

Apogee - Electron and Node.js

Apogee - Electron and Node.js

The Apogee application can be run in Electron. With it, you can open and save files directly to your local computer. See the page [Apogee in Electron](#) to learn about doing this.

An Apogee workspace can be deployed and run as a web service, using the Apogee Server, running in Node.js. See the page [Writing Web Services with Apogee](#) to learn about doing this.

Apogee in Electron

Installing and Running Apogee

NOTE: The Apogee application is not packaged as an executable. You must run it using Node.js. This process here assumes you have Node.js and NPM installed on your machine and you are familiar with using them. Admittedly, the documentation here is terse.

You can download a zip file containing the contents necessary for the Apogee application for Electron from the Apogee [download page](#).

Unzip the file. This should create a folder with the name ApogeeJS with an appended version number. Navigate inside this folder. You should see a number of files. Three in particular are:

- apogee.html
- main.js
- package.json

Here you must install the app with NPM. This downloads all the needed references for the application.

```
npm install
```

After doing this, you can run the application. This should open the Electron app running Apogee.

```
npm start
```

Adding NPM Modules for Apogee

To make new modules accessible to Apogee in Electron, you must add the modules to the package.json file and re-install the application, to download the added references.

It is assumed here you know how to update package.json and you know how to access these modules from your javascript code.

What about the references folder in the App?

There is a references folder in the app that allows you to add javascript links, CSS files and npm modules.

Adding an npm module here will not add access to that npm module. It still must be put in package.json and reinstalled.

There is one reason you would add an npm module here, if it is a self installing module. when it is added, the module is not installed but it is loaded. Any actions done by the module will be executed

New components (such as maybe a plotting component, or a GeoJSON component) can be added to the system if their modules are included.

Application Notes

Apogee running in Electron is very similar to Apogee running in the web browser. There are a few notable differences:

- Opening and Saving File - If Electron, you can open and save files directly to your local computer.
- References - The web and desktop version use different module systems. As such, workspaces used in the web version may not work in the Electron version and vice versa.
 - Web Modules - In the web version, a module system is used that downloads the modules over the network.
 - NPM Modules - In the Electron version, the npm module system is used. This allows you to use the same modules you use in Node.js, including modules for

database connections and reading from the file system.

- Debugging - Debugging is slightly different in the Electron version. You can open the debugger from the application itself. This is because there is no web browser from which to open the debugger. Otherwise, they should be the same.

Apogee Web Services

One of the uses for Apogee in Electron is if you are creating a web service using Apogee. The workspace in Apogee for Electron uses the same module system that is used in the Apogee Web Server running on Node.js. This allows you to access things like the local file system or database that you can not access from the web browser version of Apogee.

Writing Web Services with Apogee

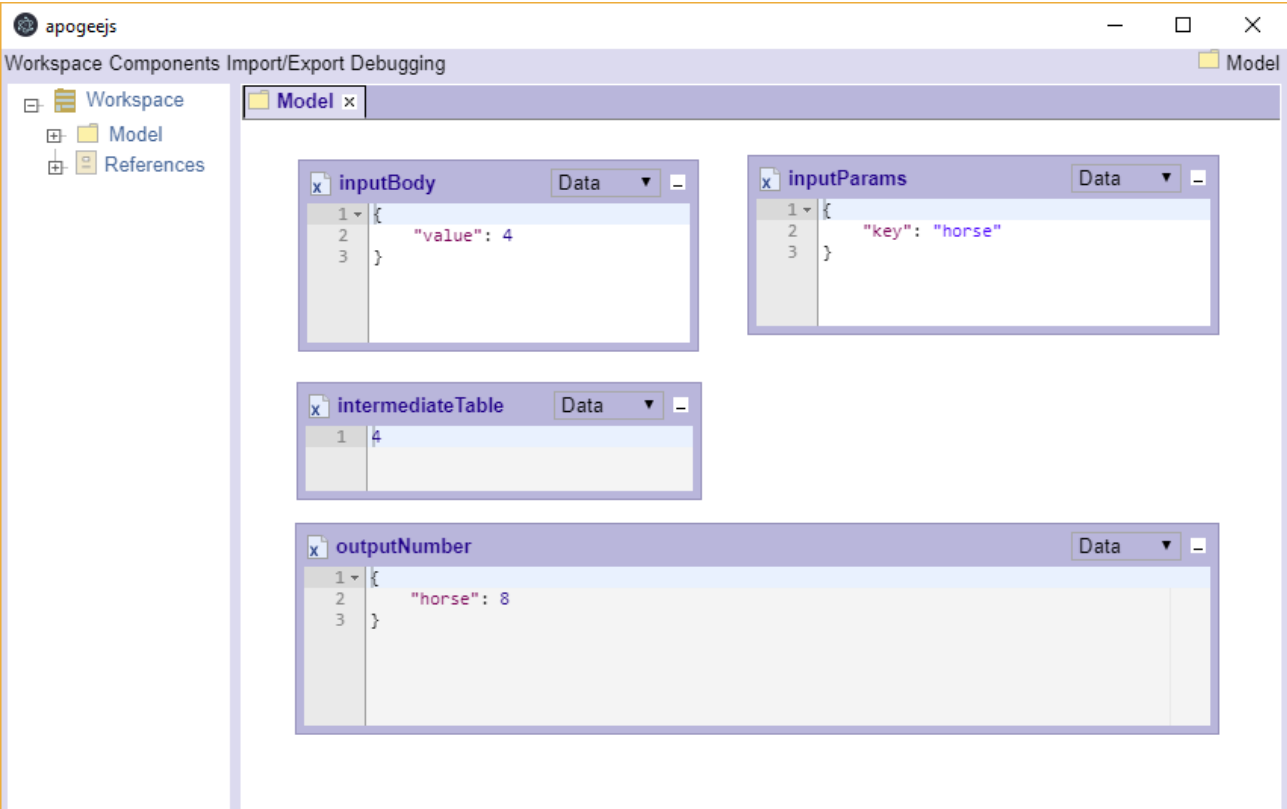
Web Services with Apogee

DISCLAIMER: This is developmental/example code. The current software is not quite ready for use trying to be a competitor to someone like Google. (But if you are so inclined, go ahead.)

To make a web service using an Apogee workspace, you specify input tables and an output table. The request data is passed to the input tables and the workspace is updated. On completion, the result of the output table is returned as the result.

Simple Example Workspace

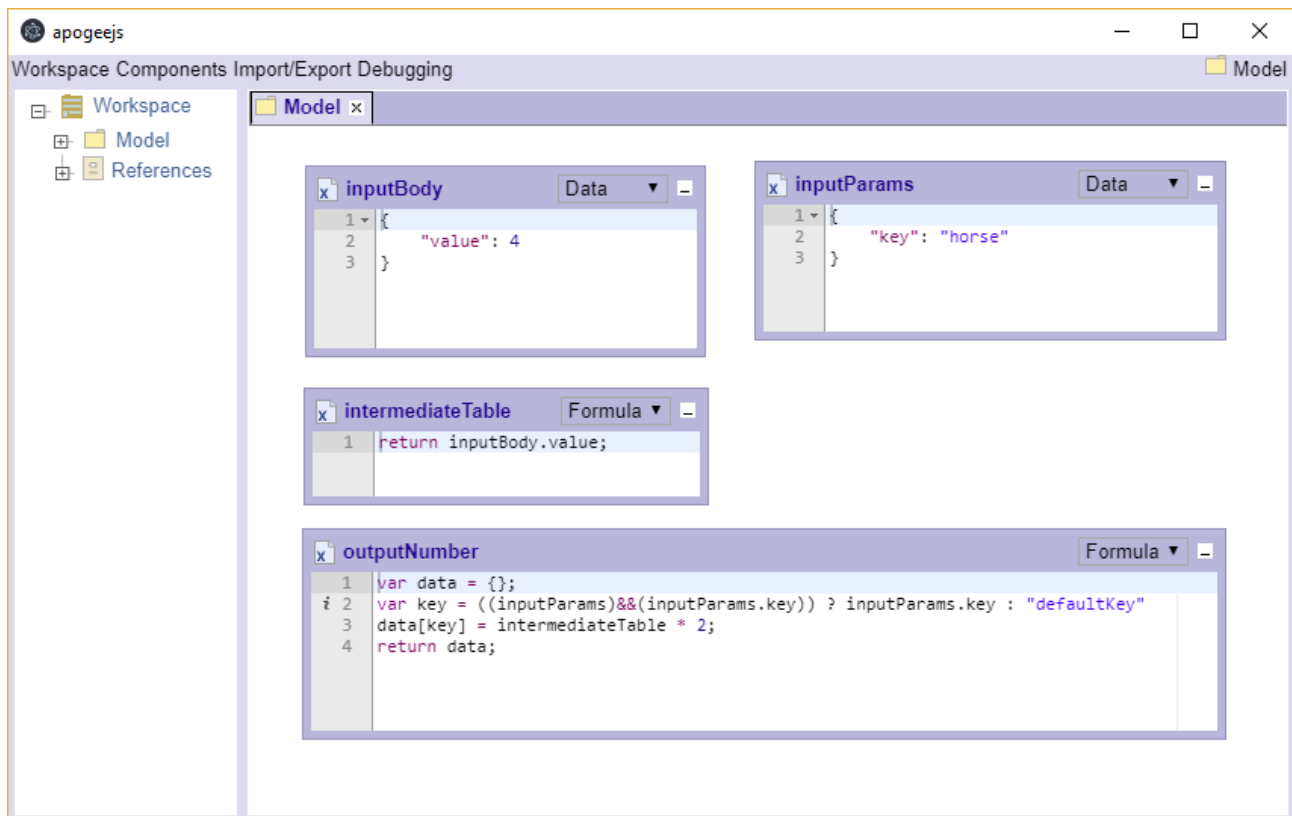
The following is a simple example workspace. We will use the table `inputBody` to accept the body of the request. The table `inputParams` will accept the query params, which will be passed in the form of a JSON key-value map.



The screenshot shows the Apogee workspace interface. The left sidebar contains a tree view with 'Workspace', 'Model', and 'References'. The main area displays four tables under the 'Model' tab:

- inputBody**: A table with 3 rows. Row 2 contains the JSON object `{ "value": 4 }`.
- inputParams**: A table with 3 rows. Row 2 contains the JSON object `{ "key": "horse" }`.
- intermediateTable**: A table with 1 row. Row 1 contains the value `4`.
- outputNumber**: A table with 3 rows. Row 2 contains the JSON object `{ "horse": 8 }`.

We have an intermediate table and the output. The formulas for these are shown below.



Descriptor

For our server, we must provide a descriptor, which specifies the workspace files(s) and the endpoints, including the inputs and outputs. Our example descriptor is below.

```
1  {  
2    "workspaces": {  
3      "simple": {  
4        "source": "test/simple/simpleWorkspace.json",  
5        "endpoints": {  
6          "processNumber": {  
7            "inputs": {  
8              "body": "Model.inputBody",  
9              "queryParams": "Model.inputParams"  
10           },  
11          "outputs": {  
12            "body": "Model.outputNumber"  
13          }  
14        }  
15      }  
16    }
```

```
17   }  
18 }
```

This descriptor says:

- We have one workspace in our service, accessed by the path fragment ***simple/***, loaded from the file *simpleWorkspace.json*.
- This workspace has one endpoint, called ***processNumber***.
 - The request body will be passed to the table ***inputNumber***.
 - The query params will be passed to the table ***inputParams***.
- The response body will be provided by the table ***outputNumber***.

This workspace is included with the downloaded version of Apogee Server. We will install that below so we can try it out.

Installing and Running the Apogee Server

You can download a zip file containing the contents necessary for the Apogee Server from the Apogee [download page](#).


Unzip the file. This should create a folder with the name ApogeeServer with an appended version number. Navigate inside this folder. You should see a number of files. In particular, you should see the file *package.json*.

Here you must install the app with NPM. This downloads all the needed references for the application.

```
npm install
```

After doing this, you can run the application. This should start the server running in Node.js.

```
npm start
```

This server includes an Apogee workspace service and it also includes a plain static files service.

- The folder ***./file*** contains the static files that are served. They can be accessed with the path prefix ***file/***.
- The folder ***./test/simple*** contains the descriptor and the apogee workspaces.
- The port is set to ***8888***.

The above configuration is contained in the javascript file ***server.js***.

Trying it Out

We can load a static test page that let's us send a request with a URL and optional body. The resulting response body is shown.

In the test page below we called the the test service. We pass a body holding the value of 10 and a query string that is used to define the output key. After making the request, we get the result we expect .

The screenshot shows a web browser window with the address bar at `localhost:8888/file/request.html`. The page title is "Request". Below the title, there is a text area with the instruction: "This is the request. An empty body will be done as a GET and a none empty will be done as a POST." Below this, the "URL:" field contains `http://localhost:8888/apogee/simple/processNumber?key=output`. The "Body:" field contains a JSON object: `{ "value": 10 }`. A "Submit" button is located below the body field. Below the submit button, there is a section for the response. It starts with the text "This is the response." followed by a "Body:" field containing the JSON object `{ "output": 20 }`.

This illustrates the calculation of the workspace using inputs that we passed in the web request, returning the result value as the response.

Further Reference

Endpoint definitions

- The options for input table are:
 - "body" - This table will be passed the body of the request. This field is optional.
 - "queryParams" - This table will be passed the query params as a JSON object. This field is optional.

- "trigger" - If there is no query params or body, this option can be used. It will write a predefined value to the specified table, to trigger the workspace action. (See the example descriptor below including the specified value.) This field is optional.
 - <no inputs> - It is also possible to set no inputs and no trigger value. In this case the workspace will not update. A value from the workspace will be read out.
- The options for output table are:
 - "body" - This table will be used to provide the output for the request. This field is optional.
 - <no outputs> - It is also possible to specify no outputs. In this case, the response will return with the workspace is finished calculating, either with success or an error.

Another Example Web Service

There is another sample web service included in the download. To try it, you must update the value of *APOGEE_DESCRIPTOR_LOCATION* in the file *server.js*. This is illustrated below, commenting out the "simple" workspace example.

```
1 //const APOGEE_DESCRIPTOR_LOCATION = "test/simple/descriptor.json";
2 const APOGEE_DESCRIPTOR_LOCATION = "test/other/descriptor.json";
```

This one contains two workspaces and a total of three endpoints. This illustrates some of the other possible endpoint configurations:

- Using the input trigger, when there is no query params or body.
- Using no input, which gives a static return value.
- Using no output

```
1 {
2   "workspaces": {
3     "triggerTestWorkspace": {
4       "source": "test/other/triggerTestWorkspace.json",
5       "endpoints": {
6         "triggerTest": {
7           "inputs": {
8             "trigger": "Model.inputTrigger"
```

```

 9      },
10      "triggerValue": "anything you want. If there is nothing set, 'tr
11      "outputs": {
12          "body": "Model.currentUTC"
13      }
14  }
15  },
16  },
17  "otherTestWorkspace": {
18      "source": "test/other/otherTestWorkspace.json",
19      "endpoints": {
20          "staticReturn": {
21              "inputs": {
22              },
23              "outputs": {
24                  "body": "Model.staticReturn"
25              }
26          },
27          "noReturn": {
28              "inputs": {
29                  "body": "Model.inputWithNoReturn"
30              }
31          }
32      }
33  }
34  }
35  }

```

The workspaces and the descriptor can be found at the above defined location. These workspaces can also be opened in the Electron version of Apogee.

Note on the Trigger

In the example workspace that uses the trigger, we define the value of the input table to be *apogee.util.INVALID_VALUE*. This means that any table that depends on this will also be flagged as an invalid value and not calculated.

In addition, we put a gratuitous reference to the table inputTrigger in a table involved in our calculation. What this means is that table will have a dependency on the table inputTrigger. When we make the request, the INVALID_VALUE flag is cleared.

Please view the example workspace in the Electron version of Apogee to see this example.

This is just one way to use the trigger. Others can also be done.

