Blueprint Web Application a.k.a Nova

Requirements / High-level Design / Architecture

Version: 0.1

Table of Contents

[Overview 3](#_Toc448238526)

[Technology Stack 3](#_Toc448238527)

[JavaScript SPA Frameworks: AngularJS & TypeScript 3](#_Toc448238528)

[Module Loader 4](#_Toc448238529)

[Frontend libraries and restrictions 4](#_Toc448238530)

[Other JS Libraries 4](#_Toc448238531)

[User Interface Requirements/Decisions 5](#_Toc448238532)

[Web API 5](#_Toc448238533)

[Hosting and deployment 5](#_Toc448238534)

[Frontend Workflow Automation – Npm, Webpack & Gulp 5](#_Toc448238535)

[Nova prototypes 6](#_Toc448238536)

# Overview

With Microsoft planning on retiring Silverlight in near future and browsers starting to drop their support for Silverlight already, we have to adopt open web standards such as HTML5/JavaScript and replace Blueprint’s current offering that had been developed using Silverlight and along the way add support for tablet and mobile.

A Web based single page application (SPA) is an evolution away from the stateless page-redraw model that browsers were originally designed for but advancements in HTML5 and new JavaScript frameworks, enable browser to retain a single page and create a fluid user experience similar to a desktop application even when the application requires server communication.

Modern browsers that can parse HTML5 allow developers to shift the user interface and related application logic from web servers to the client and JavaScript libraries such as AngularJS, ReactJS, Ember, Meteor, … have adopted single page application (SPA) principles. HTML5 web-sockets also provide a bidirectional real-time client-server communication when needed.

Because SPA architecture moves the logic from the server to the client, the role of the web server can also evolve into a pure data API or web services enabling the server to be built as reusable discrete polyglot micro-services that are highly scalable, highly available and can independently be deployed in cloud environments.

# Technology Stack

## JavaScript SPA Frameworks: AngularJS & TypeScript

With AngularJS popularity and a large open source community behind it, not to mention the backing of Google, Angular has also been Blueprint’s JavaScript framework of choice. There is some uncertainty however with next version of Angular being completely different than Angular 1.x and because of it we need to design our code in a way that we are prepared for the changes which can be categorized into the following:

* Template Syntax
* ES2015 and Modules
* Types
* Components
* Bindings

In Angular 2, “components” are the main way elements and logic are built on the page, and Angular 1.x’s directives, controllers, and scope are all combined into components. New version of Angular 1.5 also introduced components syntaxes that should be used instead of separated views and controllers. Using TypeScript with Angular 1.5(x) provides a more clear migration path to Angular2.x.

TypeScript is superset of ES5 and it wraps ES2015 so we can use all ES2015 features and on top of that it adds types and annotations that come very handy in large projects such as ours and can improve JavaScript development experience by providing type information and displaying API documentation (“intellisense”) based on type definition files provided by “Definetely Typed” repo.

TypeScript is transpiled into JavaScript so our workflow need to include this step. We also need write our controllers and services as TypeScript classes.

Using TypeScript decorators is a neat feature that Angular 2 uses which could make Angular1 code look like Angular2 but instead of worrying about the looks and syntax, we need to use an architecture that is aligned with Angular2 which means no $scope anywhere and fully componentizing our app and thinking more about how components communicate with each other.

We also using SASS (scss) as CSS pre-processor.

## Module Loader

When we break application code down into one component per file, we often end up with a project structure with a large number of relatively small files. This is a much neater way to organize things than a small number of large files, but it doesn't work that well if you have to load all those files to the HTML page with <script> tags. Especially when you also have to maintain those tags in the correct order. That's why it's a good idea to start using a module loader.

Using a module loader such as SystemJS, Webpack, or Browserify allows us to use the built-in module systems of the TypeScript or ES2015 languages in our app. We can use the import and export features that explicitly specify what code can and will be shared between different parts of the application. For ES5 applications we can use CommonJS style require and module.exports features. In both cases, the module loader will then take care of loading all the code the application needs in the correct order.

When we then take our applications into production, module loaders also make it easier to package them all up into production bundles with batteries included.

For now we are using Webpack as our module loader.

## Frontend libraries and restrictions

* Angular – 1.5.x
* Angular UI Router
* Angular UI Bootstrap
* ag-Grid (tree and grid component)
* bowser (used for unsupported browsers detection)

All currently used libraries are pure Angular or JS, we are not using any jQuery functionalities and plugins.

## Other JS Libraries

We’ve already decided to use the following 3rd party libraries:

* mxGraph – as diagram library (currently in use in RapidReview)
* d3 – for light-weight graphing (currently in use in ImpactAnalysis)
* TinyMCE – for WYSIWYG HTML editing (currently in use in RapidReview)

It’s been also decided not to use KendoUI going forward and mainly rely on Bootstrap.

# User Interface Requirements/Decisions

For the first version of Nova UI Shell we are going with simplified options:

* Collapsible / Expandable panels with title when expanded (custom build component based on CSS and simple JS) for Explorer and Utility panel. Fixed panel size can be extended to have multiple positions or be resizable in the future.
* Accordion based Utility panel with pinned tabs (custom build component based on CSS and simple JS)
* Initial Shell is designed to run Storyteller in the different browser tab for the first phase. But it (ST) will be integrated into the shell content area in the future.
* We choose ag-Grid library for Explorer/Artifact Picker trees (it does not support lazy-loading and drag-n-drop out of the box, but this features can be implemented with coding and using another libraries). In the future we are going to reuse the same library for Artifact list grid. (Angular UI Tree is our fallback option for pure Angular tree component)

# Web API

AdminStore and ArtifactStore should be main placeholders for every new Web API endpoints used in `Nova` web application.

It is possible to call existing Web API designed for Rapid Review, Impact Analysis or Storyteller from Nova, but it should be avoided when possible.

# Hosting and deployment

We are building Nova application to live side by side with existing Silverlight and current HTML5 experiences.

Build task for Nova producing static files (html, minified and bundled css/js with app code and libraries and all used assets) inside NovaWeb folder and index.html (entry point for Nova UI Shell) that should we placed into the root folder of blueprint-current main site.

After that Nova UI Shell (index.html) will be default entry point for the Blueprint app site and will have ability to call existing Web API endpoints.

# Frontend Workflow Automation – Npm, Webpack & Gulp

Moving towards becoming independent of IDE, npm, webpack and gulp is to be used for automation of frontend workflow.

Npm is a single package manager used for both frontend and dev libraries. We are using different flags to separate front-end dependencies. Npm also used to automate dev processes (see scripts section in package.json). To support VS/VS Code Task Runners these scripts also available as Gulp tasks.

Typings is used to manage TS definition files.

Unit testing - Jasmine, Karma and PhantomJS.

Please look at <https://github.com/BlueprintSys/blueprint/blob/develop/app/NovaWeb/README.md> to get latest information about dev/build environment setup and usage.

# Nova prototypes

Multiple prototypes for Nova development can be found in:

<https://github.com/BlueprintSys/blueprint/tree/develop/prototype>

Instructions on how to run Nova prototype is provided as a README on same page.

Please note, at the time of developing of most of the prototypes, UX team were also working on their designs in parallel and the main goal of the prototypes were to help us select the widget libraries that support all the features we talked about as well as features currently supported in Silverlight. As a result the look and feel of the prototypes currently does not necessarily match UX team designs and are more in line with the current Silverlight version.