# Final Design and Implementation

The Genesis Configuration Management System was designed as a Service Oriented Architecture as explained in the methodology section above. In this architecture model, the Genesis system uses the services to perform the various actions upon the data submitted by the user. Currently, and in this implementation of the system they are only RESTful CRUD services that take in information and attempt to place them into the data stores. In order for the user to supply the data to the system there is a user interface designed to operate with the system services by supplying forms that take the user inputs validate them and send them off to the services, more on that later. In order for the front end UI to know what services are currently running and where, the system has implemented a service broker which allows the services to register themselves with the network address of the machine they are running on. The front end can query this service broker to find out if it can send it’s requests off to the desired service and where those requests need to be sent in order to reach those services. Most of this was covered in the methodology section above, we will go further into the details of each of these pieces of the system in the following sections. Here is an overarching diagram of how the Genesis Configuration Management System was built.

Diagram, schematic

Description automatically generated

Figure 1: GENESIS Application Interactions

## Data Storage

The system uses three types of data storage techniques to accomplish its goals. The first is a relational database to store both the scenario information and the data provided by the user to be used within each of these scenarios. The second is a flat file storage which is used to store the template data that is used to define each of the subsections of the simulation environment within the Genesis configuration management system. The third and final method of storage is a non-relational key value store which is used to track the running services and the network address of the machine they are running on.

To implement the main data store, a relational database called MySQL[] was used. This database was chosen because it is one of the leading relational databases on the market, it is free to use, it has exceptional data resiliency, can be installed on any operating system, and uses the widely know Structured Query Language (SQL) (). This is also the database that the Space Simulation uses to store its data while a simulation is running. Genesis makes use of relational databases for storing the scenario information it is suppose to be tracking as well as setting up the relational databases for the space simulation to run off of. A schema, or database, was created specifically for Genesis, called the *genesis\_scenarios*. In which three tables were created, one table for the scenario information itself, another table for the subsystem template information, and a third junction table that links the subsystem templates to the scenarios in a many-to-many relationship. The first two tables are standard data tables, which are defined below, but the third is interesting since we needed a way to link any number of defined subsystem templates to any number of scenarios that needed to use them. This method of doing things promotes not only the reuse of the template definitions within the table but reuse of the template definitions stored as flat files, more on that later. The two data tables used the data models below to build their tables.

|  |  |  |
| --- | --- | --- |
| Column | Data Type | Description |
| PKey | int | Unique identifier for this module template, will be automatically assigned to each record by the database |
| Name | varchar(25) | A human readable identifier for this template |
| Version | varchar(10) | The version of software this template belongs to |
| FilePath | Text | The directory file path to the folder containing the template’s files. |
| Description | Text | A more in depth description of the template and its purpose. |
| Deprecated | bool | A flag indicating that this template is old and has been replaced with a newer version. |

Table 1: Template Data Model

|  |  |  |
| --- | --- | --- |
| PKey | int | Unique identifier for this scenario, will be automatically assigned to each record by the database |
| Name | Varchar(25) | A human readable identifier for this scenario |
| Description | Text | An explanation of the purpose and or event that this scenario was used to support |
| Date Last Updated | Date Time | The last time that this scenario was updated by any user |
| Templates Used | Foreign Keys | The id’s of the templates used to create this scenario |

Table 2: Scenario Data Model

|  |  |  |
| --- | --- | --- |
| Column | Data Type | Description |
| PKey | int | Unique identifier for this scenario, will be automatically assigned to each record by the database. |
| Name | Varchar(25) | A human readable identifier for this scenario |
| Constellation | Varchar(25) | An explanation of the purpose and or event that this scenario was used to support |
| Rx | double | The starting x position of the vehicle |
| Ry | double | The starting y position of the vehicle |
| Rz | double | The starting z position of the vehicle |
| Vx | double | The starting x velocity of the vehicle |
| Vy | double | The starting y velocity of the vehicle |
| Vz | double | The starting z velocity of the vehicle |

Table 3: Spacecraft Module Spacecraft Data Model

One feature of MySQL is when defining a table, you must define a key for each item in the table but it also allows you to auto assign an id when a new record is added to the table which allows that database, the one source of truth on this data, to keep track of the next id number to assign to an incoming record but also guarantee that number is unique. This particular feature takes care of this tracking so that the services do not have to do it themselves.

In order to store more complicated data, such as the template data models and SQL queries, a flat file was chosen because of its flexibility for the types of data that can be placed within each “record”. The Genesis system is using the built in file directory system within the Windows 10 operating system as its flat file database. This offers a lot of flexibility, not only to the types of data that can be stored here but also allowing the data to be organized in any way. This flexibility does come at a cost though, its is relatively unsecure because anyone with access to the directory can go in a make changes to the data but it can be slow to store and retrieve the data. This was worth it due to the flexibility since the data that defines a subsystem template is complex and varies between subsystems. Genesis operates under the following assumptions, Each template directory will be placed in a single location, each template will have a directory matching its name, there will be a sub directory for each version of this template denoted as *v\_#.* Using these assumptions to add data to this flat file database allows for simple logic on the programming side in order to access this information.

Flat File Diagram here

In order to keep track of the services and their addresses a non-relational database called Redis was selected for its key value store and the ability to subscribe to changes to particular keys. The space sim also uses Redis in its environment for the publish and subscribe feature that Genesis could make use of in the future. We will talk more about the service broker that was built on top of this data store later. In order to be able to easily find the service data stored within the data store, the keys for the services took on the following shape, *genesis.broker.<service\_name>*, and store the Ip address and port they were running at as a string key.

|  |  |  |
| --- | --- | --- |
| Key | Data Type | Description |
| genesis.broker.[service name] | string | The Ip address, including the port of the service specified in the key name. Set to expire after 30 seconds. |

Table 4: Redis service key model

## Services

The services within the configuration management system are used to accomplish the CRUD actions with the relational data store and can be sent data through an application program interface which defines how outside applications can properly communicate with the service. The two services implemented in this project were build as NodeJS servers. A framework call Express was used within the NodeJS runtime environment to transform it into a webserver. This framework allows us to build routes into the server in which the users can access through URL calls, which allows us to follow the REST principals for this API. Once a request was made of the API, it would than take the information passed to it, parse any JSON data provided in the body, validate it, perform the requested action and respond to the user. Both of these service used a library called Winston to perform logging throughout this process. Any request, response, and error was logged so if something were to break during the operation of these services, than it could be easily tracked down later by a developer.

The first CRUD service built is called the Genesis Scenario Service and it handles all of the actions related to scenario and templates. It allows any program to request it’s API documentation by accessing the base route, *“/”,* on the machine it is running on. To follow the REST principals it is designed to be stateless, thus any information that it requires needs to come through the request itself or be stored in the database. For most of the routes all of the data required could be stored in the body of any given HTTP request. It will query the database itself when creating or updating any scenario or template because it will return the object stored in the database as part of the response body. The Genesis Scenario Service will takes care of coupling and decoupling of templates and scenarios. One unique feature about this service is that it will create the scenario database within MySQL when a scenario is created and will also generate the template tables within that database when templates are linked to the scenario. This is done as long as the template data store contains a Create SQL file for the template in question. All in all this is a very straight forward RESTful CRUD service without a bunch of added extras.

EXAMPLE API DOCs

The second CRUD service, called the Spacecraft Module Service, is directly linked to the Spacecraft template used to handle basic space object data within a simulation. This is a NodeJS server running with Express and Winston, and uses the same route paradigm as the Genesis Scenario Service, but is considerably more interesting. This service performs CRUD actions in a RESTful manner on the Spacecraft Template data requests sent to it. Since the template in question can have different versions the service needs to be dynamic enough to serve any defined version of the template at any time. This will allow for older versions of the template to be used should the team want to pick up and run or edit an old scenario from a past event. To do this the service has some custom middleware that does two things, it fetches the spacecraft data model from the template data storage location and passes it along the route. It also will choose the correct database in which to perform the requested action against based upon the name of the scenario provided with the request. As mentioned before, the request needs to provide both the directory path to the template being used and the name of the scenario in which the action is taking place. These are both provided as part of the header of the request since, GET requests typically are forbidden from having request bodies under HTTP protocol standards(). Other than these differences it is built very much like the Genesis Scenario Service.

## Service Broker

In the Genesis system we have services and servers can be ran on any machine in the network, but they still need to send requests to each other in order to work. In order to do this, they need to know where the services are running within the network. This is where a service broker comes in, it will know where the services are running and can be queried for any service that is running. This was done using the Redis database key value store. When a service starts up, in the Genesis System, it will report the location in which it is running on to the database by creating a key, following the pattern discussed above. When it creates this key, it will set it to expire after 30 seconds. This means that throughout the life of the service it will need to recreate this key in under 30 seconds of the last time in order to keep receiving requests, we call this a keep alive. This will cause the service to unregister should it run into any type of exception that brings the service down, thus causing the rest of the system to continue running without it. If this service is needed, the user will be informed to take action in order to bring the service back online, and ideally create a bug report for the failure. When another application wants the address of a service that is running, all that it needs to do it pull the list of keys matching the service name desired and pick one from the list, since if it is in the database it can be assumed to be running, and pull the address stored. It will than use this address as the base URL in which to make its requests of the service.

## Front End

The front end of the Genesis system was implemented as two parts, the first being a NodeJS server that would serve the second part, a React Single Page Application, or SPA. The server would run on some machine within the network and would allow the user to access the SPA when they connected through a web browser URL. It would also serve the template file data store through the same means. This would allow the React application access to each of the templates specific data models which it uses to dynamically generate the forms for the user to fill out. The server also provides a connection to the service broker, as discussed above. Now the interesting part is the React application itself, because it is where all of the logic for creating and managing the scenarios comes together. The react application comes up and allows the user to either create a new scenario or continue working on an existing one. If the user selects to create a new scenario, they are shown a form that allows them to enter the scenario details and to link subsystem templates to this new scenario. This form doubles and the template CRUD form where the user may Create new templates and edit or remove any existing template. Once the user is done creating the template they may submit the form which reaches out an uses the Genesis Scenario Service API to add the scenario and create the links between the templates and the scenario. Upon closer inspection, you would realize that there is no way to edit a scenario through the user interface, that is by design at this time. Editing a scenario by linking new templates or removing current templates would require major changes to the scenario databases. This was determined to be a possible feature in the future but would require some complex logic and more testing to incorporate properly.

Should the user decide to select an existing scenario they are taken to the scenario editing page where they can see the data associated with the scenario. This data is split into tabs, each representing a specific template. The user may both view the data in each tab as well as perform the CRUD actions on that data. All of the controls associated with the template are dynamically generated based upon the data model stored in the template’s flat file database location. The data model contains the fields in which to display to the user as well as the fields that the user needs to fill out in order to add another record to the scenario. In addition to the field, the validation type and limitations for the field are stored within this object. When the user interface control is rendered, it reaches out and dynamically imports the model in which to use in the code, which turns out is a super handy feature when trying to create dynamically generated forms and tables. To populate the table and submit the records, the UI reaches out and talks to the template service for its CRUD actions. In this case there is only one template, the spacecraft template and thus only one service to support it.