Waldur_core:

Admin – This includes functions and classes for code such as password widgets, json widgets, optional choice fields, forms with their errors message and attributes, project roles, and customer roles.

Apps – This includes getting the user and public ssh key. The dispatch uuid is taken and used for connecting, saving, and deleting.

Authentication.py – This includes getting the token key for authentication and authenticating the user.

Schemas.py – This gets the validators descriptions, action permissions, fields such as Boolean, char, timestamp, datetime, URL, number, uuid, and ip address and email, returns possible field values, and then waldur schema generator. This generates paths, methods, and views for the paths.

Init.py – contains a get version function to get the mastermind distribution version. This finds the repository directory and finds the path. It searches for the description and name.

Waldur_azure:

Locale – Code for translating to a different language. Each folder includes a Django.po file with metadata and message id and str for models.py, serialisers.py, and views.py.

Management – import_azure_image.py

Creates the class called Command that inherits the methods and attributes of BaseCommand.

Function add_arguments:

Adds arguments to the parser module, allowing python to edit the parse tree and create executable code.

Function handle:

For loop goes through the settings of the object type of Azure. It then updates or creates attributes settings, name of type offer, sku of type sku, publisher of type publisher, and version of latest.

Migrations

0001 initial.py – This file creates the model using the createmodel function for azureservice. It provides the fields id, and autofields the primary key, auto create, serialization, and verbose name (of ID). It also creates a foreign key for the ‘customer’ account. It creates the model for AzureServiceProjectLink with the project and service foreign keys. It creates the model for an ‘image’ with an id and name. It creates the model for an InstanceEndpoint with id, localport, publicport, protocol(tcp or udp), and name. It creates the model for a 'VirtualMachine' with id, created, modified, description, name, runtime_state, state, backendid, ram, disk, min-ram, min-disk, user data, public ips, private ips, tags. It adds the fields of instanceendpoint and azureservice.

0002 immutable default json.py –Alters the fields private and public ips to display the list.

0003 redesign.py – Has dependencies depending on the file referred to at the beginning including structure, core, taggit, and waldur_azure. Has a create model for location, including id, name, uuid, latitude, longitude, backendID. Has a create model for network, including id, created, modified, description, uuid, errormessage, runtimestate, state, and backendID. Has a create model for networkinterface, including id, created, modified, description, uuid, errormessage, runtimestate, state, and backend ID. Has a create model for resource group, sql database, sql server, and subnet with the same attributes. Has a create model for Size with id, name, uuid, backend id, max data disk count, memory in mb, number of cores, os disk size in mb, and resource disk size in mb.

The rest of the file adds fields to the database as foreign keys with the option for deleting them.

0004 storageaccount.py – Has a create model for storage account including id, created, modified, description, uuid, runtime state, and backend_id.

0005 ordering.py – Has alter model options for ordering images with the size. This depends on the storage account.

0006 userdata.py – Depends on 0005. Alters the field for virtual machine with the user data described as additional data that will be added to the instance on provisioning.

0007 publicip.py – Has fields (id, created, modified, description, runtime_state, state, name, location, service_project_link, tags). Each one has related attributes. Creates models for public ip.

0008 networkinterface_publicip.py – Has class Migration and has an attribute for operations. This includes model_name, name, field.

0009 public ip resource group.py – Depends on 0008. Adds field for public ip and resource group connecting it to the database and to 'waldur_azure.resourcegroup'.

0010 sql database.py – Has different fields including network, networkinterface, publicip, and virtual machine. Publicip might be a place I have to change. These are all under the alter or remove field functions. This is mainly to change the name or model name of the attribute as well as its length and ensuring it is valid. Any fields related as a foreign key are linked to the database.

00011 sql collation.py – Depends on 0010. Has alter fields for sqldatabase as name charset and collation providing the max length and default encoding.

0012 network security group.py – Creates model for security group with id, created, modified, description, uuid, error message, backend id, name, resource group, and service project link.

0013 publicipaddress.py – Has operations including public ip. There is protocol IPv4 that needs to be changed.

0014 networkinterfaceipaddress.py – Has network interface with IPv4 protocol.

0015 location enabled.py – Has addfield for location.

0016 extend description limits.py – Has alterfields for network, network interface, publicip, resourcegroup, securitygroup, sqldatabase, sqlserver, storageaccount, subnet, and virtual machine.

0017 error traceback.py – Has addfield for the previous attributes.

Tests – factories.py – Has all the classes such as network factory, subnet factory, network interface factory, and public ip factory.

Fixtures.py – Has cached properties used to initialize functions and factories.

Test_service.py – This tests the setup, getting valid payloads, and ensuring valid credentials are validated.

Init.py – Sets the default config.

Admin.py – Registers the vms, azure service, and azure service project link.

Apps.py – Has AzureConfig class that has a function that readies the service.

Backend.py – This has the class for AzureBackend. It includes pulling services properties, pulling locations, pulling public ips, pulling sizes (cache, backend, new, stale), pulling resource groups, and creating things such as the storage accounts, resource groups, networks, and subnets. Each function configures the self client for mastermind. It has all the functions relating to the changing of all vms, groups, networks, and accounts within mastermind.

Client.py – Includes cached properties for different clients such as subscription, resource, compute, storage, network, and pgsql. These return credentials and subscription ids related to the property. The file also includes functions such as deleting resource groups, listing virtual machines, more virtual machine setting configurations, getting virtual machines, creating storage accounts, disks, networks, and subnets. Each function connects to the post gresql database for the client specifically. Connects the creation of the client with the views and credentials for the client. SSH rules are configured within the security groups and bound to the client. Creation of things such as sql database, sql firewall rules, sql server, security groups, network interfaces, and storage accounts with virtual machines are set with default configurations and certain configurations set by the user.

Executors.py – Contains executors with a series of functions. Executors for virtual machines, public ip, and sql server and database exist that are in charge of starting, deleting, and creating with a special executor at the end for cleanup.

Extensions.py – Contains the AzureExtension class containing functions for the Django_app, rest_urls, and get_cleanup_executor.

Filters.py – Contains the filters including image, location, size, base resource group, virtual machine, public ip, and sql server and database. Each contain a model suited for the class and sets the model for that class.

Models.py – Contains the classes used in filters.py. This includes location, image, size, base resource, resource group, and others such as network, subnets, security groups, virtual machine, public ip, sql server, and sql database. Each contains foreign keys and character fields set allowing the backend function to constrain the length or set functionality for each specific variable.

Serialisers.py – Contains the classes similar to those above including fields for each class. Each class contains the model, viewname, fields. Each model allows a template for the specific class to be created with each serialiser allowing the different sub fields to be initialized.

Urls.py – Creates the routes for each service. This routes to the specific view that is demanded and registers the route only for that specific class.

Validators.py – Contains the lists of valid characters for names and blacklisted words. This compiles to make sure the user entered the correct naming convention and blocks the user from any illegal names.

Views.py – Links each class to the correct view/model/url. Enables objects of the class to be displayed and orders any objects with filters. Uses api for virtual machine view set for displaying if the response was accepted or not. Links the models, filters, serialisers, and executors together.

Openstack plugin

Locale – Contains files for several different languages. Changes the language for admin.py, models.py, serialisers.py, and views.py.

Management – Contains the Command class with the functions for adding arguments, and handle. The function handle allows for pulling of the default security groups, getting the security security groups, and adding security groups to tenants.

Init.py – Sets the configuration for default.

Admin.py – Contains classes for service project link admin, tenant admin form, tenant admin, pull security groups, allocate floating ips, detect external networks, pull floating ips, image admin, network admin, subnet admin, and customer openstack inline. Each contains actions or fields executed upon the class. Classes are also linked to the executors for going to the correct method to do once chosen.

Apps.py – Contains the class OpenStackConfig. It includes functions to get the models of a tenant, network, subnet, security group, and security group rules. Contains for loops for models and resources for connecting the aforementioned groups.

Backend.py – Contains class OpenStackBackend with several functions. Has a function to check the admin tenant for authorization using keystone. It then pulls service properties (flavors, images, volume types, service settings quotes), resource (tenants), and pulls subresources (security groups, floating ips, networks, subnets, routers, and ports.

Executors.py – Contains the classes for creating, updating, and deleting security groups. These connect to the backend method task method for security groups. A class for tenant creation, import, updating, deletion. All connect to state transition task and backend method task for the serialized tenant and choosing which method to execute. Classes exist for Floating Ips including creation, deletion, pull, pull floating, push quotas, and pull executors. Classes also exist for Network creation, update, delete, and pull executors as well as for subnets. Most connect to the backend method task for the core tasks. Serialised networks, tenants, security groups, and floating ips exist with a relevant state transition.

Extension.py – Contains the class OpenStackExtension. Contains the class settings within for oepnstack containing name, description, rules, protocol, cidr, icmp_type, and icmp_code. Contains static methods for connecting it to the front end.

Filters.py – Contains filter classes for service projects, security groups, floating ips, flavor, images, volume type, routers, ports, network, and subnets.All connect to the URL Filter and connect to the Django filters.

Handlers.py – Has functions removing ssh keys from tenants, logging tenant quota, updating the service settings, and logging for security groups, networks, and subnets. Uses for loops for ensuring the admin has access for project then going through each tenant to remove the keys.

Log.py – Has functions for logging tenant quotas, routers, security groups, security group rules, networks, subnets, and ports. Each have the same event types available such as created, imported, updated, pulled, deleted, and cleaned.

Models.py – Has classes for service usage aggregator quota, openstack service, quotas, service project link, flavor, image, volume type, security group (including changing backend quotas usage), security group rule, floating ip, tenant, generating usernames, getting access urls, router, network, subnet, port, and customer open stack. Each class links with a foreign key and models object.

Quotas.py – Includes the possible tenant quotas and paths. Includes a function for injecting tenant quotas.

Serialisers.py – Includes initializing and validating all the fields within each class. Includes setting all the possible valid categories and attributes to each variable.

Tasks.py – Includes classes tenant create error task, create success task, and pull quotas. Includes what to in each case of unsuccessful or successful. Links to the models object for tenants, networks, and subnets.

URLs.py – Registers all the routes for each view. Includes the openstack images, volume types, tenants, service project links, security groups, ports, ips, routers, networks, and subnets.

Views.py – Contains classes for linking each view to the needed model, serialiser, and filter. This also includes creating the executor variables for each executor link and exposing the api for valid responses. Includes checking for valid responses and setting the variables to what data is given.

Waldur homeport

.vscode – contains configurations, and version.

Cypress (integration test) – Contains all files including configurations for the marketplace offering. Initialises all the components with empty or null details. Used for testing on the browser of scripts.

Docs – Documentation including configs, development guidelines, i18n, plugins, resource-actions, sidebare-customisation, and terminology policy.

I18n – Contains the locale changes for multi languages.

Dockerfile – Script for building the environment.

Angular-gettext-compiler.js – Contains browser converted html entities. Contains variables for items, ctx, msgid, and entities (converted, unconverted) and uses a for loop for replacing the unconverted entities with the converted ones. Strips context from string that do not have context.

Angular-gettext-plugin.js – Contains function for getting the text plugin via angular. Returns content based on if the options were compile translations or extract strings.

Cypress.json – Contains the base url, project id, and support file.

Docker push.sh – Exports specific version or prefix version and contains commands for building the image file.

Jest.config.js – Part of module.exports and contains transformations with regex.

Package.json – Contains dependencies with required versions aas well as scripts for building homeport.
postcss.config.js – Contains from module.exports plugins.

Svgfonts.fong.js – Contains from module.exports files from images with fontName, classPrefix, base selector, types, fixedwidth, and filename.

Tsconfig.json – Contains compiler options.

Typings.d.ts – Declares module *.json as a constant and exports default value.

Webpack.config.common.js – Contains tests for using loaders, options, publicpaths, and names. Contains declaration of new plugins from views and formats a path for such plugins.

Webpack.config.dev.js – Merges configurations with plugins including context and manifests.

Webpack.config.dll.js – Exports entry vendors and contains outputs from the target path.

Webpack.config.prod.js – Contains requirements for constants (of plugins) including dev tool, mode, optimization, and plugins. (filepath, sourcemap, outputpath, publicpath, and hash).

Webpack.config.test.js – Contains base config and merge webpack with the devtool inline source map.

Webpack.utils.js – Contains constants for path, isprod, target, formatpath, vendor manifest, and vendor bundle.