BinderHub Documentation

Release 0.1.0

Yuvi Panda

Oct 19, 2018

Contents

Note: BinderHub is under active development and subject to breaking changes.

CHAPTER 1

Getting started

The primary goal of BinderHub is creating custom computing environments that can be used by many remote users. BinderHub enables an end user to easily specify a desired computing environment from a GitHub repo. BinderHub then serves the custom computing environment at a URL which users can access remotely.

This guide assists you, an administrator, through the process of setting up your BinderHub deployment.

To get started creating your own BinderHub, start with *Create your cloud resources*.

CHAPTER 2

Extending JupyterHub

If you'd like to extend your JupyterHub setup, see Zero to JupyterHub.

CHAPTER 3

BinderHub Deployments

Our directory of BinderHubs is published at *BinderHub Deployments*.

If your BinderHub deployment is not listed, please open an issue to discuss adding it.

CHAPTER 4

Zero to BinderHub

A guide to help you create your own BinderHub from scratch.

4.1 Create your cloud resources

BinderHub is built to run on top of Kubernetes, a distributed cluster manager. It uses a JupyterHub to launch/manage user servers, as well as a docker registry to cache images.

To create your own BinderHub, you'll first need to set up a properly configured Kubernetes Cluster on the cloud, and then configure the various components correctly. The following instructions will assist you in doing so.

4.1.1 Setting up Kubernetes on Google Cloud

Note: BinderHub is built to be cloud agnostic, and can run on various cloud providers (as well as bare metal). However, here we only provide instructions for Google Cloud as it has been the most extensively-tested. If you would like to help with adding instructions for other cloud providers, please contact us!

4.1.2 Install Helm

Helm, the package manager for Kubernetes, is a useful tool for: installing, upgrading and managing applications on a Kubernetes cluster. Helm packages are called *charts*. We will be installing and managing JupyterHub on our Kubernetes cluster using a Helm chart.

Helm has two parts: a client (`helm`) and a server (`tiller`). Tiller runs inside of your Kubernetes cluster as a pod in the kube-system namespace. Tiller manages both, the *releases* (installations) and *revisions* (versions) of charts deployed on the cluster. When you run `helm` commands, your local Helm client sends instructions to `tiller` in the cluster that in turn make the requested changes.

Installation

While several methods to install Helm exists, the simplest way to install Helm is to run Helm's installer script in a terminal:

```
curl https://raw.githubusercontent.com/kubernetes/helm/master/scripts/get | bash
```

Initialization

After installing helm on your machine, initialize Helm on your Kubernetes cluster:

1. Set up a ServiceAccount for use by `tiller`.

   ```
   kubectl --namespace kube-system create serviceaccount tiller
   ```

2. Give the `ServiceAccount` full permissions to manage the cluster.

   Note: While most clusters have RBAC enabled and you need this line, you **must** skip this step if your Kubernetes cluster does not have RBAC enabled.

   ```
   kubectl create clusterrolebinding tiller --clusterrole cluster-admin --
   ↪serviceaccount=kube-system:tiller
   ```

   See our RBAC documentation for more information.

3. Initialize `helm` and `tiller`.

   ```
   helm init --service-account tiller
   ```

   This command only needs to run once per Kubernetes cluster, it will create a `tiller` deployment in the kube-system namespace and setup your local `helm` client.

   Note: If you wish to install `helm` on another computer, you won't need to setup `tiller` again but you still need to initialize `helm`:

   ```
   helm init --client-only
   ```

Verify

You can verify that you have the correct version and that it installed properly by running:

```
helm version
```

It should in less then a minute, when `tiller` on the cluster is ready, be able to provide output like below. Make sure you have at least version 2.11.0 and that the client (`helm`) and server version (`tiller`) is matching!

```
Client: &version.Version{SemVer:"v2.11.0", GitCommit:
↪"2e55dbe1fdb5fdb96b75ff144a339489417b146b", GitTreeState:"clean"}
Server: &version.Version{SemVer:"v2.11.0", GitCommit:
↪"2e55dbe1fdb5fdb96b75ff144a339489417b146b", GitTreeState:"clean"}
```

Note: If you wish to upgrade the server component of Helm running on the cluster (`tiller`):

```
helm init --upgrade --service-account tiller
```

Secure Helm

Ensure that tiller is secure from access inside the cluster:

```
kubectl patch deployment tiller-deploy --namespace=kube-system --type=json --patch='[{
↪"op": "add", "path": "/spec/template/spec/containers/0/command", "value": ["/tiller
↪", "--listen=localhost:44134"]}]'
```

Now that you've installed Kubernetes and Helm, it's time to *Set up the container registry*.

4.2 Set up the container registry

BinderHub will build Docker images out of Git repositories, and then push them to a Docker registry so that JupyterHub can launch user servers based on these images. You can use any registry that you like, though this guide covers how to properly configure two popular registries: the **Google Container Registry** (`gcr.io`) and DockerHub (`hub.docker.com`).

4.2.1 Set up Google Container Registry

To use Google Container Registry, you'll need to provide BinderHub with proper credentials so it can push images. You can do so by creating a service account that has authorization to push to Google Container Registry:

1. Go to console.cloud.google.com
2. Make sure your project is selected
3. Click `<top-left menu w/ three horizontal bars> -> IAM & Admin -> Service Accounts` menu option
4. Click **Create service account**
5. Give your account a descriptive name such as "binderhub-builder"
6. Click `Role -> Storage -> Storage Admin` menu option
7. Check **Furnish new private key**
8. Leave key type as default of **JSON**
9. Click **Create**

These steps will download a **JSON file** to your computer. The JSON file contains the password that can be used to push Docker images to the `gcr.io` registry.

Warning: Don't share the contents of this JSON file with anyone. It can be used to gain access to your google cloud account!

Important: Make sure to store this JSON file as you cannot generate a second one without re-doing the steps above.

4.2.2 Set up Docker Hub registry

To use **Docker Hub** as a registry first you have to create a Docker ID account in Docker Hub. Your Docker ID (username) and password are used to push Docker images to the registry.

If you want to store Docker images under an organization, you can create an organization. This is useful if different Binder instances want to use same registry to store images.

See the next section for how to properly configure your BinderHub to use Docker Hub.

4.2.3 Next step

Now that our cloud resources are set up, it's time to *Set up BinderHub*.

4.3 Set up BinderHub

BinderHub uses Helm Charts to set up the applications we'll use in our Binder deployment. If you're curious about what Helm Charts are and how they're used here, see the Zero to JupyterHub guide.

Below we'll cover how to configure your Helm Chart, and how to create your BinderHub deployment.

4.3.1 Preparing to install

To configure the Helm Chart we'll need to generate several pieces of information and insert them into `yaml` files.

First we'll create a folder where we'll store our BinderHub configuration files. You can do so with the following commands:

```
mkdir binderhub
cd binderhub
```

Now we'll collect the information we need to deploy our BinderHub. The first is the content of the JSON file created when we set up the container registry. For more information on getting a registry password, see *Set up the container registry*. We'll copy/paste the contents of this file in the steps below.

Create two random tokens by running the following commands then copying the outputs.:

```
openssl rand -hex 32
openssl rand -hex 32
```

Note: This command is run **twice** because we need two different tokens.

4.3.2 Create `secret.yaml` file

Create a file called `secret.yaml` and add the following:

```
jupyterhub:
    hub:
      services:
        binder:
          apiToken: "<output of FIRST `openssl rand -hex 32` command>"
    proxy:
      secretToken: "<output of SECOND `openssl rand -hex 32` command>"
```

Next, we'll configure this file to connect with our registry.

If you are using `gcr.io`

Add the information needed to connect with the registry to `secret.yaml`. You'll need the content in the JSON file that was created when we created our `gcr.io` registry account. Below we show the structure of the YAML you need to insert. Note that the first line is not indented at all:

```
registry:
  # below is the content of the JSON file downloaded earlier for the container␣
↪registry from Service Accounts
  # it will look something like the following (with actual values instead of empty␣
↪strings)
  # paste the content after `password: |` below
  password: |
    {
    "type": "<REPLACE>",
    "project_id": "<REPLACE>",
    "private_key_id": "<REPLACE>",
    "private_key": "<REPLACE>",
    "client_email": "<REPLACE>",
    "client_id": "<REPLACE>",
    "auth_uri": "<REPLACE>",
    "token_uri": "<REPLACE>",
    "auth_provider_x509_cert_url": "<REPLACE>",
    "client_x509_cert_url": "<REPLACE>"
    }
```

Tip:

- The content you put just after `password:  |` must all line up at the same tab level.
- Don't forget the `|` after the `password:` label.

If you are using Docker Hub

Update `secret.yaml` by entering the following:

```
registry:
  username: <docker-id>
  password: <password>
```

Note:

- **"<docker-id>"** and **"<password>"** are your credentials to login to Docker Hub. If you use an organization to store your Docker images, this account must be a member of it.

4.3.3 Create `config.yaml`

Create a file called `config.yaml` and choose the following directions based on the registry you are using.

If you are using `gcr.io`

To configure BinderHub to use `gcr.io`, simply add the following to your `config.yaml` file:

```
registry:
  prefix:  gcr.io/<google-project-id>/<prefix>
  enabled: true
```

Note:

- **"<google-project-id>"** can be found in the JSON file that you pasted above. It is the text that is in the `project_id` field. This is the project *ID*, which may be different from the project *name*.
- **"<prefix>"** can be any string, and will be prepended to image names. We recommend something descriptive such as `binder-dev` or `binder-prod`.

If you are using Docker Hub

Using Docker Hub is slightly more involved as the registry is not being run by the same platform that runs BinderHub.

Update `config.yaml` by entering the following:

```
registry:
  enabled: true
  prefix: <docker-id/organization-name>/<prefix>
  host: https://registry.hub.docker.com
  authHost: https://index.docker.io/v1
  authTokenUrl: https://auth.docker.io/token?service=registry.docker.io
```

Note:

- **"<docker-id/organization-name>"** is where you want to store Docker images. This can be your Docker ID account or an organization that your account belongs to.
- **"<prefix>"** can be any string, and will be prepended to image names. We recommend something descriptive such as `binder-dev` or `binder-prod`.

4.3.4 Install BinderHub

First, get the latest helm chart for BinderHub.:

```
helm repo add jupyterhub https://jupyterhub.github.io/helm-chart
helm repo update
```

Next, **install the Helm Chart** using the configuration files that you've just created. Do this by running the following command:

```
helm install jupyterhub/binderhub --version=0.1.0-...  --name=<choose-name> --
↪namespace=<choose-namespace> -f secret.yaml -f config.yaml
```

Note:

- `--version` refers to the version of the BinderHub **Helm Chart**. Available versions can be found here.
- `name` and `namespace` may be different, but we recommend using the same `name` and `namespace` to avoid confusion. We recommend something descriptive and short, such as `binder`.
- If you run `kubectl get pod --namespace=<namespace-from-above>` you may notice the binder pod in `CrashLoopBackoff`. This is expected, and will be resolved in the next section.

This installation step will deploy both a BinderHub and a JupyterHub, but they are not yet set up to communicate with each other. We'll fix this in the next step. Wait a few moments before moving on as the resources may take a few minutes to be set up.

4.3.5 Connect BinderHub and JupyterHub

In the google console, run the following command to print the IP address of the JupyterHub we just deployed.:

```
kubectl --namespace=<namespace-from-above> get svc proxy-public
```

Copy the IP address under `EXTERNAL-IP`. This is the IP of your JupyterHub. Now, add the following lines to `config.yaml` file:

```
hub:
  url: http://<IP in EXTERNAL-IP>
```

Next, upgrade the helm chart to deploy this change:

```
helm upgrade <name-from-above> jupyterhub/binderhub --version=v0.1.0-85ac189  -f␣
↪secret.yaml -f config.yaml
```

4.3.6 Try out your BinderHub Deployment

If the `helm upgrade` command above succeeds, it's time to try out your BinderHub deployment.

First, find the IP address of the BinderHub deployment by running the following command:

```
kubectl --namespace=<namespace-from-above> get svc binder
```

Note the IP address in `EXTERNAL-IP`. This is your BinderHub IP address. Type this IP address in your browser and a BinderHub should be waiting there for you.

You now have a functioning BinderHub at the above IP address.

4.3.7 Increase your GitHub API limit

Note: Increasing the GitHub API limit is not strictly required, but is recommended before sharing your BinderHub URL with users.

By default GitHub only lets you make 60 requests each hour. If you expect your users to serve repositories hosted on GitHub, we recommend creating an API access token to raise your API limit to 5000 requests an hour.

1. Create a new token with default (check no boxes) permissions here.
2. Store your new token somewhere secure (e.g. keychain, netrc, etc.)
3. Update `secret.yaml` by entering the following:

```
github:
  accessToken: <insert_token_value_here>
```

This value will be loaded into `GITHUB_ACCESS_TOKEN` environment variable and BinderHub will automatically use the token stored in this variable when making API requests to GitHub. See the GitHub authentication documentation for more information about API limits.

For next steps, see *Debugging BinderHub* and *Tear down your Binder deployment*.

4.4 Tear down your Binder deployment

Deconstructing a Binder deployment can be a little bit confusing because users may have caused new cloud containers to be created. It is important to remember to delete each of these containers or else they will continue to exist (and cost money!).

4.4.1 Contracting the size of your cluster

If you would like to shrink the size of your cluster, refer to the Expanding and contracting the size of your cluster section of the Zero to JupyterHub documentation. Resizing the cluster to zero nodes could be used if you wish to temporarily reduce the cluster (and save costs) without deleting the cluster.

4.4.2 Deleting the cluster

To delete a Binder cluster, follow the instructions in the Turning Off JupyterHub and Computational Resources section of the Zero to JupyterHub documentation.

Important: Double check your cloud provider account to make sure all resources have been deleted as expected. Double checking is a good practice and will help prevent unwanted charges.

CHAPTER 5

Customization and more information

5.1 The BinderHub Architecture

This page provides a high-level overview of the technical pieces that make up a BinderHub deployment.

5.1.1 Tools used by BinderHub

BinderHub connects several services together to provide on-the-fly creation and registry of Docker images. It utilizes the following tools:

- A **cloud provider** such Google Cloud, Microsoft Azure, Amazon EC2, and others
- **Kubernetes** to manage resources on the cloud
- **Helm** to configure and control Kubernetes
- **Docker** to use containers that standardize computing environments
- A **BinderHub UI** that users can access to specify GitHub repos they want built
- **BinderHub** to generate Docker images using the URL of a GitHub repository
- A **Docker registry** (such as gcr.io) that hosts container images
- **JupyterHub** to deploy temporary containers for users

5.1.2 What happens when a user clicks a Binder link?

After a user clicks a Binder link, the following chain of events happens:

1. BinderHub resolves the link to the repository.
2. BinderHub determines whether a Docker image already exists for the repository at the latest `ref` (git commit hash, branch, or tag).
3. **If the image doesn't exist**, BinderHub creates a `build` pod that uses repo2docker to do the following:

- Fetch the repository associated with the link
- Build a Docker container image containing the environment specified in configuration files in the repository.
- Push that image to a Docker registry, and send the registry information to the BinderHub for future reference.

4. BinderHub sends the Docker image registry to **JupyterHub**.
5. JupyterHub creates a Kubernetes pod for the user that serves the built Docker image for the repository.
6. JupyterHub monitors the user's pod for activity, and destroys it after a short period of inactivity.

5.1.3 A diagram of the BinderHub architecture

Here is a high-level overview of the components that make up BinderHub.

5.2 Debugging BinderHub

If BinderHub isn't behaving as you'd expect, you'll need to debug your kubernetes deployment of the JupyterHub and BinderHub services. For a guide on how to debug in Kubernetes, see the Zero to JupyterHub debugging guide.

5.2.1 Changing the helm chart

If you make changes to your Helm Chart (e.g., while debugging), you should run an upgrade on your Kubernetes deployment like so:

```
helm upgrade binder jupyterhub/binderhub --version=v0.1.0-397eb59 -f secret.yaml -f␣
↪config.yaml
```

5.3 Customizing your BinderHub deployment

Because BinderHub uses JupyterHub to manage all user sessions, you can customize many aspects of the resources available to the user. This is primarily done by modifications to your BinderHub's Helm chart (`config.yaml`).

To make edits to your JupyterHub deplyoment via `config.yaml`, use the following pattern:

```
binderhub:
   jupyterhub:
      <JUPYTERHUB-CONFIG-YAML>
```

For example, see this section of the mybinder.org Helm Chart.

For information on how to configure your JupyterHub deployment, see the JupyterHub for Kubernetes Customization Guide.

5.4 BinderHub API Documentation

5.4.1 Endpoint

There's one API endpoint, which is:

```
/build/<provider>/<spec>
```

Even though it says **build** it is actually performs **launch**.

5.4.2 Provider

Provider is a supported provider, and **spec** is the specification for the given provider.

Currently supported providers and their specs are:

Provider	prefix	spec	notes
GitHub	`gh`	`<user>/<repo>/<commit-sha-or-tag-or-branch>`	
Git	`git`	`<url-escaped-url>/<commit-sha>`	arbitrary HTTP git repos
GitLab	`gl`	`<url-escaped-namespace>/<commit-sha-or-tag-or-branch>`	

Next, construct an appropriate URL and send a request.

You'll get back an Event Stream. It's pretty much just a long-lived HTTP connection with a well known JSON based data protocol. It's one-way communication only (server to client) and is straightforward to implement across multiple languages.

When the request is received, the following happens:

1. Check if this image exists in our cached image registry. If so, launch it.
2. If it doesn't exist in the image registry, we check if a build is currently running. If it is, we attach to it and start streaming logs from it to the user.
3. If there is no build in progress, we start a build and start streaming logs from it to the user.
4. If the build succeeds, we contact the JupyterHub API and start launching the server.

5.4.3 Events

This section catalogs the different events you might receive.

Failed

Emitted whenever a build or launch fails. You *must* close your EventStream when you receive this event.

```
{'phase': 'failed', 'message': 'Reason for failure'}
```

Built

Emitted after the image has been built, before launching begins. This is emitted in the start if the image has been found in the cache registry, or after build completes successfully if we had to do a build.

```
{'phase': 'built', 'message': 'Human readable message', 'imageName': 'Full name of␣
↪the image that is in the cached docker registry'}
```

Note that clients shouldn't rely on the imageName field for anything specific. It should be considered an internal implementation detail.

Waiting

Emitted when we started a build pod and are waiting for it to start.

```
{'phase': 'waiting', 'message': 'Human readable message'}
```

Building

Emitted during the actual building process. Direct stream of logs from the build pod from repo2docker, in the same form as logs from a normal docker build.

```
{'phase': 'building', 'message': 'Log message'}
```

Fetching

Emitted when fetching the repository to be built from its source (GitHub, GitLab, wherever).

```
{'phase': 'fetching', 'message': 'log messages from fetching process'}
```

Pushing

Emitted when the image is being pushed to the cache registry. This provides structured status info that could be in a progressbar. It's structured similar to the output of docker push.

```
{'phase': 'pushing', 'message': 'Human readable message', 'progress': {'layer1': {
↪'current': <bytes-pushed>, 'total': <full-bytes>}, 'layer2': {'current': <bytes-
↪pushed>, 'total': <full-bytes>}, 'layer3': "Pushed", 'layer4': 'Layer already exists
↪'}}
```

Launching

When the repo has been built, and we're in the process of waiting for the hub to launch. This could end up succeeding and emitting a 'ready' event or failing and emitting a 'failure' event.

```
{'phase': 'launching', 'message': 'user friendly message'}
```

Ready

When your notebook is ready! You get a endpoint URL and a token used to access it. You can access the notebook / API by using the token in one of the ways the notebook accepts security tokens

```
{"phase": "ready", "message": "Human readable message", "url": "full-url-of-notebook-
↪server", "token": "notebook-server-token"}
```

5.4.4 Heartbeat

In EventSource, all lines beginning with `:` are considered comments. We send a `:heartbeat` every 30s to make sure that we can pass through proxies without our request being killed.

5.5 BinderHub Deployments

BinderHub is open-source technology that can be deployed anywhere that Kubernetes is deployed. The Binder community hopes that it will be used for many applications in research and education. As new organizations adopt BinderHub, we'll update this page in order to provide inspiration to others who wish to do so.

If you or your organization has set up a BinderHub that isn't listed here, please open an issue on our GitHub repository to discuss adding it!

5.5.1 GESIS - Leibniz-Institute for the Social Sciences

Deployed on bare-metal using `kubeadm`.

- Deployment repository
- BinderHub / JupyterHub links

5.5.2 Pangeo - A community platform for big data geoscience

Pangeo-Binder allows users to perform computations using distributed computing resources via the *dask-kubernetes* package. Read more about the *Pangeo project here*. Pangeo-Binder is deployed on Google Cloud Platform using Google Kubernetes Engine (GKE).

dask-kubernetes: https://dask-kubernetes.readthedocs.io/en/latest/ Pangeo project here: https://pangeo.io/

- Deployment repository
- BinderHub / JupyterHub links
- Pangeo-Binder documentation

5.6 Event Logging

Events are discrete & structured items emitted by BinderHub when specific events happen. For example, the `binderhub.jupyter.org/launch` event is emitted whenever a Launch succeeds.

These events may be sent to a *sink* via handlers from the python `logging` module.

5.6.1 Events vs Metrics

BinderHub also exposes prometheus metrics. These are pre-aggregated, and extremely limited in scope. They can efficiently answer questions like 'how many launches happened in the last hour?' but not questions like 'how many times was this repo launched in the last 6 months?'. Events are discrete and can be aggregated in many ways during analysis. Metrics are aggregated at source, and this limits what can be done with them during analysis. Metrics are mostly operational, while events are for analytics.

5.6.2 What events to emit?

Since events have a lot more information than metrics do, we should be careful about what events we emit. In general, we should pose an **explicit question** that events can answer.

For example, to answer the question *How many times has my GitHub repo been launched in the last 6 months?*, we would need to emit an event every time a launch succeeds. To answer the question *how long did users spend on my repo?*, we would need to emit an event every time a user notebook is killed, along with the lifetime length of the notebook.

Wikimedia's EventLogging Guidelines contain a lot of useful info on how to approach adding more events.

5.7 Configuration and Source Code Reference

5.7.1 app

Module: `binderhub.app`

`BinderHub`

5.7.2 build

Module: `binderhub.build`

Contains build of a docker image from a git repository.

`Build`

class binderhub.build.**Build**(*q*, *api*, *name*, *namespace*, *repo_url*, *ref*, *git_credentials*, *builder_image*, *image_name*, *push_secret*, *memory_limit*, *docker_host*, *node_selector*, *appendix=''*, *log_tail_lines=100*)

Represents a build of a git repository into a docker image.

This ultimately maps to a single pod on a kubernetes cluster. Many different build objects can point to this single pod and perform operations on the pod. The code in this class needs to be careful and take this into account.

For example, operations a Build object tries might not succeed because another Build object pointing to the same pod might have done something else. This should be handled gracefully, and the build object should reflect the state of the pod as quickly as possible.

name The `name` should be unique and immutable since it is used to sync to the pod. The `name` should be unique for a `(repo_url, ref)` tuple, and the same tuple should correspond to the same `name`. This allows use of the locking provided by k8s API instead of having to invent our own locking code.

cleanup()
Delete a kubernetes pod.

classmethod cleanup_builds(*kube*, *namespace*, *max_age*)
Delete stopped build pods and build pods that have aged out

get_cmd()
Get the cmd to run to build the image

progress(*kind*, *obj*)
Put the current action item into the queue for execution.

stop()
Stop watching a build

stream_logs()
Stream a pod's logs

submit()
Submit a image spec to openshift's s2i and wait for completion

5.7.3 builder

Module: binderhub.builder

BuildHandler

5.7.4 main

Module: binderhub.main

MainHandler

ParameterizedMainHandler

LegacyRedirectHandler

5.7.5 registry

binderhub.repoproviders

Module: binderhub.registry

Interaction with the Docker Registry

DockerRegistry

class binderhub.registry.**DockerRegistry**(*auth_host*, *auth_token_url*, *registry_host*)

5.7.6 repoproviders

Module: binderhub.repoproviders

Classes for Repo providers

Subclass the base class, `RepoProvider`, to support different version control services and providers.

RepoProvider

class binderhub.repoproviders.**RepoProvider**(***kwargs*)
Base class for a repo provider

config c.RepoProvider.banned_specs = List()
List of specs to blacklist building.

Should be a list of regexes (not regex objects) that match specs which should be blacklisted

config c.RepoProvider.banned_specs = List()
List of specs to blacklist building.

Should be a list of regexes (not regex objects) that match specs which should be blacklisted

get_build_slug()
Return a unique build slug

get_repo_url()
Return the git clone-able repo URL

is_banned()
Return true if the given spec has been banned

GitHubRepoProvider

class binderhub.repoproviders.**GitHubRepoProvider**(**args*, ***kwargs*)
Repo provider for the GitHub service

config c.GitHubRepoProvider.access_token = Unicode('')
GitHub access token for authentication with the GitHub API

Loaded from GITHUB_ACCESS_TOKEN env by default.

config c.GitHubRepoProvider.banned_specs = List()
List of specs to blacklist building.

Should be a list of regexes (not regex objects) that match specs which should be blacklisted

config c.GitHubRepoProvider.client_id = Unicode('')
GitHub client id for authentication with the GitHub API

For use with client_secret. Loaded from GITHUB_CLIENT_ID env by default.

config c.GitHubRepoProvider.client_secret = Unicode('')
GitHub client secret for authentication with the GitHub API

For use with client_id. Loaded from GITHUB_CLIENT_SECRET env by default.

config c.GitHubRepoProvider.hostname = Unicode('github.com')
The GitHub hostname to use

Only necessary if not github.com, e.g. GitHub Enterprise.

config c.GitHubRepoProvider.access_token = Unicode('')
GitHub access token for authentication with the GitHub API

Loaded from GITHUB_ACCESS_TOKEN env by default.

config c.GitHubRepoProvider.client_id = Unicode('')
GitHub client id for authentication with the GitHub API

For use with client_secret. Loaded from GITHUB_CLIENT_ID env by default.

config c.GitHubRepoProvider.client_secret = Unicode('')
GitHub client secret for authentication with the GitHub API

For use with client_id. Loaded from GITHUB_CLIENT_SECRET env by default.

get_build_slug()
Return a unique build slug

get_repo_url()
Return the git clone-able repo URL

config c.GitHubRepoProvider.hostname = Unicode('github.com')
The GitHub hostname to use

Only necessary if not github.com, e.g. GitHub Enterprise.

Python Module Index

b

Index