kunstWerkNg (kwNg)

A Rapid Application Development (RAD) Tool for Angular 6

Property of iTKunst corporation

August 14, 2018

pdooley@itkunst.com

Table of Contents

Introduction

"kunstWerk" is German for "Work of Art, or "Masterpiece". We agree that it is quite a grandiose name. We at iTKunst think that kwNg is deserving of such a name. It has been designed to allow one with limited experience to build SPAs of unlimited complexity, quickly and easily.

kWng

kWng is a declarative Angular application framework. I allows one to almost completely with declarations create a full application (custom components not included). And it does so dynamically. You need to change the signature of an API? No problem, modify the apis.json file and deploy - no build or debugging required.

kWng provides CRUD functionality right out of the box, and it provides simple debugging tools that help solve any problem in minutes.

kWng provides:

- Dynamic configuration of Apis and Data Models
- Dynamic configuration of Views and styles
- Dynamic configuration of data/view connectivity
- Support for multi-language
- Support for Application state (logon, logoff, register, retrieve pwd ...)
- Comprehensive tracing for debugging and quick development
- Combined development and test framework
- Support for Currencies, Languages and Timezones
- Support for Auth0 authentication and tokens
- Support for Authorization
- Support for Optimistic publishing, Automatic Retries, Cancellation
- Support for Application Caching
- Adheres to John Papa Coding speciifications.

Here are just some of the benefits of kWng.

kWng creates Meta Applications

By default the tool is set up to load all required data from json files at startup. However that same data could be retrieved via Ajax calls. This means that bootstrap data could be stored in a SQL database and updated from an admin portal. This functionality could also be exposed to the customer, to allow a certain level of customization. Allow being able to modify many aspects of an application remotely without any code changes. Once the changes are made, the next time the app launches, it is a new app. kWng allows one to create extremely flexible components quickly and easily.

kWng "talks" to the Developer

kWng has been designed to inform the developer what he/she needs to do next. The architecture is "aware". The code literally tells you what you need to do. Very similar to test-driven design, but it is embedded in the "live" architecture.

As an example, if you add controllers and services for a new API and you start the application, you will see errors in Chrome's dev console. i.e:

dlAssocCtrlApi ::load() api for **[associate]** is not provided.

What this is telling you is that the "associate" API is coded, but there does not exist an entry in the Apis.json file for it yet. It still needs to be added. You may see similar messages refering to missing "mdls" and missing "views".

It does not completely eradicate the need for tests, but it does provide a high level of certainty as to the quality of the code. For complex algorithmic code unit tests should be done, but really only in that case. It does strongly lessen the need to write unit tests for the apis - particularly when resources or time do not allow it.

kWng - A Lego Approach

If you look at the code in the App folder, you will see many folders and files, however, there is very little code in those files. They are simply gluing the states and services together. There is not much coding taking place because most of the code is in kWng. The process is essentially a simple automatic process of copying, pasting and renaming. In the near future a CLI will be created to handle. Study the state folders and you will see this very clearly. The comp folder will also look like this

Views are first created specifically in terms of the API calls. Again this is done to ensure that The APIs work - for ease of testing. There is very little code in those files. They are simply gluing the states and services to the views. The strategy used is to create a view for each API with three different components: the create view, the update view and the full display view. These are the visual lego bricks. Once these bricks are created, specific views can then be built from these blocks. (i.e. a creation wizard) In the near future a CLI will be created to create these components.

kWng automatically transforms Data Types

kWng transform data types. i.e. in MySQL, a boolean is returned as number. If the field in the model is encoded as a true boolean, it will automatically convert the number to boolean. And it will try to force it. Lets say it comes back as a string, the framework will try to convert it to a boolean anyways. And the same thing goes for sending data to the server. And if finally it is unable to convert it. It will tell you. The key thing is, if it is broken - if something has changed, you will know about it.

kWng provides powerful Tracing

One of the difficulties in SPA development is in ordering state change properly.. kWng allows one - with the flip of a switch to see the states change in real time. This allows one to see the "flow " of the application. It is extremely useful for debugging and understanding.

Bootstrap Settings

Here is where the important application settings are stored. This is found in the **src/assets** folder

General Settings

```
"sMode": "debug",
  "sRedirect": "http://www.itkunst.com",
  "bAutoLogin": true,
  "bTraceApp": false,
  "bTraceRoute": false,
  "bTraceState": false,
  "credentials": {
    "sUserName"    : "ironforgedev4@gmail.com",
    "sPassword"    : "12345678"
  },
  "display": {
    "bDispData": false,
    "bDispId": false,
    "bDispStyle": false,
    "bDispTag": false,
    "bDispView": false
  },
```

These top level settings are geared for simplicity in development. The allow the developer to see what the application is doing.

bAutoLogin (boolean)

If true is chosen, the application will attempt to login using the credentials provided in the credentials section. It will then go directly to the main page.

bTraceApp (boolean)

If true is chosen, the basic bootstrap and app status is traced, straight from startup ot logged in.

bTraceRoute (boolean)

If true is chosen, the frameworks own route tracing mechanism will be enabled. This functionality is not yet in place.

bTraceState (boolean)

If true is chosen, all state changes that take place will be traced. This is where any class like dlDealList, or dlDealView, or kwBSStatusVal, essentially anything derived from kwSt are storing information.

bTraceState is critical because it allows you to see the actual flow of the application and fix flow problems quite easily

display (object)

Display allows other aspects of the application to be viewed.

bDispData (boolean)

If true is chosen, the app will show the data objects for each component and page in **console.**

bDispView (boolean)

If true is chosen, the app will show the view objects for each component and page in **console.**

bDispId/bDispStyle/bDispTag (boolean)

if you set any bDispId/bDispStyle/bDispTag to true, the app will create a tag that will appear with the appropriate element and with the desired infoon the page.

If all three are set to true you would get **< kwNgPage style=default id=hello >**

If you set only bDispTag to true you would only get **< kwNgPage >**

If all three are set to false, not tags will appear.

Routes <object>

Routes is where the route paths are stored to specific application functionality. Naturally the values provided must be valid entries in the routing structure.

```
"routes": {
  "login": "login",
  "logout": "logout",
  "main" : "apps/deals",
  "register": "register",
  "reset": "reset"
 },
```

login <string>

login specifies which route should be activated when the application is ready to login the user.

logout <string>

logout specifies which route should be activated when the user has logged out of the application.

main <string>

Main specifies which route should be activated when the user has loggin successfully; It is the main landing page.

register <string>

Register specifies which route should be activated when the user has not yet been created and wants to register himself.

reset <string>

Reset specifies which route should be activated when the user has forgotten his/her password and wants to reset it.

Services <array>

Services is where the various Ajax services used by the application are defined.

```
"services": [
  {
    "nId": 0,
    "sCode": "dealwip_unsecure",
    "sHost": "//dealwipserver.herokuapp.com",
    "nPort": 8000,
    "sProtocol": "http"
  },
  {
    "nId": 1,
    "sCode": "dealwip",
    "sHost": "//dealwipserver.herokuapp.com",
    "nPort": 8000,
    "sProtocol": "https"
  },
  {
    "nId": 2,
    "sCode": "local",
    "sHost": "localhost",
    "nPort": 5000,
    "sProtocol": "http"
  }
],
```

In this case, three services are defined. One secure service, one unsecured service, and a local service for development. There is no limit to the number of services that can be defined.

nId <number>

This value is required by the framwork. It should be unique.

sCode <string>

This value is also required by the framework. This is how the various apis defined in apis.json are tied to the services. This value must also be unique.

sHost <string>

sHost defines the server to be used.

nPort <number>

nPort defines the port to be used for the service.

sProtocol <string> ["https", "http"]

sProtocol defines the protocol to be used for the service.

Credentials <object>

The credentials entry is used to define necessary information to pre-populate a login screen.

```
"credentials": {
  "sUserName"    : "ironforgedev4@gmail.com",
  "sPassword"    : "12345678"
},
```

sUserName <string>

sUserName is used to store the users id/email

sPassword <string>

sPassword is used to store the users password

AWS <object>

AWS is used to store the required paramaters for uploading and downloading documents from Amazon Web Store. It is currently still in development.

```
 "aws": {
  "bucket": "dealwip",
  "accessKey": "AKIAIORA26AL37IARCFA",
  "secretKey": "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",
  "region": "us-east-1",
  "folder": "jsa-s3",
  "serverEncryption": "AES256"
},
```

API Settings

Here is where (most of) the API information is stored. As you create APIs you add information here. The file is called **apis.json**. It is located in the **src/assets/meta** folder.

Single Record Actions

kwNg distinguishes between those actions that act on one record and those actions that act on multiple

```
"user": {
  "sMode": "debug",
  "actions": {
    "delete": {
      "live": {
        "sService": "dealwip",
        "sTemplate": "user/[0]",
        "sToken": "org"
      },
      "debug": {
        "sService": "local",
        "sTemplate": "user/[0]",
        "sToken": "org"
      }
    },
    "get": {
      "live": {
        "sService": "dealwip",
        "sTemplate": "user/get?user_id=[0]",
        "sToken": "org"
      },
      "debug": {
        "sService": null,
        "sTemplate": "./assets/data/user/user.json",
        "sToken": null
      }
    },
```

records (i.e. retrieve multiple records. Thus an entry for single records and for multiple recoreds is required.

```
  ("user" (cont))
   "post": {
      "live": {
        "sService": "dealwip",
        "sTemplate": "user",
        "sToken": "org"
      },
      "debug": {
        "sService": "local",
        "sTemplate": "user",
        "sToken": "org"
      }
    },
    "patch": {
      "live": {
        "sService": "dealwip",
        "sTemplate": "user",
        "sToken": "org"
      },
      "debug": {
        "sService": "local",
        "sTemplate": "user",
        "sToken": "org"
      }
    }
  }
},
```

<API Name> <object>

This is a sample entry for the user API. It is made up of actions and entries for debug and live are available. The API Name must be unique across the whole application and it is case sensitive.   It specifies the service it will be using. A debug and live version are provided and can be set. It is recommended that one create a json object and store it in the **src/assets/data folder** and point the debug entry to that.

sMode <string> ["live", "debug"]

The framework allows the granularity of determining whether an api may be live or debug. These values are loaded when the application starts up. Simply modify the value, save the file and restart the app.

actions <array> ["delete", "get", "post", "patch"]

Under the api name, four objects may be defined. All four are not necessary. These refer to the http action that is to be performed.

delete <object> ["debug", "live"]

This object describes the http delete action. It deletes a record on the server. There must be a debug and a live entry.

get <object> ["debug", "live"]

This object describes the http get action. It retrieves a record on the server. There must be a debug and a live entry.

patch <object> ["debug", "live"]

This object describes the http update action. It modifies a record on the server. There must be a debug and a live entry.

post <object> ["debug", "live"]

This object describes the http post action. It creates a record on the server. There must be a debug and a live entry.

sService <string>

Each action object must specify a service. This service entry must correspond to one of the services entries in the **bootstrap.json** file located in the **src/assets** folder.

If the action is created to retrieve a json object, set sService to null. Provide the path to the json file in the sTemplate field starting with **./assets/data.**

Set sToken to null.

```
"get": {
    "live": {
      "sService": "dealwip",
      "sTemplate": "user/get?user_id=[0]",
      "sToken": "org"
    },
    "debug": {
      "sService": null,
      "sTemplate": "./assets/data/user/user.json",
      "sToken": null
    }
  },
```

sTemplate <string>

The actions signature is stored here. If paramaters are required, specify them with the "[n]" indicator. The framework will take the necessary parameter and replace the indicator with the paramater at run time.

sToken <string>

If the action for the service requires a token, that information must be provided here. sToken refers to the list of tokens in the **bootstrap.json** file located in the **src/assets** folder.

If the action is simply retrieving a json file and a token is not required, then set sToken to null.

Multiple Record Action(s)

When the potential of retrieving multiple records exist, then a separate entry is required in the apis.json file. It typically has the plural name of the single object. All of the paramaters are the same as the single object

```
"users": {
  "sMode": "debug",
  "actions": {
    "get": {
      "live": {
        "sService": "dealwip",
        "sTemplate": "user",
        "sToken": "org"
      },
      "debug": {
        "sService": null,
        "sTemplate": "./assets/data/user/users.json",
        "sToken": null
      }
    }
  }
},
```

Running the App

1. Build Docker base image

run *`./baseBuild.sh`*

2. Build Docker image

run *`./build.sh`*

3. Run Docker image

run *`./run.sh`*

4. Debug app while in container

run *`./debug.sh`*

Application can be viewed at localhost:4204

5. Rectify node-saas Problem (If occurs)

```
exit docker
run `npm rebuild node-sass --force`
run `./run.sh`
This should resolve it
```

6. Build Release while in Container

run *`./release.sh`*

Application can be viewed at localhost:4204

To clear all docker artifacts

run *`./erase.sh`*

To view all docker artifacts

run *`./view.sh`*

Who is iTKunst?

“Kunst” is German for “Art”. We are artists who love to program, and love helping programmers. Our name reflects our philosophy. We strongly believe that the foundation of successful, innovative companies consists of:

1. Information,
2. Technology and,
3. Art.

Artists build software a little differently from main stream software developers; we model problem spaces. Modelling requires a unique way of thinking; An artistic way of thinking.

We at iTKunst have dedicated ourselves to the elimination of the Software Complexity Barrier (SCB). We believe that the promises of the software industry are yet to be realized and we are convinced that with the correct approach, no system is too complex. Say goodbye to bugs.

We have had many years of experience in building complex applications, particularly in building Single Page Applications (SPA). Over the years we have distilled the objects in the client application problem space and combined them in such a way so as to create a tool that dramatically simplifies and expedites the process of building SPAs - independant of the level complexity desired.