**How to add a module to E3 Version 1:**

E3 is designed to be modular, allowing for the easy addition of new calculations, input formats, and outputs that extend its default capabilities. While the ability to add new input formats can be useful, if using basic E3 calculations it’s best to make any conversions from a custom format to E3 as an additional step on the GUI side prior to sending it to E3.

This guide will go through the key elements in adding a new input object, output object and calculation module along with an example to illustrate the process.

**Adding an input object:**

All input objects require an associated serializer to allow the conversion of JSON format to and from python objects. Any new input object should be placed in the “e3\_django/API/objects” directory while the associated serializer should be placed in “e3\_django/API/serializers” for organizational structure. It is recommended to maintain the current E3 naming convention for objects (camel case) and serializers (name of object and followed by “serializer”, camel case). The only exception should be supplementary files, such as the “fields” file, which provide additional functionality that is shared across multiple objects or serializers.

The internal structure of any input object is left to user’s discretion. Serializers have a much more rigid structure. The class for the serializer should include the full list of attributes from the associated object with any requirements or restrictions (using the rest\_framework field definitions and any custom field definitions from the “field” file). A validation method should also be included to ensure all inputs are valid for any created object.

For the input object to be properly recognized it needs to be added to the list of inputs in the “Input’ object file and the associated serializer. Edits may need to be made to the “tasks” file as well. If the input file doesn’t make any edits or alterations to existing inputs, then no other changes are required. The additional code in the example is to allow the Edges code to overwrite certain inputs in other input files (these overwrites are done in the “Edges” file) and replace the sensitivity summary with the Edges calculations. An explanation of how E3 knows when and how to use the input file is explained in the section on adding a module.

**Adding an output object:**

Adding an output object is very similar to the process for adding an input object. The primary differences are where the files should be located (“e3\_django/API/compute/objects” and “e3\_django/API/compute/serializers”) and some features in how the output objects may be constructed. The serializers will follow the same rules as the input serializers except there is no need for a validation function as all values should be taken from output that has already been verified by the associated modules. A new output object may end up calling output from other E3 objects, thus it may require additional input in its *init* function. The call to create a new instance will occur in the associate module (see section on how to create a module for more information).

**Adding a module:**

Modules are where the bulk of the calculations for an analysis are performed. As the specifics of adding an analysis are highly dependent on what the user is attempting to do, the process of coding a module will be highly variable. In some cases, the changes may not require a new module at all, instead relying on additions to the base code. An example of this would be adding a new measure to the measure summary calculations that relies on already calculated values. Doing so could be done by properly adding a new attribute to the appropriate class and serializer and a new method in the class file. While additions of the described nature are simple, making direct changes to the base code that alter existing calculations or objects is risky. Doing so without proper version control may result in code that produces incorrect results without the ability to easily return to the original calculations. In general, a new module should be created if base functionality is being overridden.

At present all modules are in the “compute” directory, although there is no hard requirement that this be the case. Each module should have its own directory and should contain the following files:

\_\_init\_\_.py – Usually blank but may have uses in certain situations.

admin.py – Leave blank unless there is some need to create admin level methods.

apps.py – The primary file for module calculations

models.py – The model component of an MVC application.

tests.py – Used to store tests for the module.

views.py – Structure for the interface if the module will have its own page.

The main class within the apps files should be named the module named followed by “Config” with “E3ModuleConfig” as the input. The config class must have the following attributes:

name – A string containing the path from the compute directory to the module folder separated by periods, i.e. “compute.modulename”.

verbose\_name – A string containing a brief plain text name for the module.

depends\_on – A list of other nodes in the graph that the current module pules froms.

output – The name of the output for the module.

serializer – An item that defines the serializer for the module output object.

As the main class for the module the return call should be included here.

For the code to be able to access new modules the module must be added to the “graph” of the app. The graph can be considered a collection of nodes representing specific modules and objects with lines connecting them based on how they interface with each other. This is done in “settings.py”. In the settings code there is an INSTALLED\_APPS list which contains several key components of the app graph. At the end of this list are all the user generated modules. To add a module to the graph the directory path to its “app” file must be added to the list. Assuming that the module is added to the compute folder it would follow the pattern of ‘compute.modulename.apps.ModuleNameConfig’. Remember to separate it from the other items in the list with a comma.

**An example of adding a module: EDGe$**

A module named edges has been developed in order to illustrate the general process of adding a module as well as extend E3’s functionality to perform the calculations performed in NIST’s EDGe$ program. The related files are (all paths relative to “E3/e3\_django”):

“API/objects/Edges.py”

“API/serializers/EdgesSerializer.py”

“compute/edges” – All files in folder

“compute/objects/EdgesSummary.py”

“compute/objects/EdgesSensitivitySummary.py”

Edits were made to the following files:

“API/objects/Input.py”

“API/objects/\_\_init\_\_.py”

“API/objects/InputSerializer.py”

“API/serializers/\_\_init\_\_.py”

“API/tasks.py”

“e3\_django/settings.py”

“compute/objects/\_\_init\_\_.py”

“compute/serializers/\_\_init\_\_.py”

While the new files can be examined in their entirety as new code, the edited files have only certain sections that need to be looked at. For ease of understanding, the screenshots below show the added sections for the edited files along with an explanation of what their purpose is.

Starting with “API/objects/Input.py”. The first addition is “edgesObject=None”. This adds the Edges object to the list of inputs while the “=None” defines it as optional.



Additionally, the Edges object needs to be defined in the “init” method of the “Input.py” file. This is done as follows:



With the Edges object added to the general Input object, the serializer needs to be updated. The first change is to import the Edges object and serializer into the Input serializer as follows.



Note: in order for python to import the object and serializer as shown the “\_\_init\_\_.py” file in “API/objects” needs to have the following line added

from .Edges import Edges

and the similarly in “API/ serializers /\_\_init\_\_.py”:

from .EdgesSerializer import EdgesSerializer

Alternatively, the \_\_init\_\_ files could be left unaltered, and any imports may be done through a full definition from the API root directory (“E3/e3\_django”).

Once the appropriate classes are imported, the InputSerializer class needs to have the Edges serializer defined:



The validated data then needs to be pulled, if It exists:



And added to the Input object (See the “edges” input at line 101):

Text

Description automatically generated

Lastly the function to update the Input object needs to allow for an updated Edges object (line 101).

Text

Description automatically generated

While the above steps are generally required for any new input format, changes to “tasks.py” may differ based on the changes made, if any changes need to be made at all. As such the edits shown below are not prescriptive. The code below looks for Edges in the list of potential outputs, overrides input based on the code in the Edges code and potentially adds in the Edges sensitivity calculations if necessary.



Similar to the input \_\_init\_\_ files, “compute/objects/\_\_init\_\_.py” requires:

from .EdgesSummary import EdgesSummary  
from .EdgesSensitivitySummary import EdgesSensitivitySummary

and “compute/serializers/\_\_init\_\_.py” requires:

from .EdgesSummarySerializer import EdgesSummarySerializer  
from .EdgesSensitivitySummarySerializer import EdgesSensitivitySummarySerializer

Lastly the modules must be added to the application graph (“e3\_django/settings.py”). See lines 88 and 89):

Text

Description automatically generated

Internal testing can be done without setting up a server instance of E3. For this example, the Community of PyCharm is used as an IDE as well as to run tests. An example test file is provided (see “API/tests/test\_sensitivity\_WUI.py”). A key component of the code is shown below.

Text

Description automatically generated

The above builds the application graph and is required for the code to run. For the code to run some environmental variables need to be defined. In PyCharm you can do so through the “Edit Configurations” option. The variables that need to be defined and their assigned values are:

ALLOWED\_HOSTS = []

DATABASE\_HOST = []

DATABASE\_PASSWORD = []

DATABASE\_PORT = []

DATABASE\_USER = []

Additionally, a SECRET\_KEY variable needs to be defined. This can be any random string, but it must not be left blank. The referenced database is a list of registered users and their API keys. Since this is a local run the database isn’t available and the way the code is being run in the file means there is no call to the database, as such those values can be left blank for local tests.

A local run can be done without a virtual environment (venv), but is far easier when using one, as the required libraries will be automatically downloaded for the venv without the need to manually install them using Python’s pip. PyCharm community automatically establishes a venv when a project is loaded. If using PyCharm be sure to set “E3/e3\_django” as the root directory.