MaGiC: a DSL Framework for Implementing Language Agnostic Microservice-based Web Applications

User Manual

Contents

1 DSL Documentation

1.1 Microservice DSL

The Microservice DSL represents the domain-specific language used to specify a REST MSA, particularly users can express a microservice suite where each microservice exposes CRUD operations which are used to interact with a particular set of data.

1.1.1 Syntax

The DSL consists of several properties which are used to specify metadata about the microservice such as its name, version, description, and maintainer but also technical specific configuration such as its running port, generated language target, and data reference. In addition to this, a microservice can be configured to expose several operation methods each operation supporting CRUD operation types interacting with a given *DataType* which represents a JSON array data structure. Furthermore, the operation's API endpoint is automatically generated, however the operation's query parameters, expected / delivered payload, and respectively its success / error message, need to be specified.
Listing 1.1 illustrates a formal representation of the DSL features' abstract syntax using Backus–Naur form (BNF) notation.

```
microserviceName ::= <string>
version ::= <string>
description ::= <string>
maintainerEmail ::= <string>

port ::= <integer>
language ::= Python | NodeJs
dataReference ::= <DataType>

Operation features syntax:

method ::= CreateEntity | DeleteEntity | GetEntities | GetEntitiesBy |
    GetEntity | GetEntityBy | UpdateEntity
type ::= CREATE | READ | UPDATE | DELETE
entityType ::= <EntityType>
queryParams ::= <key> { " " <key> }
expectedPayload ::= Empty | Entity | EntityID | Entities
deliveredPayload ::= Empty | Entity | EntityID | Entities
successMessage ::= [ <string> ]
```

```
errorMessage ::= [ <string> ]

The DataType is represented by the following syntax:

<DataType> ::= <value>, where <value> represents a valid JSON array data
    structure

Respectively, the EntityType is represented by the following syntax:

<EntityType> ::= { <key> : <value> } { "," <key> : <value> }, where <key> is a
     string constant that defines a data property, and <value> ::= string |
    number | integer | boolean
```

Listing 1.1: Representation of Microservice DSL features using BNF notation.

The abstract syntax features presented above (see Listing 1.1) are then incorporated in the DSL concrete syntax. The following Listing 1.2, illustrates the specification of a microservice expressed through the Microsevice DSL concrete syntax.

```
Microservice name <microserviceName>
version <version>
description <description>
maintainer email <maintainerEmail>

port <port>
language <language>
data reference:
 <dataReference>

exposes
  operation <method> with type <type> on <dataType> data
    with entity type <entityType>
    at endpoint location with query params <queryParams>
    expecting payload <expectedPayload>
    delivering payload <deliveredPayload>
    with success message <successMessage>
    and reporting error message <errorMessage>
```

Listing 1.2: Microservice DSL syntax.

1.1.2 Language Documentation

The following enumeration presents the Microservice DSL documentation of the language features and concepts:

port , represents the destination port of the process in which the microservice is running in.

language , is the generated programming language which is either Python or Node.JS.

dataType , the JSON array representation of the data. Due to the scope of the project, it was decided that the data with which a given microservice interacts using CRUD operations, will not be kept in a database system but instead it will be stored locally in a JSON file. It is worth mentioning that the underlying data structure of JSON in which the information is stored is similar to the way NoSQL databases store information.

entityType , the entity type reference that the *dataType* contains.

method and **type**, the user of the DSL can choose between several operations methods related to CRUD (CREATE, READ, UPDATE, DELETE) operation types, such as:

CreateEntity , used for creating a new entity in the *dataType* and it is used concomitantly with *CREATE*.

GetEntities , used for retrieving all the entities from the *dataType* and it is used concomitantly with *READ*.

GetEntitiesBy , used for fetching a collection of entities from the *dataType* based on *entityType* properties, used with *READ* type.

GetEntity , used for retrieving an entity from the *dataType* by its id, represents an operation of type *READ*.

GetEntityBy , used for retrieving an entity from the *dataType* by its properties, *READ* type.

UpdateEntity used for updating an entity, represents an *UPDATE* type.

DeleteEntity , used for deleting a new entity from the *dataType* and it is used concomitantly with *DELETE*.

queryParams , represent the URL parameters with which the request is made to invoke an action of a microservice at a given endpoint. It is important to note that the parameters are properties of the *entityType*, as the service is designed so that it interacts with a given *entityType*.

expectedPayload and **deliveredPayload**, each microservice endpoint can be set up to expect a payload and deliver a payload as a response to the initial request. This design decision was made to align the microservice exposed operations with the HTTP message standard, and the *expectedPayload* and *deliveredPayload* represent the body of the message. The types of *expectedPayload* and *deliveredPayload* are:

entity , which represents a single *entityType*.

entities , represents a collection of *entityTypes*.

entityId , represents the id of a given *entityType*.

empty , the request body is empty.

successMessage and **errorMessage**, represent the text messages sent back as response depending on the outcome of the service request. If the request if successful the *successMessage* message is sent back as response along with the requested data, otherwise if the request fails the *errorMessage* is sent.

1.2 Gateway DSL

The Gateway DSL is used to specify and generate the communication layer between the server-side / suite of microservices and the client-side / application's user interface.

1.2.1 Syntax

The DSL consists of several properties which denote metadata about the gateway such as its name, version, description, and maintainer, but also configuration specific information, namely its port and generation language target. Furthermore, the DSL exposes different operations each supporting CRUD operations communicating with a referenced microservice API endpoint.
Listing 1.3 illustrates the Gateway DSL's abstract syntax in Backus-Naur form.

```
BFFName ::= <string>
version ::= <string>
description ::= <string>
maintainerEmail ::= <string>

port ::= <integer>
language ::= Python | NodeJs

Operation features syntax:

method ::= CreateEntity | DeleteEntity | GetEntities | GetEntitiesBy |
    GetEntity | GetEntityBy | UpdateEntity
type ::= CREATE | READ | UPDATE | DELETE
route := <string>
entityType ::= <EntityType>
queryParams ::= <key> { " " <key> }
deliveredPayload ::= Empty | Entity | EntityID | Entities
expectedProperties ::= <key> { " " <key> }

microservice ::= <MicroserviceName>
microserviceEndpointLocation ::= <string>
```

Listing 1.3: Representation of Gateway DSL features using BNF notation.

The abstract syntax features of the Gateway DSL (see Listing 1.3) are then incorporated in the DSL concrete syntax. Listing 1.4, illustrates the syntax of specifying a gateway using the Gateway DSL.

```
BFF name <BFFName>
version <version>
description <description>
maintainer email <maintainerEmail>

port <port>
language <language>

exposes
  operation <method> with type <type> at endpoint location <route> on entity
   type <entityType>
    delivering query params <queryParams> or payload type
    <deliveredPayload>
    expecting properties <expectedProperties>
  which communicates with
    microservice <microservice>
    at endpoint location <microserviceEndpointLocation>
```

Listing 1.4: Gateway DSL syntax.

1.2.2 Language Documentation

In terms of language documentation the Gateway DSL is strongly related to the Microservice DSL language documentation illustrated in previous section. Nevertheless, the DSL for specifying the gateway has some specific features, such as:

route , represents the URL route to which the gateway API endpoint listens for potential incoming requests.

expectedProperties , specifies the *EntityType* properties that should be sent to the client-side.

microservice , represents the reference of the microservice with which the gateway is supposed to communicate with.

microserviceEndpointLocation , represents the URL route of the microservice API endpoint to which the gateway should forward the request from the client.

1.3 Client DSL

The Client DSL represents the language for specifying and generating the client-side / UI of the microservice based web application.

1.3.1 Syntax

The syntax of the DSL is based on multiple concepts, particularly the structure of the client-side application which is split in pages and routes. Additionally it consists of microservice-domain rest specific aspects extracted from the Gateway DSL and respectively Microservice DSL, particularly the client pages support CRUD operations, but also from client-side application specific features. The Client DSL also consists of a configuration structure used to specify the port in which the client application is running and which one of the client types (desktop / mobile) it exposes. Listing 1.5 presents the abstract syntax of the Client DSL.

```
Client configuration syntax
    appName ::=  <string>
    port ::= <string>
    mobileClient ::= <ClientName>
    desktopClient ::= <ClientName>

Client syntax
    name ::= <string>
    maintainerEmail ::= <string>
    description ::= <string>
    version ::= <string>
    clientType ::= Mobile | Desktop

    globalState ::= <string> { " " <string> }

Page syntax
    pageName ::= <string>
    bff ::= <BFFName>
    bffRoute ::= <string>

    pageRoute ::= <string>
```

```
    queryParameters ::= <key> { " " <key> }
    globalState ::= { <globalState> : <key> } { "," <globalState> :
    <key>}, where <globalState> is one of the specified global state
    properties, and <key> refers to a string constant that is a property  of
    an <EntityType>.
    entityType ::= <EntityType>
    showInNavigation ::= boolean
    hasInternalState ::= boolean

Page operation syntax
    operationName ::= <string>
    bff ::= <BFFName>
    bffRoute ::= <string>
    type ::= CREATE | READ | UPDATE | DELETE
    entity ::= <EntityType>

    deliveredQueryParams ::= <key> { " " <key> }
    deliveredGlobalStateProperties ::= <globalState> { " " <globalState> }
    deliveredPayload ::= Empty | Entity | EntityID | Entities
    redirectsTo ::= <PageName>
    globalState ::= { <globalState> : <key> } { "," <globalState> : <key>},
    where <globalState> is one of the specified global state properties, and <
    key> refers to a string constant that is a property  of an <EntityType>.

    components =  <Component> { " " <Component> }

<EntityType> ::= { <key> : <value> } { "," <key> : <value> }, where <key> is a
    string constant that defines a data property, and <value> ::= string |
    number | integer | boolean
```

Listing 1.5: Representation of the Client DSL abstract syntax using Backus-Naur form notation.

Furthermore, the Client DSL has an additional syntax for specifying the UI components (denoted by *¡Component¿* in Listing 1.5) of the application which is inspired from HTML. The syntax of each of the UI component is illustrated in Listing 1.6.

```
<Title>
    text ::= <string>
</Title>

<Text>
    text ::= <string>
 </Text>

<Link>
    text ::= <string>
    links to ::= <PageName>
    queryParms ::= <key> { " " <key> }
</Link>

<Input>
    name ::= <string>
    label ::= <string>
    type ::= email | number | password | tel | text
    required ::= boolean
 </Input>
```

```
<Image>
    imgSrc ::= <string>
    altText ::= <string>
</Image>

<Form>
    interactsWithOperation ::= <OperationName>
    actionText ::= <string>
    inputs ::= <Input> { " " <Input> }
</Form>

<CrudAction>
    text ::= <string>
    variant ::= primary | secondary | success | danger | warning | info | link
    interactsWithOperation ::= <OperationName>
 </CrudAction>

<Card>
    image ::= <string>
    title ::= <string>
    action ::= <CrudAction>
 </Card>
```

Listing 1.6: Representation of the Client DSL UI component syntax.

Listing 1.7 illustrates the abstract syntax used to specify the clide-side of the MSA specified and generated using the MaGiC framework.

```
Client configuration abstract syntax

App name: <appName>
Port: <port>
Mobile client: <mobileClient>
Desktop client: <desktopClient>

Client abstract syntax

Name: <name>
Maintainer email: <maintainerEmail>
Description: <description>
Version: <version>
Client type: <clientType>

Global state {
  <globalState>
}

Page <pageName> {
  Communicates with <bff> BFF and fetches state from <bffRoute>
  Route <pageRoute> with query params <queryParams> and global state <
    globalStateRef> of entity type <entityType>

  Show in navigation: <showInNavigation>
  Has internal state: : <hasInternalState>

  Operations {
    <operationName> {
      Communicates with <bff> BFF
```

```
      at endpoint location <bffRoute> with operation type <type>
      interacts with entity <entityType>

      delivers {
        query params <deliveredQueryParams>
        global state properties <deliveredGlobalStateProperties>
        payload type <deliveredPayload>
      }

      on success {
        redirects to <redirectsTo>
        updates global state <globalState>
      }
    }
  }

  Components {
    <Component>
  }
}
```

Listing 1.7: Representation of the Client DSL concrete syntax.

1.3.2 Language Documentation

The Client DSL consists of several concepts from the client application development domain but also form the REST microservice-domain inspired from the Microservice DSL, and respectively the Gateway DSL, which are documented in below list.

port , is the destination port of the process in which the client side is running in.

clientType , is a configuration parameter of the DSL referencing a client and specifying the supported device clients for which the UI was specified. Currently, the Client DSL supports two types of clients, mobile and desktop, which can be accessed via a web browser. It is worth mentioning that any major device such as laptop, tablet, and smartphone, is supported by this configuration.

globalState , represents the information (state) available throughout the whole client application which can be used and modified in the context of CRUD operations.

bff , serves as a reference to the MPS structure representing the configuration of a gateway specified by using the Gateway DSL. The software design pattern employed for implementing the MSA is BFF which is defined by a gateway for each single client type, that is one gateway for mobile and one for desktop.

bffRoute , is the URL route of the BFF API endpoint with which the client interacts.

pageRoute , is the URL of the client application's page.

queryParams and **globalStateRef**, serves as the URL parameters with which the client page request is made which then are further used to invoke the referenced BFF API endpoint.

entityType , represents the entity type reference of the entity with which the client operation interacts.

showInNavigation , boolean denoting whether or not the page should be displayed in the navigation bar of the application.

hasInternalState , boolean indicating whether or not the client page has an internal state. If the page has an internal state the page information is fetched from the microservice suite before the page is rendered. In contrast if the page does not have an internal state, no data is fetched from the services.

type , used for configuring the operation type related to CRUD (CREATE, READ, UPDATE, DELETE).

deliveredQueryParams and **deliveredGlobalStateProperties**, serves as the URL parameters with which the client operation request is made to invoke the referenced BFF API endpoint.

redirectsTo , the client page reference to which a redirect should happen if the page CRUD operation is successful.

2 Build a "Hello World!" App

The following chapter presents a guide for implementing and deploying a "Hello World!" microservice-based web application using the MaGiC framework. It important to note that the application which will be implemented has a minimal set of features, however more complex applications can be built by following the provided documentation of the languages. Additionally, the MaGiC framework package comes with an already built complex apllication named **ECommerceSandbox** which can be built and deployed using the related steps from the below guide.

2.1 Software Requirements

In order to start developing there are several software requirements that need to be installed.

MPS , version 2021.2 .

Node.js , latest version.

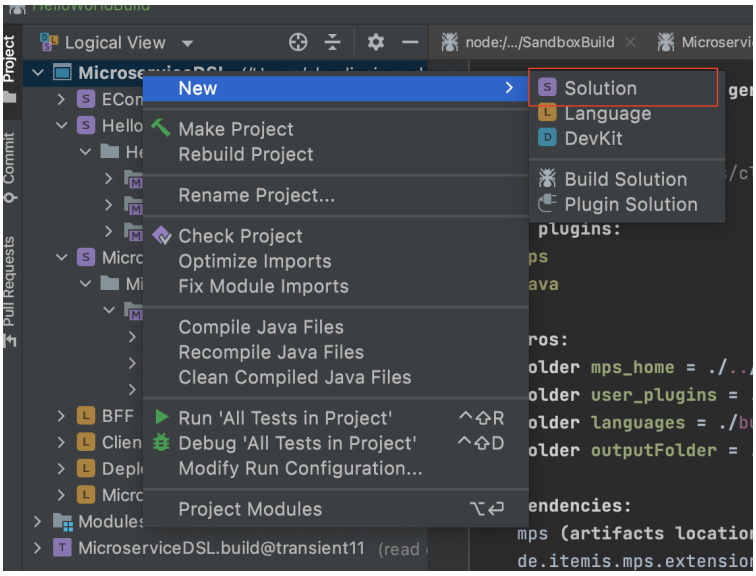
Docker , latest version.

GIT , latest version.

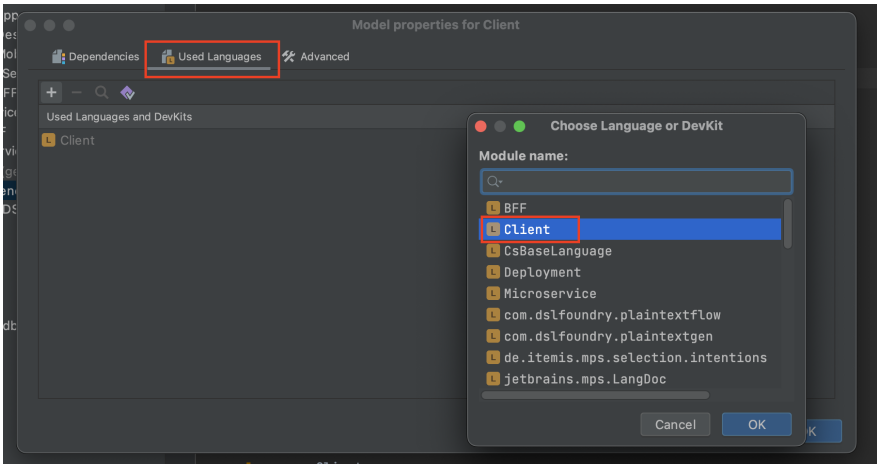
MaGiC framework , the framework can be cloned from the following GitHub repository `https://github.com/claudiuciumedean/language-agnostic-microservice-dsl`.

2.2 Guide

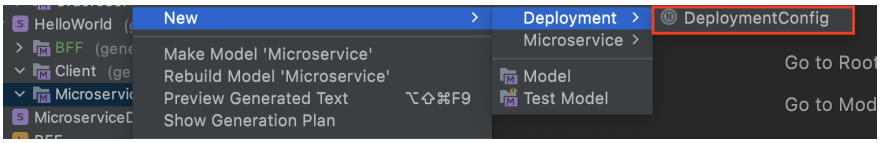
1. Open the cloned project using MPS 2021.2.
2. Create a new MPS solution called "**Hello World**".



3. Right click on the newly added solution and add a new model named **Client**. After the model has been created, right click on it and click on the **Model Properties** button from the dropdown. Navigate to the **Used Languages** tab and add the **Client** language.
   Repeat the same procedure by adding a new model named **Microservice**. Add the **Microservice** and **Deployment** languages in the **Used Languages**.
   Now do the same for the **BFF** model, and add the **BFF** language in the **Used Languages** tab.



4. Right click on the **Microservice** model, hover the **New** button in the dropdown then hover on the **Deployment** button and click on the **DeploymentConfig** button. A new **DeploymentConfig** module should be added under the **Microservice** model. Repeat the same step but this time hover on the **Microservice** button and add **DataType** and **Microservice** modules. After having done this step you should be able to see three different modules added under the **Microservice** model.



5. Navigate to **<no name>[DataType]**. Add the following data as in the below image.

```
Data (valid JSON format) data = [
                                  {
                                    "text": "Hello World!"
                                  }
                                ]


Entity data {
  text : string
}
```

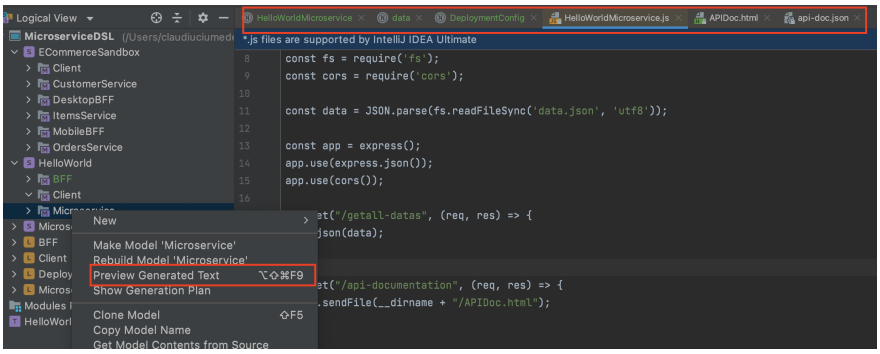
6. Navigate to **<no name>[Microservice]**. Add the following data as in the below image.

```
Microservice name HelloWorldMicroservice
version 1.0
description Hello World!
maintainer email helloworld@gmail.com

port 5000
language node js
data reference:
 data

exposes
  operation get entities with type READ on data data
    with entity type data
    at endpoint location with query params << ... >>
    expecting payload <no payloadType>
    delivering payload entities
    with success message Entity retrieved
    and reporting error message Something went wrong.
```

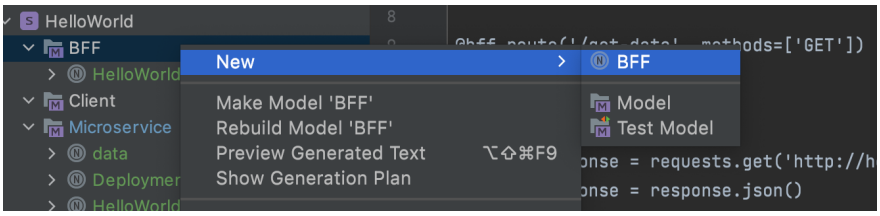
7. Navigate to **DeploymentConfig**. Add the following data as in the below image.

```
Deployment configuration

maintainer helloworld@gmail.com
microservice HelloWorldMicroservice
```

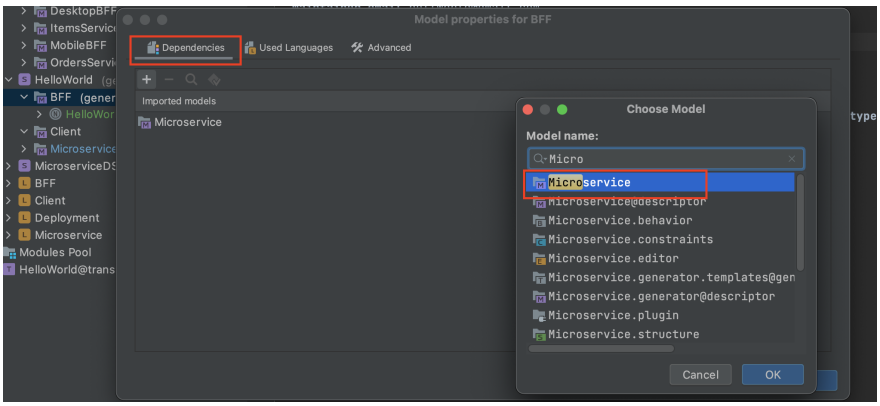
8. Click on the **Microservice** model and click the **Preview Generated Text** button from the dropdown. If you inserted all the data correctly you should not get any errors and some new files should be added to the open files tab. See image below for example.



9. Right click on the **BFF** model, hover the **New** button and click on the **BFF** button.



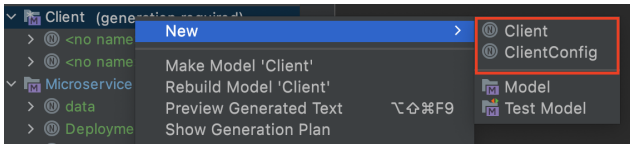
10. Right click on the **BFF** model and click on the **Model Properties** button from the dropdown. Click on the **Dependencies** tab and add the **Microservice** as a dependency.



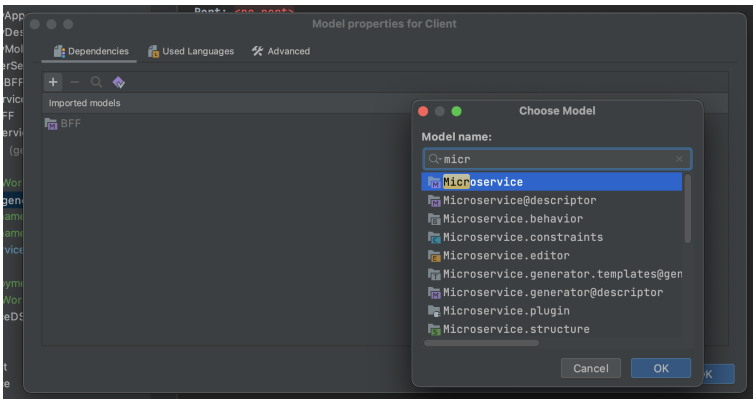
11. Navigate to **<no name>[BFF]**, add the following data as in the below image. After you have inserted the data repeat step **8**, but this time for the **BFF** model, to make sure you did insert all the correct data properly and the files are generated.

```
BFF name HelloWorldBFF
version 1.0
description Hello World!
maintainer email hellworld@gmail.com

port 5001
language python

exposes
  operation get entities with type READ at endpoint location /get-data on entity type data
    delivering query params << ... >> or payload type entities
    expecting properties << ... >>
  which communicates with
    microservice HelloWorldMicroservice
    at endpoint location /getall-datas
```

12. Right click on the **Client** model, hover the **New** button and click on the **Client** button. Repeat the same process but now click on the **ClientConfig** button. You should have now created the **Client** and **ClientConfig** modules.



13. Right click on the **Client** model and click on the **Model Properties** button from the dropdown. Click on the **Dependencies** tab and add the **BFF** and **Microservice** as dependencies.



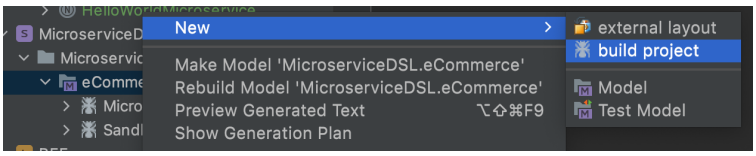
14. Navigate to **<no name>[Client]**, add the following data as in the below image. Note that if you want to add a mobile client as well, repeat the previous step by adding a new **Client** module and add the same data as for the desktop one but for the **Client type** property select **mobile**.

```
Client

Name: HelloWorld
Maintainer email: helloworld@gmail.com
Description: Hello World!
Version: 1.0
Client type: desktop

Global state {
  << ... >>
}

Page Home {
  Communicates with HelloWorldBFF BFF and fetches state from /get-data
  Route /home with query params  << ... >> and global state << ... >> of entity type data

  Show in navigation: true
  Has internal state: true

  Operations {
    << ... >>
  }

  Components {
    <Title>
      text: text
    </Title>
  }
}
```

15. Navigate to **<no name>[ClientConfig]**, add the following data as in the below image. After having done this repeat step **8** but this time for the **Client** module to make sure that you introduced all the data correctly.

```
Client configuration

App name: HelloWorld
Port: 3000

Mobile client: <no mobileClientRef>
Desktop client: HelloWorld
```

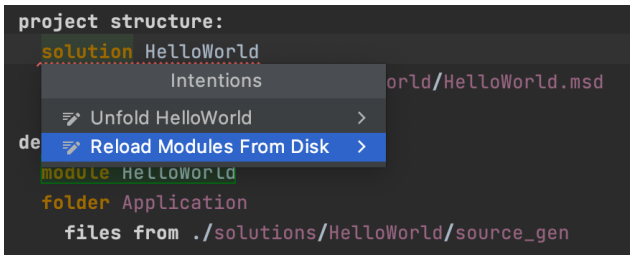
16. Create a new **build project** under **MicroserviceDSL.build >MicroserviceDSL >eCommerce**.



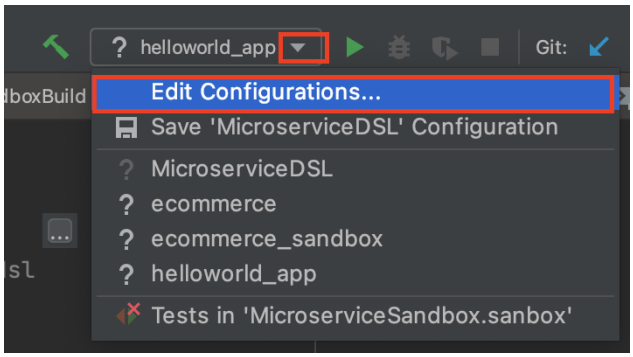
17. Navigate to **<no name>[build project]**, and add the following data. Make sure that the paths highlighted in the image are relative to your system.

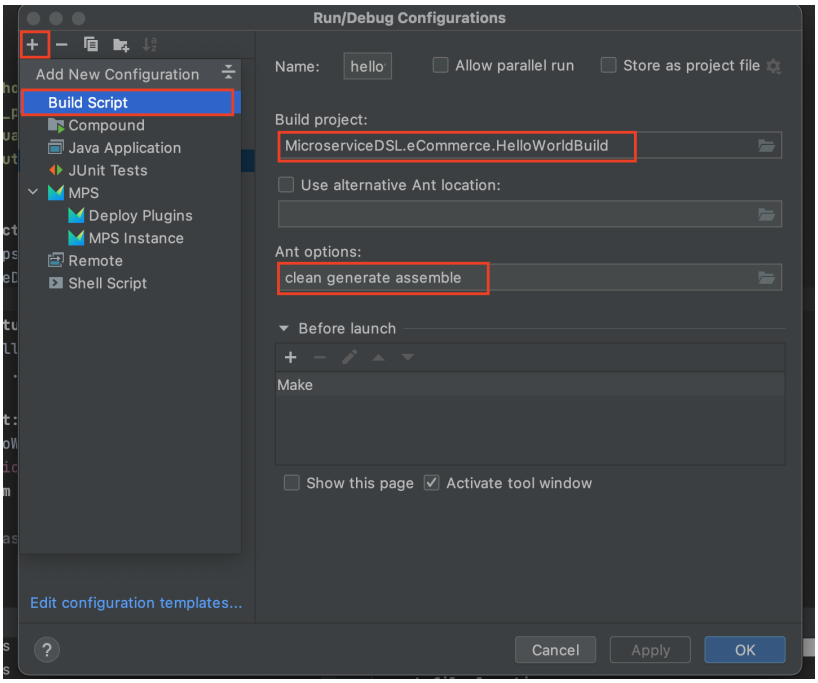
```
build HelloWorldBuild generates sandbox.xml

base directory: ../..
                /Users/claudiuciumedean/Storage/School/DTU/Thesis/language-agnostic-microservice-dsl

use plugins:
  mps
  java

macros:
  folder mps_home = ./../../../../../../../../Applications/MPS 2021.2.app/Contents
  folder user_plugins = ./../../../../../../Library/Application Support/JetBrains/MPS2021.2/plugins
  folder languages = ./build/artifacts/MicroserviceDSL
  folder outputFolder = ./../

dependencies:
  mps (artifacts location $mps_home)
  de.itemis.mps.extensions (artifacts location $user_plugins)
  MicroserviceDSL (artifacts location $languages)

project structure:
  solution HelloWorld
    load from ./solutions/HelloWorld/HelloWorld.msd

default layout:
  module HelloWorld
  folder Application
    files from ./solutions/HelloWorld/source_gen
```

18. If you get an error under **solutuion HelloWorld** click on the light-bulb and click **Reload Modules From Disk**.

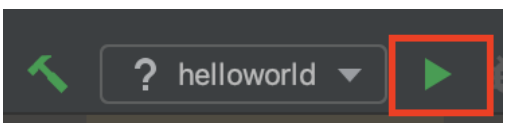


19. Add a **Run/Debug configuration** by clicking the highlighted buttons in the below image.



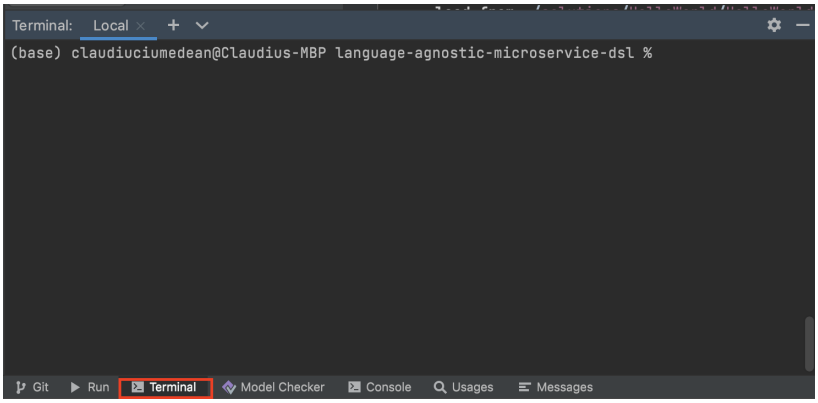


20. Build the project by selecting your newly added configuration and press play.



21. Open the **Terminal** tab in MPS and deploy the application following the steps. It is important that you installed **Node.js** and **Docker**, and have **Docker** running in the background.

    - In the terminal tab navigate to the root folder of the MaGiC framework.
    - Run the following command: **npm install**.
    - After NPM packages are installed, run the following command: **node build.js -l ./build/artifacts/HelloWorldBuild/Application/** . Again, it is important to make sure Docker is running in the background.
    - Open a browser window and navigate to **localhost:3000/home**.



22. Congratulations you have just built your **Hello World!** microservice-based web application using the MaGiC framework.