Pre-Requisites

This document is intended for developers that are familiar with SWP, RDF and various Javascript libraries. This is an advanced topic and while the intent is to be informative some content will appear out of place without prior knowledge to those technologies.

What is SWP?

SWP is short for SPARQL Web Pages, which is a model-driven template language and Engine. More information about SWP can be found at https://www.topquadrant.com/technology/sparql-web-pages-swp/,

What is RDF?

RDF short for Resource Description Framework is a W3C specification for data modeling. More information about RDF can be found at https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/.

What is React?

This documentation assumes a level of understanding and familiarity with the React library. It will not be covering React level features unless pertinent to the documentation. More information about React can be found at https://reactjs.org/.

What is Webpack?

Webpack is a Javascript bundler, TopQuadrant utilizes the Webpack build system to support the new React based feature sets. More information about Webpack can be found at https://webpack.js.org/.

What is Babel?

Babel is a Javascript compiler, in short it focuses on converting ES2015+ syntax into compatible versions for older browsers. More information about babel can be found at https://babeljs.io/.

If you find that you don’t already know the answers to those questions it will be best to review each of the technologies before continuing.

React with EDG

Starting in 6.0 EDG has started the process of introducing React based components. These components had a half in half out approach where React was only controlling a portion of the UI. Starting in 6.3 EDG made a major transition by committing to a full React based UI for the Editor portions of EDG. In making this transition new paradigms needed to be developed to provide similar extensibility to EDG as in the previous SWP based offering. This documentation will cover how to get started developing with these new paradigms.

Getting Started

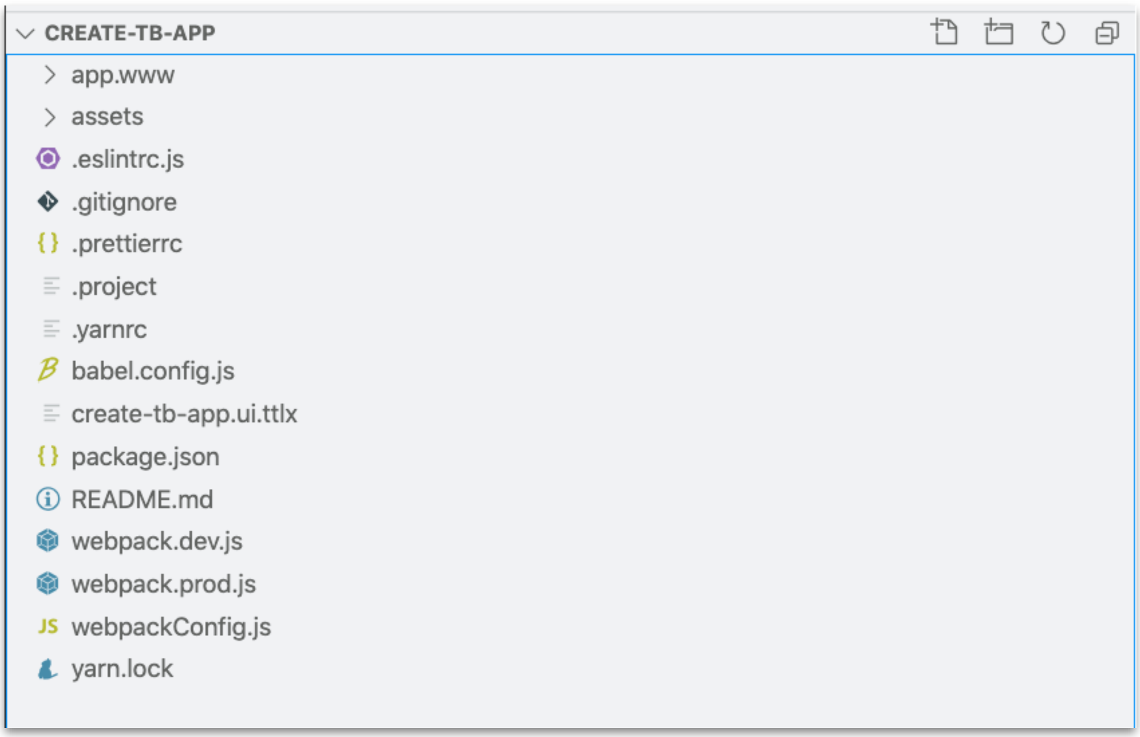
To help jump start the development process TopQuadrant has defined 'create-tb-app' a template that can be utilized to spin up extension projects that hook into the existing React layers.

What you'll need?

The template is a combination of TopBraid Composer project and a modern Javascript project. Create-tb-app takes advantage of the following technologies:

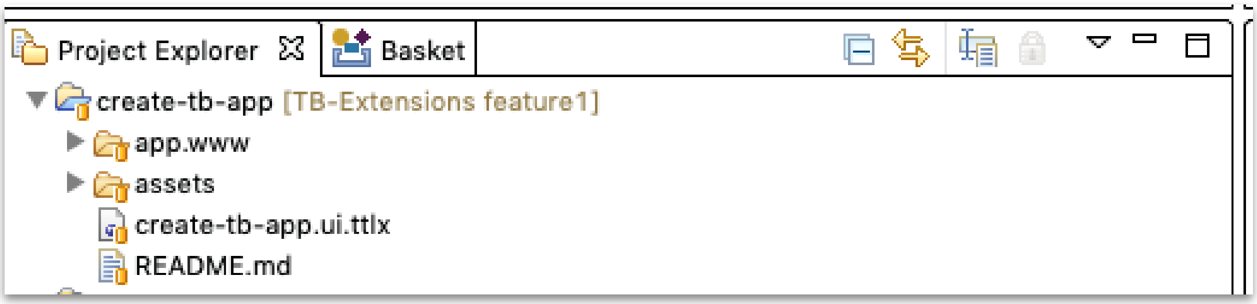
Node v10.19.0 (or greater)
Yarn v1.22.4
Webpack 4.44.0
Babel 7.10.5

Project Overview



```
CREATE-TB-APP
  > app.www
  > assets
    .eslintrc.js
    .gitignore
    .prettierrc
    .project
    .yarnrc
    babel.config.js
    create-tb-app.ui.ttlx
    package.json
    README.md
    webpack.dev.js
    webpack.prod.js
    webpackConfig.js
    yarn.lock
```

Hierarchy
FIGURE 1: CREATE-TB-APP HIERARCHY

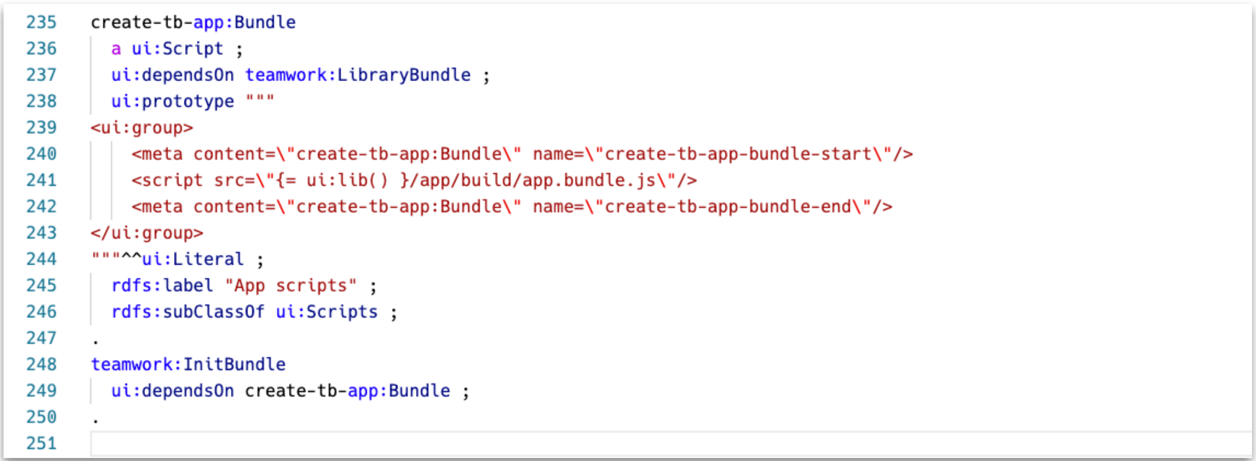


```
▼ create-tb-app [TB-Extensions feature1]
    ▶ app.www
    ▶ assets
      create-tb-app.ui.ttlx
      README.md
```

FIGURE 2: CREATE-TB-APP IN TOPBRAID COMPOSER

create-tb-app.ui.ttlx
This file is the glue if you will that binds the the server side to the client side. EDG makes use of React on the client side but still relies on SWP for static asset delivery and server side rendering of non React components. The use of SWP here allows EDG to assemble a chain of dependencies via instances of the **ui:Script** class. This feature allows third parties to extend the chain of dependencies through custom extension projects.

Below is an example of how to extend the dependency chain.

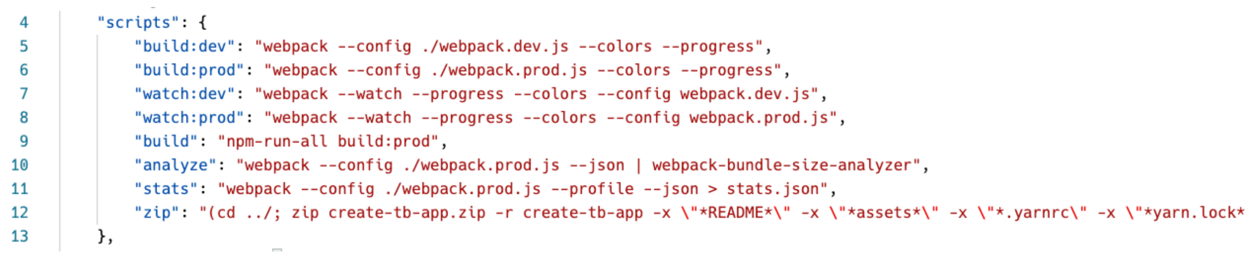


```
create-tb-app:Bundle
  a ui:Script ;
  ui:dependsOn teamwork:LibraryBundle ;
  ui:prototype """
<ui:group>
    <meta content=\"create-tb-app:Bundle\" name=\"create-tb-app-bundle-start\"/>
    <script src=\"{= ui:lib() }/app/build/app.bundle.js\"/>
    <meta content=\"create-tb-app:Bundle\" name=\"create-tb-app-bundle-end\"/>
</ui:group>
"""^^ui:Literal ;
  rdfs:label "App scripts" ;
  rdfs:subClassOf ui:Scripts ;
.
teamwork:InitBundle
  ui:dependsOn create-tb-app:Bundle ;
.
```

FIGURE 3: CREATE-TB-APP STATEMENTS.

package.json

package.json has two jobs, keep track of the Javascript dependencies and provide script shortcuts. There are several scripts but there are two of particular interest ‘watch:prod’ and ‘zip’. Watch:prod can be utilize to auto build the assets as changes are being made. Zip can be



```
    "scripts": {
        "build:dev": "webpack --config ./webpack.dev.js --colors --progress",
        "build:prod": "webpack --config ./webpack.prod.js --colors --progress",
        "watch:dev": "webpack --watch --progress --colors --config webpack.dev.js",
        "watch:prod": "webpack --watch --progress --colors --config webpack.prod.js",
        "build": "npm-run-all build:prod",
        "analyze": "webpack --config ./webpack.prod.js --json | webpack-bundle-size-analyzer",
        "stats": "webpack --config ./webpack.prod.js --profile --json > stats.json",
        "zip": "(cd ../; zip create-tb-app.zip -r create-tb-app -x \"*README*\" -x \"*assets*\" -x \"*.yarnrc\" -x \"*yarn.lock*
    },
```

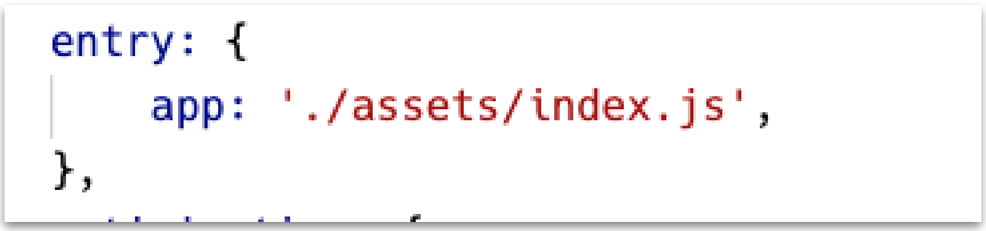
utilized to zip the current state of project and that zip can then be uploaded to an EDG server.

FIGURE 4: PACKAGE.JSON - SCRIPTS

How to call the scripts?
yarn run build:prod
yarn run watch:prod
yarn run zip

webpack.prod.js
wepback.prod.js is the production build definition for create-tb-app, while there are many configuration items in the file for this document we are only concerned about the entry section and the externals definition.

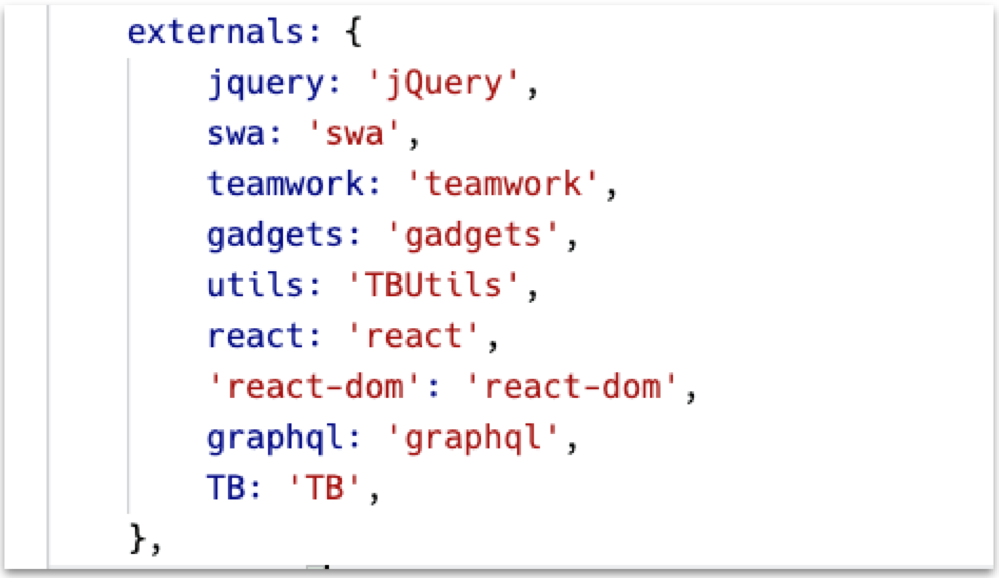
The entry section informs the build where to start the build process, in this case it’s



```
entry: {
    app: './assets/index.js',
},
```

assets/index.js (not shown in the previous screenshots).
FIGURE 5: ENTRY

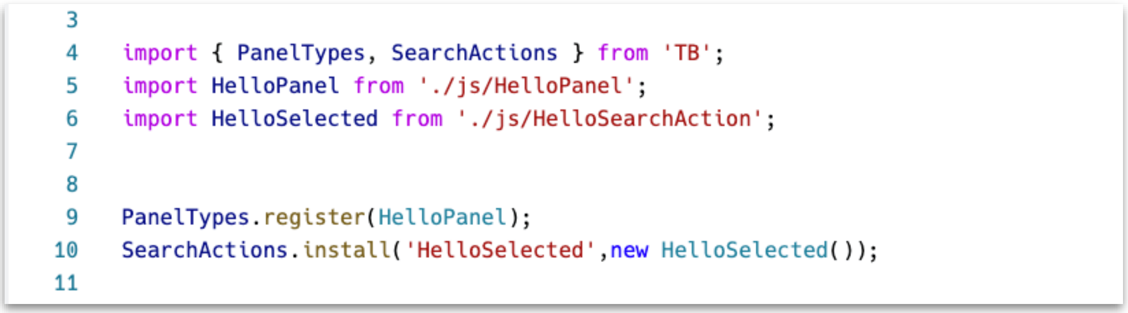
Externals are the libraries that the project is dependent upon, but will be provided external, ie from the global scope of the browser. Take note of swa, teamwork, gadgets, utils, and TB, these are all namespaces that are delivered via EDG.



```
externals: {
    jquery: 'jQuery',
    swa: 'swa',
    teamwork: 'teamwork',
    gadgets: 'gadgets',
    utils: 'TBUtils',
    react: 'react',
    'react-dom': 'react-dom',
    graphql: 'graphql',
    TB: 'TB',
},
```

FIGURE 6: EXTERNALS

index.js
create-tb-app provides a few examples of how to define custom components. Below index.js is registering the sample panel HelloPanel and the sample action HelloAction.



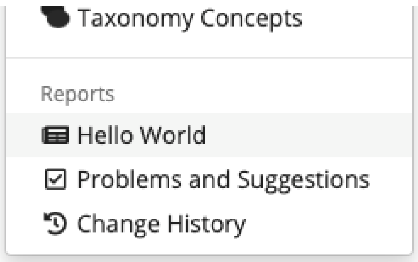
```

import { PanelTypes, SearchActions } from 'TB';
import HelloPanel from './js/HelloPanel';
import HelloSelected from './js/HelloSearchAction';


PanelTypes.register(HelloPanel);
SearchActions.install('HelloSelected',new HelloSelected());

```

FIGURE 7: INDEX.JS

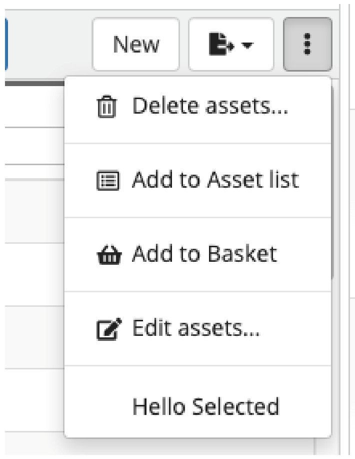
Results

So how does EDG look with these extensions installed?

A new Panel is available in the Panels menu:



A new search action is now available:



Next Steps

Now that we've covered what create-tb-app offers what are the next steps?

- Obtain create-tb-app
- Ensure the utilized technologies are installed (Node, Yarn)
- Copy the template
  - Because create-tb-app is a template, the best course of action is to duplicate the template and replace all instances of create-tp-app with your chosen name. This will allow you to re-use the template over and over.
- Install the dependencies (**yarn install**)
- Zip the project (**yarn run zip**)
- Deploy the project to an EDG instance
- Observe results