.3 Creating a git repository

Once you have a Content Folder, you'll want to create a git repository.

Navigate to your content folder. By default it is created on the same directory level as your Builder repository, so it should look like this on your file system:

```
/Builder
/Content
```

Navigate to `/Content` and initialize a new repository, and commit all contents:

```
git init
git add .
git commit -m "init content"
```

Now you'll need to connect it to a remote. Easiest way to do this is to go to Github and .

Note: This repository must be public. If you'd like to setup a private repository integration, contact us team@chainshot.com

Then connect to your remote repository:

```
git remote add origin PUT_YOUR_REPO_GIT_URL_HERE
git push -u origin master
```

1.2.4 Connecting your Github Repository

For instructions on linking your new Github Repository to Chainshot, check out our *Linking your Github Account* section.

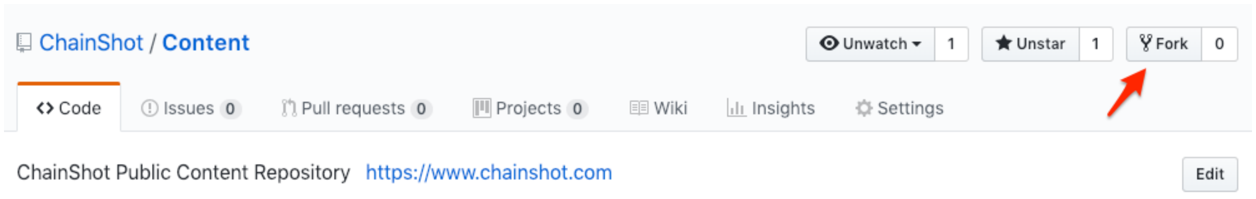
1.3 Fork Existing Content

The beautiful thing about open-source content repositories is you don't need anyone's permission to fork their content and begin making your own modifications!

Let's see how to go about cloning a content repository and hosting it on ChainShot.

1.3.1 Forking a Content Repository

Once you find a content repository that you'd like to fork you can do so from Github. For example for the :



ChainShot Public Content Repository https://www.chainshot.com

Then you can clone your forked repository and start editing it with the Builder!

1.3.2 Point the Builder

To edit your forked repository with the Builder you'll want to install `chainshot-builder` globally using `npm` if you haven't already:

```
npm i chainshot-builder -g
```

Once you've done that, you'll want to navigate your forked repository, then simply:

```
cd path/to/my/repo
chainshot-builder run
```

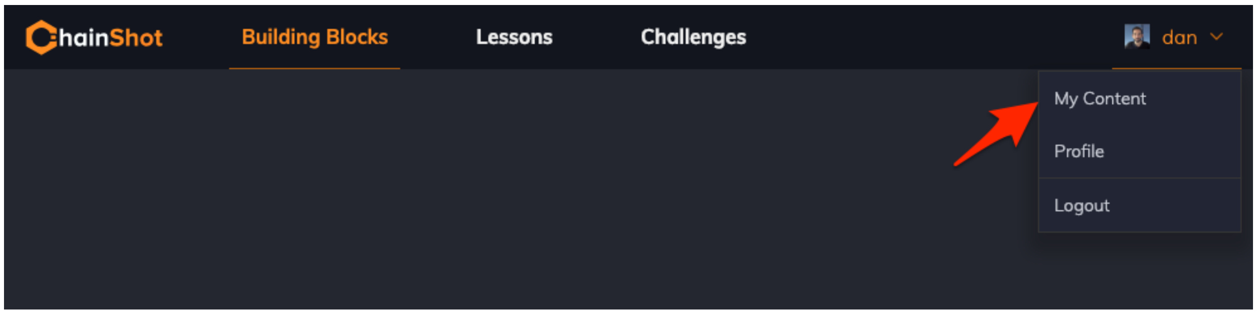
The app will spin up and you'll be ready to start making updates! Any changes made within the app will update your file system and can be committed back up to Github.

1.4 Linking your Github Account

Once you have your Content repository up on Github you'll be able to link to it from ChainShot to hook up to automatic deployments. ChainShot will transform your content repository into interactive tutorials. Any changes to your content repository will automatically be reflected on the ChainShot site.

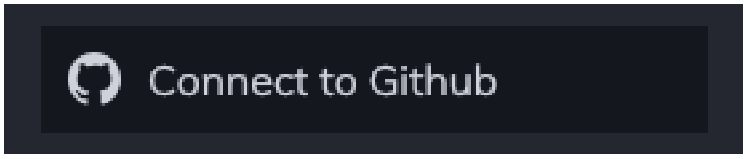
1.4.1 Connect to Github

To connect your ChainShot account, visit ChainShot. Navigate to "My Content" in your profile dropdown ChainShot:

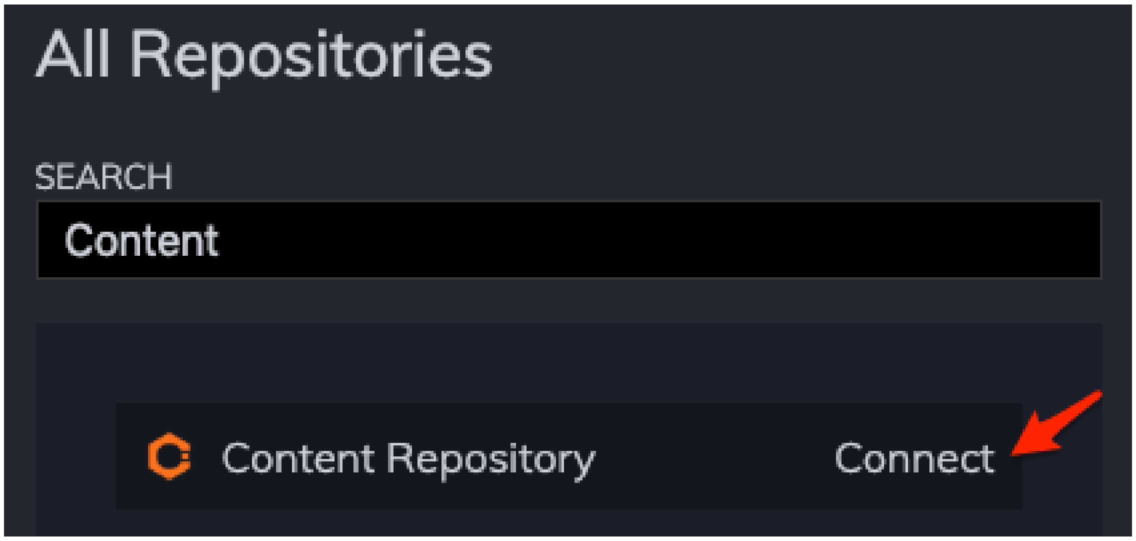


Or follow this link to .

You'll have to connect your Github account via oauth:



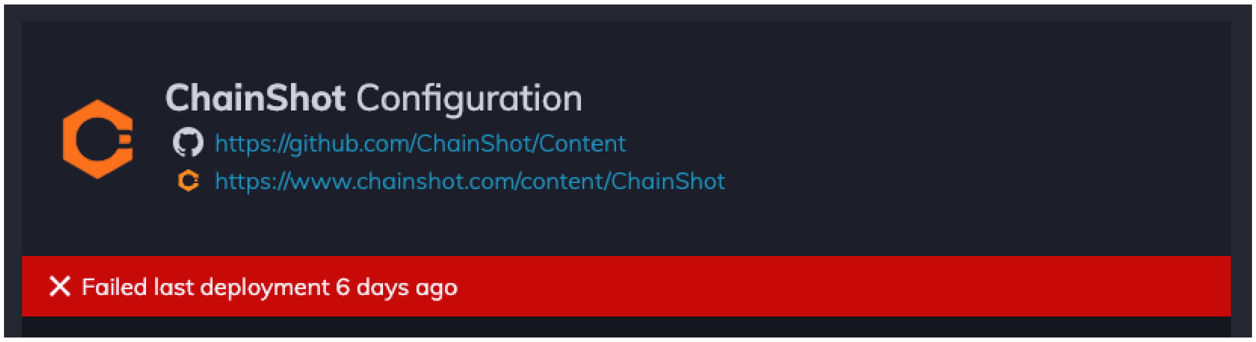
And then you'll be able to connect your Content Repository:



1.4.2 Github Webhook

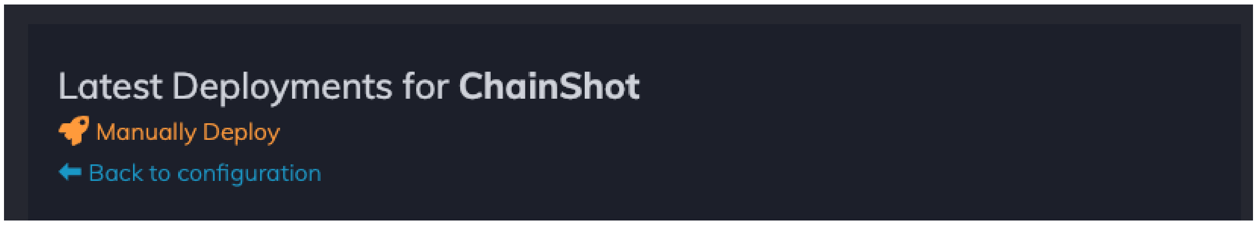
Once you have hooked up your Content repository on ChainShot a webhook will be created on your Github repository. This webhook will automatically notify ChainShot when changes are made to your Github repository so that the new Content can be deployed.

You can view the status of the latest deployments on your deployments page:


ChainShot Configuration
https://github.com/ChainShot/Content
https://www.chainshot.com/content/ChainShot
Failed last deployment 6 days ago

1.4.3 Manual Deployment

If your deployments ever fail for some reason you can always kick off a manual deployment:



For any help or assistance on why your content has failed deployment, reach out to ChainShot administrators on Slack and we'll get you set up :)

1.5 Content Repositories

ChainShot learning content is stored in git repositories to allow for maximum collaboration between content creators.

Content is built using the Builder application locally and then committed up to Github. From there, ChainShot users can connect their Github account, link their content repository and start teaching their audience.

Now let's take a closer look at content repositories.

1.5.1 Structure

Content Repositories are split into two folders: config and projects.

/config

Within the config directory you'll find subdirectories containing JSON documents of every type of model described in *Model Types*.

These JSON documents contain mostly the configuration data that is used to create the database documents when the content repository is deployed.

For any properties that are file-like (i.e. markdown, long text or code) the value will be stored as a $$LOOKUP which means that it will be created under the projects directory within a particular file. This makes it easier to see file differences when many collaborators are working on the file, just like how a file diff would look in a codebase.

/projects

This directory contains all of the project files. Each stage is meant to be it's own standalone project that can be run locally as well as within the Builder UI.

The structure of the projects are determined by the model and it's properties as follows:

/{StageContainerGroup.title}/{StageContainer.version}/{Stage.title}/

Whenever these properties change, the corresponding files will be relocated to the new directory accordingly.

1.5.2 Reserved URLs

One user can have many Content Repositories and can reserve URLs on ChainShot to display their learning content. For instance:

wwww.chainshot.com/content/chainshot

This is the main content repository for ChainShot. On Github, this repository can be found at:

www.github.com/ChainShot/content

1.6 Model Types

1.6.1 Stage Container Groups

Stage Container Groups are used to group together different versions of the same content.

For instance, if we had a tutorial that used Solidity and web3.js, there might be another version that uses Vyper and web3.js. Or maybe one that uses Solidity and ethers.js.

Fields

Field	Description
id	Mongo ID identifier
title	Short Name
description	Description of the Tutorials Purpose
containerType	Refers to the stage container *Container Types*
productionReady	Should this appear in production to users

1.6.2 Stage Containers

As you might expect, a stage container holds many stages. Stage Containers are abstractly named because there are three different *Container Types*: *Building Blocks*, *Lessons* and *Challenges*.

Fields

Field	Description
id	Mongo ID identifier
version	Uniquely identifies this container from others in the group
type	See *Container Types*
intro	Markdown which is shown to the user when they begin
stageContainerGroupId	The ID of the group to which this container belongs

1.6.3 Stages

Stages are the main interactive piece of content. They contain some text documentation which helps the user understand their goals.

There are four different types of Stages. To learn more about each type of stage see *Stage Types*.

The fundamental properties for all stages are:

Fields

Field	Description
id	Mongo ID identifier
title	Short Name
type	See *Stage Types*
completionMessage	A message shown to the user after completion
position	Integer value for the stage position in the container (zero-indexed)
task	Markdown file describing the users task in this stage
details	Markdown file containing additional context information for the user

1.6.4 Code Files

Code Files represent a file within a CodeStage. They come with a number of properties that describe how they should be rendered and executed within the stage.

Fields

Field	Description
id	Mongo ID identifier
name	Short Name
initialCode	The code which this CodeFile should begin with for the user
mode	The Monaco Code Editor mode to display the code in
stageContainerId	Which of the *Stage Containers* this belongs in
codeStageIds	Array of IDs of all *Code Stage* this code file belongs to
executable	Should this be included when the stage code is executed?
executablePath	The path at which the code file is executed
hasProgress	Should ChainShot track users progress for this CodeFile?
readOnly	Should users be allowed to change this file?
testFixture	Is this a file with test cases in it?
visible	Should this be shown to the user? (sometimes turned off for utils)

1.6.5 Solutions

Solutions are predominantly for reference and help designing the test cases.

They ensure that there is a working solution and can be shared across collaborating content creators to ensure the test cases are still working properly for any updates.

Fields

Field	Description
id	Mongo ID identifier
codeFileId	Which of the *Code Files* this belongs to
stageId	Which *Code Stage* this belongs to
code	The actual solution code

1.6.6 Badge Types

Badges are a great way to reward users for completing your content! The BadgeType model defines the badge classification.

Each user who completes your content will be rewarded with an instance of that badgeType as long as the *badgeLimit* hasn't been reached.

Fields

Field	Description
id	Mongo ID identifier
name	Short Name
description	How did the user earn this badge?
badgeLimit	The number of badges of this type to be distributed
thumbnailUrl	A URL for the image of this badge

1.7 Container Types

1.7.1 Building Blocks

Tutorials based around building a project that can be continued locally after the web tutorial is complete.

Use Cases

Building Blocks are super useful as a starting point for hackathons or projects to be worked on without creative limitations.

For instance, a Building Block could start all students off with an ERC20 project where the Building Block will teach them how to write ERC20 tokens and connect them to a JavaScript application. Objectively, the Building Block can judge whether or not the student has mastered the concepts by assessing the code through a series of test cases.

Once the student has passed all their test cases, they can download their code into a fully working project using *Project Skeletons*.

1.7.2 Lessons

Self-Contained tutorials that walk someone through an important concept through a series of guided code challenges, videos and walkthroughs.

Use Cases

Lessons are particularly suitable for tutorials that are more theoretical or academic in nature. They don't necessarily revolve around a project like a Building Block. Instead, they focus on writing code and teaching concepts that may not come up in the process of creating an application.

For instance, a lesson could teach "How to write a Blockchain". Typically, most students won't need to write their own blockchain, and the resulting code may not be the most secure code in the world. However, the conceptual knowledge will come in handy.

1.7.3 Challenges

Code Tests designed to extend upon knowledge that is taught in lessons and building blocks. Unlike lessons, necessary information to complete it may need to be looked up.

Use Cases

Challenges are often useful at the end of a Lesson or a Building Block to take knowledge learn and apply it in a more difficult environment. Hopefully, users do not often get stuck within a lesson or a building block. However, for a challenge a certain amount of "healthy frustration" may be helpful for learning.

Challenges can also be used in Competitions, which are not currently available through the Builder but will likely be added in the near future.

1.8 Stage Types

1.8.1 Code Stage

This is the bread and butter stage for ChainShot. It has some text documentation and a Code Editor which can flip between many files.

The user will need to pass all test cases for the *testFixture* in order to move onto the next stage.

Along with all the fields mentioned for *Stages*, here are the Code Stage fields:

Fields

Field	Description
language	The execution environment in which the code is run
languageVersion	The version of the execution environment where the code is run
testFramework	The test framework that is used to execute the test cases
abiValidations	See *ABI Validations*
validatedContract	The contract that is validated with the abi validations

1.8.2 Download Stage

This stage is the primary ending point for *Building Blocks*. Since Building Block tutorials are based around a project, the download stage allows the user to download all of their code into *Project Skeletons*.

Along with all the fields mentioned for *Stages*, here are the Download Stage fields:

Fields

Field	Description
projectSkeletons	An embedded array of *Project Skeletons*

1.8.3 Video Stage

A stage that renders with an embedded video that can teach concepts that require more of a visual or audio perspective.

Fields

Field	Description
youtubeId	YouTube ID for the video (found on the URL, i.e. ?v=ID)

1.8.4 IFrame Stage

A Stage that which will embed an IFrame that can point at another site. This is used for stages that want to show some kind of visualization or other UI that cannot be created from the other Stage Types.

Note: Currently the site needs to be whitelisted before it can be rendered. In the future this may change to a UI that clearly shows the user is working on another site.

Fields

Field	Description
src	The URL to render in the IFrame

1.9 Project Skeletons

Project Skeletons are working applications that are intended to work along with *Building Blocks*. Since Building Blocks are project-based tutorials, at the end the user will be able to download all of their code into a "skeleton" of a project. The user's code combined with the skeleton creates a fully functional app that is partially written by the user.

1.9.1 Github Repository

To create a Project Skeleton, the best first step is to create a working application. This will be application you essentially want your users to build.

Once you've created this application, upload it to Github.

Next, you'll want to take out the core components that can be written by the user, that will serve as good learning points and turn them into Code Stages in a Building Block.

1.9.2 Connecting The Building Block

Once you have your working application built, you'll want to connect it to a Building Block Tutorial.

Create the Block

If you've setup your Builder application (*Quick Start*), you'll be able to create your own Building Block.

You can create some stages that are designed to help the user learn about the project by having them write code to pass certain test cases.

Setting File Locations

For Code Files that will be combined with a Project Skeleton, it is necessary to set the `file_location` property. This property will indicate where the file will be placed in the project skeleton relative to the base of the project.

Download Stage

Once you have finished your Building Block and setup the Code File locations, the final step is to create a *Download Stage*. This stage will allow your users to pick a project skeleton among your project skeleton(s) to download their code into.

You'll need to set the Project Skeleton Fields and associate it to a Github repository.

1.9.3 Fields

Here are the fields for the embedded project skeleton:

Field	Description
id	Mongo ID identifier
title	Short Name
description	Short Explanation of the Skeleton
ghNodeId	Global Github Identifier
ghRepoId	Github Repository Identifier
thumbnailUrl	Image URL to represent the Skeleton
zipName	The name of the zip folder the user will download the code into

1.9.4 Example

For an example of a Skeleton, see the which is used for the .

1.10 ABI Validations

These are a set of rules that help a user determine if the smart contract they are writing are exposing properties as expected.

Since test cases often require that a function signature looks a particular way, it's very easy to quickly guide the user and let them know if they are on the right track.

Since compilation happens very quick relative to code execution validations are often preferable to having a user execute their code and have to wait only to find out they misnamed a function or forget to add a public keyword.

Note: Within the Builder UI ABI Validations are validated as JSON. This makes it difficult for non-advanced content creators to edit for the moment.

1.10.1 Languages

Since ABI Validations are used to validate the ABI created by EVM languages, they are available for Solidity and Vyper only.

1.10.2 JSON Format

ABI Validations store an array of JSON objects with properties that allow ChainShot to take a contract, compile it to an ABI, and compare to see if the properties exposed are as expected.

Here are the fields for each individual validation:

Fields

Field	Description
id	Mongo ID identifier
name	The name of the exposed property
interface_type	The type of the exposed property
task_display	The task to show to the user, for them to satisfy this validation
constant	Is this a constant function?
payable	Is this a payable function?
inputs	An embedded array of required inputs
outputs	An embedded array of required outputs

1.11 Markdown Editor

Many of the models in the builder have properties that will be rendered as markdown within the ChainShot application.

Markdown is a simple language that looks similar to text and can be output as HTML.

For instance the following:

```
## A Header
```

Would be translated to the HTML:

```
<h2>A Header</h2>
```

Here's a .

1.11.1 Embedding YouTube Videos

ChainShot will render YouTube videos. You can embed them by clicking "share" on your YouTube video and then finding the "Embed" option. This will give you some HTML in the form of an `<iframe>` which you can add to your markdown.

You can choose to enable privacy-enhanced mode, ChainShot will render videos from both `www.youtube.com` as well as `www.youtube-nocookie.com`.

1.12 Developing on Builder

1.12.1 Setting up for Development

Looking to contribute directly to the development of Builder? Great!

First, clone the .

```
git clone https://github.com/ChainShot/Builder.git
```

The Builder runs with a client/server architecture so it is broken into two main components client and server. We'll need to install the dependencies and run these components separately.

Install Dependencies

1) *Install Client Dependencies*
2) *Install Server Dependencies*

Start Client & Server

1) *Start Client*
2) *Start Server*

Configure

Both the server and the client can be configured to run with different environment settings, *Client Configuration* and *Server Configuration*.

1.13 Builder Structure

Hey there! If you're looking to help contribute to the builder the project is divided into three main folders: client, server and docs.

1.13.1 Client

The Builder Client is the main interface building content repositories. The UI is meant predominantly as an extension of an IDE for ChainShot content. As there are some intricacies involved with building the content, it is preferable to use this UI rather than edit the content repository directly.

Currently the client allows for the creation and modification of all the main model types (with the exception of a few special stage types). As more stage types and features are added to the ChainShot system, the Builder client will continue to add more functionality for recording videos, running competitions and holding livestreams.

Install Client Dependencies

To install dependencies for the Builder Client you can navigate to the /client folder and run `npm install`.

Start Client

To run the Builder Client you can navigate to the `/client` folder and run `npm start`.

Tech Stack

The client's main tech stack includes react, scss, and GraphQL.

Initially it was created using the create-react-app tool and then ejected for further configuration of it's webpack settings.

Client Configuration

Builder Client uses the dotenv node module which makes it easier to create override settings for your own environment without committing them.

To create your own environmental variables simply create a `.env` file in the `client/` directory. You can then override settings with the format:

```
REACT_APP_SETTING1=VALUE1
REACT_APP_SETTING2=VALUE2
```

Note: You'll notice the REACT_APP_* prefix on these settings. This is used by create-react-app to store environmental variables. Builder client was generated and ejected from the create-react-app tool.

Some of the override-able settings include:

Field	Description
REACT_APP_API_URL	The API URL for the builder server

Note: The client will fallback to the `window.location.port` unless given an explicit API URL. This is done so that the client can bind to dynamic ports from the CLI after it's been packaged up in a release build.

1.13.2 Server

The Builder Server acts as the intermediary layer between the client and the file system. It interprets many of the clients requests and translates them to ensure that the content is written and read in the expected fashion from the file system.

It uses websockets to ensure that updates are broadcasted to all listening clients, so that multiple tabs of the client UI do not get out of sync.

The builder is expected to be run on a single machine, locally, for a single user. As such, it's not safe for concurrency. Sending multiple updates at once may result in some unexpected overrides.

Install Server Dependencies

To install dependencies for the Builder Server you can navigate to the `/server` folder and run `npm install`.

Start Server

To run the Builder Server you can navigate to the `/server` folder and run `npm start`.

Tech Stack

The client's main tech stack includes node, express, socket.io and GraphQL.

Server Configuration

Builder Server uses the `dotenv` node module which makes it easier to create override settings for your own environment without committing them.

To create your own environmental variables simply create a `.env` file in the `server/` directory. You can then override settings with the format:

```
SETTING1=VALUE1
SETTING2=VALUE2
```

Some of the override-able settings include:

Field	Description
PORT	The port that builder runs on
CONTENT_REPO_NAME	The name of the content repo that builder looks for or creates

1.13.3 Docs

The Builder Docs are built using Sphinx and are hosted on Read The Docs. Since the docs are stored in rst files the intention is that anyone can come and make updates as necessary.

For installation instructions, see *Sphinx Docs* <'*http://sphinx-doc.org/*>'_. To build the docs locally simply run the `make` command and open the `index.html` file within the generated `_build` folder.