# Docker Bakery Management System


## Poojan Shah

## Mentor: Vinayak Trivedi | Manager: Yoginkumar Patel


# Code Documentation

### Functions from functions.py:

1) **create_docker_graph:**

This function takes in three parameters: array_index (the index of the root image node in the image_list), image_list (a list of image nodes), and node_id_to_array_index (a dictionary that maps the ID of an image node in the MongoDB database to its index in the image list).

The function creates a hierarchy by recursively appending the child image node object inside the parent image node's children array. It stores the child image nodes in sorted order based on their name. The function returns the image_list in the hierarchical format. For further clarification, please refer to the /static/data.json file in the project directory.

This is then used by JavaScript in the UI to create the hierarchy for visualization.

2) **parse_script:**

This function takes the “build-component.sh” shell script as input and parses it to return a dictionary that maps the component name with the path of its associated Dockerfile in the repository. The component name and Dockerfile path are identified using regular expressions.

3) **parse_dockerfile:**

This function takes Dockerfile contents as input and parses it to find the name of the parent image, as well as the list of repository paths for "requirements.txt" files needed to build that image. The function returns the parent image name along with its tag, as well as the list of paths for the "requirements.txt" files.

**The function handles multi-stage Docker builds by extracting the parent name from the last line containing the "FROM" statement.**

4) **build_image:**

This function takes a dictionary as input and passes it as an argument to the API to build a Docker image.

If the image belongs to a repository, build_component_image_api is called. Otherwise, build_non_component_image_api is called.

The API returns a dictionary that includes a job_id, which is used by poll_for_docker_build function to check the status of the Docker build process executing on the API server.

In case of a connection failure with the API server, the function handles the exception.

It receives the status (success or failure) of the Docker build process from the poll_for_docker_build process and returns it.

5) **poll_for_docker_build:**

The function takes as input the image name and job ID and utilizes it to invoke the "poll_build_image_api" function.

This API provides the status of the Docker build process, which can be categorized as "Success," "Failed," or "In Progress."

The function continuously calls the API every 5 seconds until it receives a status of either "Success" or "Failed."

6) **delete_image:**

The function takes a dictionary parameter that includes the image name and tag. It proceeds to invoke the "delete_node_api" function, passing the dictionary parameter as an argument. It then checks whether the status code of the API response is 200 or not. If the status code is not 200, it indicates a failure in deleting the image.

The API response is in the form of a dictionary, which contains the status of the delete-image request. In the event that the API server is down or there is a failure in establishing a connection, the function handles the exception gracefully.

Overall, the function ensures that the image deletion process is carried out properly by leveraging the delete_node_api, checking the status code of the response, and handling potential exceptions related to server connectivity.

7) **should_rebuild_image_and_sync:**

The function accepts two inputs: the image node to be synced and the repository name to which the image belongs. Its purpose is to determine whether the last sync time of the image node is earlier than the last commit time of either the Dockerfile or the requirements.txt files.

To do this, the function compares the last sync time of the image node with the last commit times of the Dockerfile and requirements.txt files. If the last sync time is indeed earlier than either of the last commit times, the function returns True. Otherwise, the function returns False.

Essentially, the function helps identify whether the image node requires syncing based on the comparison of sync and commit timestamps for the associated Dockerfile and requirements.txt files.

8) **get_parameters:**

This function takes input the image node and returns the parameters that will be passed to the API to build the image.

If the image is built from a repository, the parameters: repo_name, dockerfile, component_name, tag, repo path of the dockerfile are returned.If the image is not built from repository, then the parameters: dockerfile, image name, tag and requirements.txt file are returned.

9) **sync_upgrade_node:**

This function takes the input an image node and the repo_name of the image of which it is a part of.First it checks whether the image is can be synced or not by checking whether any of its ancestor is being processed currently by some other user.

If yes we cannot sync the image now and throw an error. This is to avoid the race condition.

If the image can be synced, then it is checked whether should we rebuild the image by comparing the last-sync-time with the last commit time using the function should_rebuild_image_and_sync function.

If yes, the image is first locked to avoid race condition, then a new image is created which is the copy of the old image node but with different tag and updated dockerfile and requirements.txt content.

Then it parses the dockerfile and checks whether the parent of the image has changed. If it has changed, then we first check whether the changed parent image exists or not.

If it exists then the newly created image node is assigned as child to the new parent node.

Then the image is rebuilt. If the status of rebuild of image is success, then its deployments are re-deployed and then its child images are rebuilt recursively using dfs_upgrade_node function. At the end the lock is released to avoid deadlock.

10) **can_process_node:**

This function takes input the image node, and checks whether itself or any of its ancestor till the root node, are already locked. If YES, then we cannot process the node now and we will return False & image name of the node which is locked. Else if will return True.

11) **sync_update_node:**

This function takes the input an image node and the repo_name of the image of which it is a part of.

First it checks whether the image is can be synced or not by checking whether any of its ancestor is being processed currently by some other user.

If yes we cannot sync the image now and throw an error. This is to avoid the race condition.

If the image can be synced, then it is checked whether should we rebuild the image by comparing the last-sync-time with the last commit time using the function should_rebuild_image_and_sync function.

It parses the dockerfile and checks whether the parent of the image has changed. If it has changed, then we first check whether the changed parent image exists or not.

If it exists then the image node is shifted from old parent to the new parent node.

Then the node is locked so that other user cannot update/edit/delete it until the sync_update_function doesn't complete its execution

Then the dockerfile and requirements.txt files are fetched from repo and the image is rebuilt.

If the status of rebuild of image is success, then its deployments are re-deployed and then its child images are rebuilt asynchronously.

After rebuilding the child images asynchronously, dfs_update_node function is called for each child node. At the end the lock is released to avoid deadlock.

12) **add_new_component:**

The function takes input the repo_name and a dictionary component_name_to_dockerfile_path which maps the component-name to its associated dockerfile path.

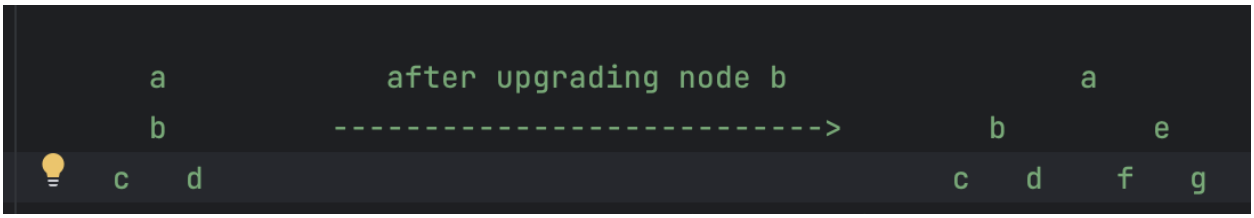
It checks whether there exists any component whose image does not already exist.

If Yes, it builds its image and stores its meta-data in the mongoDB database.

Also, takes care of the edge cases like if the parent node of that image does not exist. In such case the component cannot be built.

13) **dfs_upgrade_node:**

The function takes three arguments: one is the old_child_id --> id of the child node before upgrading the image, old_parent_node --> parent node of the old child, upgraded_parent_node --> upgraded version of the old_parent_node




When we upgrade node b, sibling node(node -> e) will be created and similarly its subtree will be created recursively.

I am just making a copy of subtree rooted at b by mapping the nodes of subtree of b with the nodes of subtree of its sibling(e).

So the new subtree is build in the following order, b is mapped to e -> c is mapped to f -> d is mapped to g.

Basically it's just DFS. I am also maintaining a dictionary(old_img_to_new_img) to establish the siblings relationship between mapped nodes.

The function takes care of the case, when the parent in the updated dockerfile does not exist.

14) **autosync_update_node:**

The function takes the input the name of the repository that is to be synced.

The function first adds any new component that is present in the repository but not present in our image hierarchy using add_new_component function.

Then the sync_update_node function is called for each component in the repository.

15) **delete_subtree:**

This function takes input the node_id of the image node whose subtree the user wants to delete.

It recursively deletes the images in the subtree of the node user wants to delete.

Also, when the image is deleted, the deployments that were made using that image are re-deployed using the latest version of that image with the help of redeploy_using_latest_img function.

16) **redeploy_using_latest_img:**

This function is called when the User wants to delete an image and its deployments are to be re-deployed using the latest version of that image.

The function takes input the deployments_list that were made using the image the user wants to delete.

Also, it takes input the image name. The function finds the image with same image name and with latest tag and re-deploy the deployments_list using that version of image.

17) **replace_document:**

This function takes input an image-node and a dictionary: image_name_to_deployments_list and updates the list of deployments using an image in the mongo-db database.

18) **get_k8s_deployments:**

This function calls the API **get_deployments** and passes the argument: a dictionary with env as prod.

The API returns a dictionary which that maps the image name to the list of deployments that were made using that image in the **env: prod**

Using the dictionary returned by API, the function updates the data in the database using ThreadPoolExecutor which helps to speed-up the process of updating it in the database.

The function also does the required exception handling if the connection to API server fails.

19) **make_redeployments:**

The function takes input the image-node whose deployments are to be re-deployed.

It calls an API **deploy_component** by passing the required parameters.

The API re-deploys the deployments using Helm-Chart.

The API returns a dictionary containing the status of the re-deployment of the deployment.

The function returns the status of the re-deployment.

20) **get_data_from_repository:**

This function takes input the repository object which is used to access the GitHub repository and the repository path of the file.

Using the provided repository object, the function accesses the specified file using its repository path. It retrieves the contents of the file and converts it into a string format. Finally, the function returns the file contents as a string.

21) **update_parent_in_dockerfile:**

This function takes input child image node and its parent image node.

It updates the parent-image name in the dockerfile of the child image node by parsing the dockerfile.

The function also takes care of the multi-stage docker build by updating the last line containing 'FROM' statement.Returns the updated dockerfile in string format.

22) **get_parameters_from_treenode:**

This function takes a image node of treenode object type and returns the parameters that are required to build image.

For the image that is a part of a repository the parameters are: repo_name, dockerfile content, dockerfile_path, component_name and tag.

For the image that is not a part of a repository the parameters are: dockerfile content, img_name, tag and requirements.txt content.

23) **get_dependencies:**

The function takes input a repository object which is used to access the repository and the list of paths of requirements.txt

It returns a list of dictionary containing dependency_name, dependency_repo_path and content of that file.

24) **async_build_image:**

This function takes input an aiohttp session and a dictionary parameter returned by get_parameters function.

The If the image has a repository of which it is a part of, then build_component_image_api is called else, build_non_component_image_api is called.

The API returns a dictionary which contains a job_id, which is used by the poll_for_docker_build function to poll and get the status of the docker build process getting executed on the API server.

The function along handles the exception when the connection to the API server fails.

This functions takes the status(success or failed) of the docker build process from the poll_for_docker_build process and returns it.

25) **multiple_docker_build:**

This function takes argument a list of node-ids whose image that can to be rebuilt asynchronously.

It creates a list of schedules of build-image tasks that will be executed asynchronously.

All these tasks are executed asynchronously so as to utilize the time when the server is waiting for the response from the API server.

It returns a dictionary node_id_to_status which maps the node-id to the status of the docker build of that image.

26) **dfs_update_node:**

This function takes input a node_id, a dictionary node_id_to_status and a boolean variable to_update_last_sync_time.
If the re-build status of the node is Success, then only the function will redeploy the deployments that were made using that image and it will recursively call the same function for its child images.
For the child images, the images are first re-built asynchronously and then the dfs_update_node function is called recursively for child images.

27) **update_dictionary:**

The update_dictionary function takes two parameters: dictionary1 and dictionary2 and recursively overwrites the value of a key that is present in both the dictionary in dictionary1 from dictionary2.

It takes care of the fact that the value of a key itself can be a dictionary. For this purpose the function is called recursively when a list or a dictionary is encountered.

28) **get_deployments_from_helm_repo:**

This function parses the helm-repository and makes a map of image-name to list of deployments using that image.
The function creates a dictionary of the values.yaml file present in the root directory of the helm repository.
Then it goes inside the releases folder of the repo and parses all the releases directories.
For each release, it takes its values.yaml file and overwrites the values.yaml file present in the root directory of the repository. Then it goes inside the env directory, takes its values.yaml file and overwrites the values.yaml file present in the root directory.
It then creates a dictionary, which has image name as key and list of dictionary containing deployment names and the environment in which they were deployed.
It then stores the dictionary for each image in the mongoDB database

29) **can_be_deleted:**

This function takes input a node that the user wants to delete and then checks in the subtree of that node, that whether there is any sibling image of each node using which the deployments can be re-deployed. If yes then when the node(image) is deleted, its deployments are redeployed using the latest version of that image.
If an image node does not contain any sibling image, then the subtree cannot be deleted as it still has some deployments running.

## Functions from views.py:

1) **get_data_from_mongodb:**

This function brings the data of all the nodes from the mongoDB database, converts them to hierarchical format using make_docker_graph function from functions.py  and

then instead of storing the sibling's id we replace it with sibling's name and tag and store the data in a file ./static/data.json.

The data for the visualization is taken from this file.

2) **homepage_view:**

This function first creates a list of image names that are associated with some repository, by fetching data from mongoDB.
This list of images are the ones, the user can sync.
Then it calls the get_data_from_mongodb function, which will store the data in hierarchical format in ./static/data.json file.
Then it renders the homepage and passes the list of images as parameter.

3) **add_child_view:**

This function takes argument parent_id from the UI and renders a html form to add new child node.

4) **add_child_component:**

If the request method is POST, then the function will receive the following inputs: component_name, parent_id, tag and repo_name and builds a new component image and stores its metadata in the mongoDB.

The function takes care of multiple edge-cases like if Image already exists, if the tag is incremental or not, if the repository exist or not, does the component exist or not, does the parent image mentioned in the dockerfile matches the  with the one he/she wants to make this new node child of.

5) **add_child_image:**

**This function is invoked when the user adds a child image by entering the dockerfile and requirements.txt in the form to add a new node.**

The function takes input: parent_id, dockerfile content, requirements.txt content if entered by user and then it builds this new non_component image and store its metadata in the database.
The function takes care of multiple edge-cases like if Image already exists, if the tag is incremental or not, is the dockerfile empty, if the user has specified the parent image in the dockerfile or not or if the parent name matches with the one he/she wants to make this new node child of.
in this function, it is assumed that dockerfile and requirements.txt are in the same path

6) **build_new_image_util:**

This function takes two arguments a node and its parent node.
It first locks the node, and then builds its docker image using build_image function.
then if the status of build image is Success, then it stores its metadata in mongoDB and returns the status Success or Failed depending upon whether the image is created successfully or Failed.

7) **manual_sync_on_image:**

The function takes input the image name. If the node does not contains any repo then we give an error that we can't sync the image without repository.
The function searches for the node with the same name but with the highest tag.

If the sync_type == 0, it means that the user wants to upgrade the node, else the user wants to update the node.

The respective parameters are passed to the function based on the type of sync the user wants. The return_code will be a single integer if the image is synced successfully or Failed. However, when the sync operation is denied because some of its ancestor is already locked and is under process, then the return value will be a tuple with the 1st element the return code **2** and the 2nd element: the image name which is already locked.

8) **manual_sync_on_node: Manual sync by right clicking on a Node**

The function takes the input of a node_id, representing the node that the user intends to sync. If the node does not have a repository associated with it, an error is thrown indicating that the image cannot be synced without a repository.

The function proceeds to search for a node with the same name but with the highest tag in MongoDB. This is done to identify the most recent version of the node for syncing purposes.

If the sync_type is 0, it signifies that the user intends to upgrade the node. Alternatively, if the sync_type is any other value, it indicates that the user wants to update the node. Based on the type of sync requested, the respective parameters are passed to the relevant functions.

The return_code from the sync operation will be a single integer indicating the success or failure of the image sync process. However, if the sync operation is denied because one of its ancestors is already locked and currently being processed, the return value will be a tuple. The tuple will have a return code of 2 as the first element and the name of the locked image as the second element.

9) **edit_node:**

If the request method is GET, then the function returns the node_id and the content of the dockerfile to the user.
If the request method is POST, then the function takes input the node_id and the updated content of the dockerfile from the user.
It first checks whether the node can be processed or not, by checking whether any of its ancestor are locked or not.
If the node can be processed, then we first apply the lock on the node and rebuild its image. If the image is built successfully, then it will redeploy the deployments that were made using that node on k8s. Then it will re-build the images of its children asynchronously using dfs_update_node function.

10) **delete_node:**

The function first calls the get_k8s_deployments function which will store the metadata about the list of deployments that are made using each image node.
Then we check whether the image node can be deleted or not. In case the image node itself or some of its ancestor node is already locked, then we cannot delete the node as it is already undergoing some process.
If the can_be_deleted function returns true, then it is first locked to avoid the race condition. Then delete_subtree function is called which deletes the subtree recursively and alongside makes re-deployments using the latest-version of the deleted image.
At last the lock is released so as to avoid deadlock.

11) **if_parent_matches:**

This function takes input the parent_node_id and the Dockerfile content.
It parses the docker file and checks whether the parent image in the dockerfile matches the one selected by user using UI.
returns true if the parent matches else return False.

12) **make_new_component_node:**

This function is used when the image the user wants to add is a part of some repository.
This function takes input repository object which is used to access the repository, repo_name, image name, tag, parent, component_name and repository path of dockerfile
Using this parameters it creates a new treenode object and stores the metadata as well as the parameters passed to the function.
Returns a treenode object --> a new image node.

13) **make_new_non_component_node:**

The function is designed to handle cases where the image the user wants to add is not associated with any repository. It accepts several input parameters including the image name, tag, parent_id, dockerfile content, and requirements.txt content.

Using these parameters, the function creates a new treenode object. This treenode object represents the new image node that will be added. The function stores the relevant metadata and the parameters passed to it within this treenode object.

Finally, the function returns the treenode object, which encapsulates all the necessary information about the new image node that has been created.