Sprites

High-level application design and notes

Table of Contents

Introduction

Below sub-sections outline the purpose of this document, give recommendations on the target audience as well as provide audit information related to this document alone.

What This Document Is About

This document gives a high-level overview of the design and implementation of Sprites application and its components. Sprites is an online tool for creating and publishing interactive, animated infographics and presentations.

Who This Document Is For

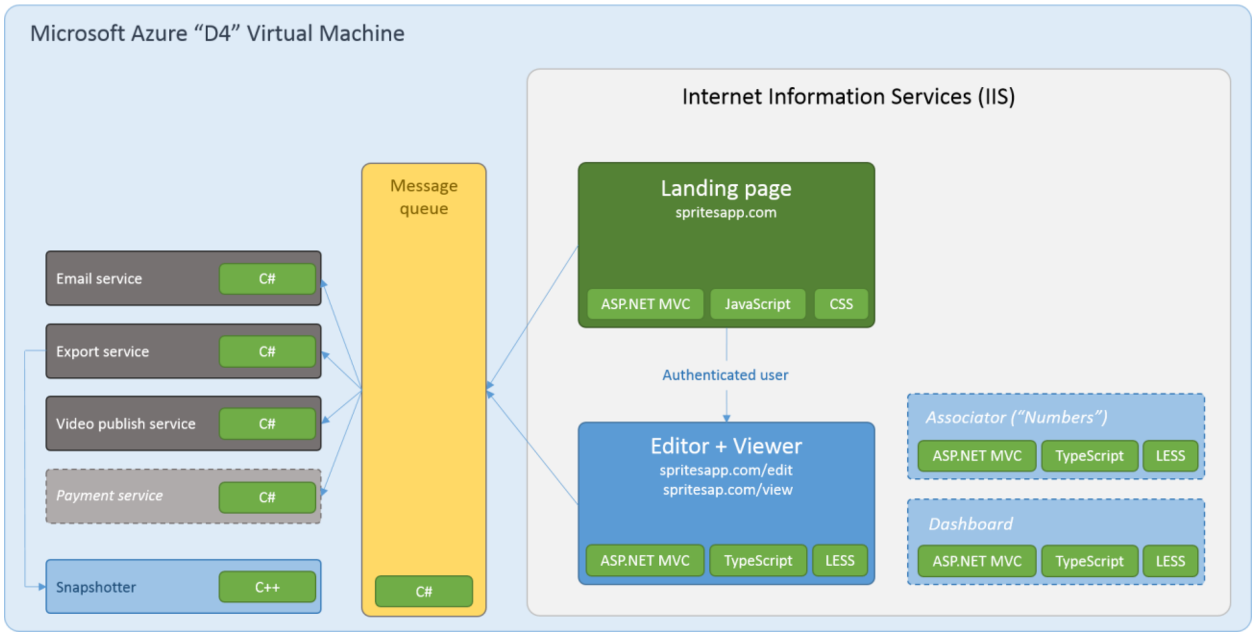
This document is for software engineers wishing to get a high-level overview of Sprites application structure as well as some of the implementation details.

Revision History

Version number	Publish date	Author(s)	Comments
1.0	January 9th, 2016	Pavel Volgarev	First version.

Application Structure

Below diagram shows different components from which Sprites application is composed:



Picture 1 – Application components

There're several main components:

1. **Landing page**
   Contains product description, quick links sign-up and login forms as well as the account overview page (spritesapp.com/account).

2. **Editor + Viewer**
   Main component responsible for editing and displaying of an infographic/presentation. Although, it's the same piece of code, it's organized as two different Virtual Applications (with their own Application Pools) within IIS.

   Rationale: Scalability. The two components, when separated, provide a higher level of reliability when it comes to ASP.NET application recycles caused by resource re-allocation, re-deployment of an application, extreme load, etc.

3. **Associator ("Numbers")**
   Promotional piece representing a web-application which is capable of comparing objects by their quantitate characteristics using the given input (e.g. "3 meters is one-hundredth of a height of Eiffel Tower) by leveraging Worfram|Alpha technology for computations. This component has been discontinued.

4. **Dashboard**
   A web-application for viewing custom analytical data about Sprites (sign-ups, usage patterns,

etc.). This component has been discontinued (replaced by Google Analytics leveraging custom goals).

5. **Message queue**
   Custom, XML- and file system-based, implementation of a message queue used for communicating with external services (such as email dispatcher, image/video export broker, etc.)

   Rationale: Scalability and Cost-Effectiveness. A message queue improves overall performance of a software system by allowing services to pick up and execute new units of work as resources become available. Having a custom implementation of a message queue eliminates additional expenses associated with purchasing a service from various cloud providers and allows for an application-specific, tailored implementation. It's worth mentioning, though, that at some point the Amazon SQS was used beneath the abstraction layer of a Sprites message queue.

6. **Email service**
   Responsible for filtering and routing incoming email messages ("Contact us" form submissions, error reports, etc.) for delivery via SendGrid API.

7. **Export service**
   Responsible for triggering jobs associated with exporting an infographic into PNG, PDF or a video. It's worth mentioning that the actual job of producing the output is handed over to a native component (Snapshotter) whereas the Export service is only responsible for gathering the input parameters and passing them to Snapshotter application (which, in turn, is implemented as Windows console application).

   Rationale: Functionality and Scalability. The Snapshotter is a crucial part of Sprites application, delivering one of the most-used pieces of functionality. It was, therefore, decided to avoid any wrapper-/runtime-associated management and performance costs and implement Snapshotter as fully-decoupled, native piece.

8. **Video publish service**
   Responsible for publishing exported infographic videos to Facebook and YouTube by using the corresponding APIs.

9. **Payment service**
   Responsible for periodically checking for expiring subscriptions and sending out email reminders. This component has been deactivated for a time being.

10. **Snapshotter**
    A native component responsible for capturing image/video output of a given resolution from a given URL.

Not showing on a diagram above is Sprites blog which is hosted at GitHub (source code is publicly available at https://github.com/spritesapp/spritesapp.github.io).

Rationale: Simplicity. Having a statically-generated website hosted on GitHub Pages makes it extremely easy to iterate on the new/updated content and removes the potential extra costs (both financial and in terms of time/risks) in managing a dedicated CMS-based solution.

Editor + Viewer

The following sub-sections describe (at a high level) how Sprites Editor application is implemented. As mentioned in the previous section, the Editor shares the code-base with the Viewer, therefore, when referred to as “Sprites Editor” or “Editor”, the “Viewer” component is also implied.

Frameworks and Tools

Sprites Editor is an ASP.NET MVC4 web-application with the back-end implemented in C#. It’s a fairly standard setup in terms of middleware and asset pipeline:

- Markup is organized in a set of partial Razor views;
- For minification and bundling, default mechanisms are used (no CDN);
- Client-side piece is communicating with the back-end via a set of WebAPI REST endpoints;
- Standard SignalR setup for real-time communications (team chat, simultaneous editing);

On the front-end, LESS is used for organizing all the CSS stylesheets whereas all the client-side logic is implemented in TypeScript.

Rationale: Maintainability and Familiarity. TypeScript is a good choice for writing more robust and efficient JavaScript. On top of a number of quite useful features it has (modules, interfaces and inheritance, generics, etc.), there’s also a great support for it in Visual Studio (full IntelliSense, “Go to definition”, refactoring, auto-generation of source maps). It's also a tool of choice within many other projects that the author has been working on previously. It's worth mentioning, though, that with an expansion of ECMAScript 6 (ES6) and transpiler tools like Babel, converting the implementation to "vanilla" JavaScript would not possess a big effort.

Other JavaScript libraries and tools used within the application:

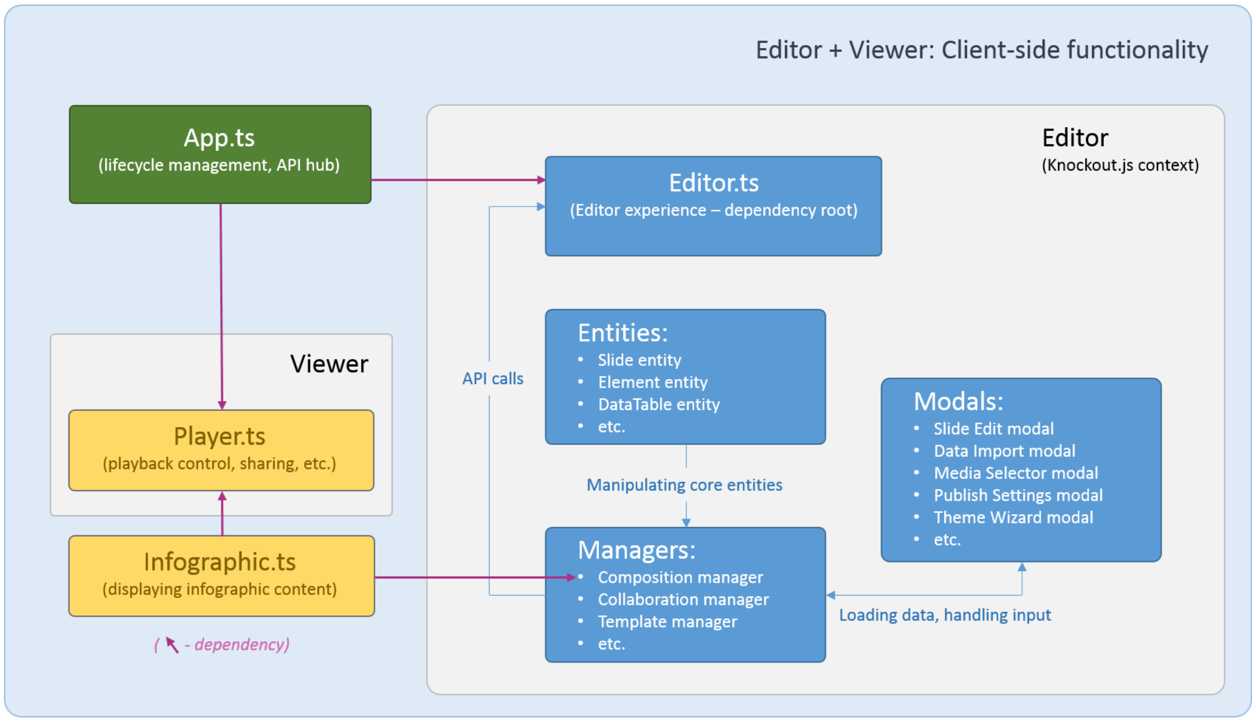
- jQuery (including jVectorMap for rendering SVG-based maps);
- Knockout (MVVM implementation – used for describing pretty much all UI interactions);
- Google Visualization API (rendering of all the charts);
    - In Compatibility Mode, Sprites uses Chart.js for rendering charts. *Compatibility Mode* is a set of flags which make it possible for existing infographics to still be used/edited with the legacy functionality while all the new infographics receive and updated set of features;
- ACE Editor (code editor for maintaining custom widgets);
- Moment (handling of dates);
- Q (a polyfill for ES6 promises);

For storing and retrieving data (within Editor in particular and within application in general), a NoSQL solution was chosen – RavenDB.

Rationale: Simplicity and Scalability. At the initial design phase a few things become apparent: entities were quite complex and so it made sense to represent them as graphs or documents, fast writes for the good editing experience were required, models would evolve fast and so it was preferable to avoid RDBMS enforcing any kind of schema, the use LINQ for querying data was essential. RavenDB was written specifically for .NET, it's quite easy to set-up and effortless to use in a default configuration, it's *fully transactional, with configurable consistency model. It's worth mentioning that the author did use MongoDB on his previous projects, but was left with mixed feelings (its tendency to loose data even when there's almost no load on the system, was quite frustrating). Also, a research into using Azure DocumentDB was made but this solution turned out to be very costly, financially.*

High-Level Design

Below diagram provides a high-level overview of the application design (structure as well as the main moving pieces):



Picture 2 – Client-side responsibility break-down

The responsibility of each module is the following:

1. **App.ts**
   Responsible for (pre-) loading application assets (scripts, fonts, images, etc.) as well as providing an interface for communicating with the back-end APIs.

2. **Editor.ts**
   Serves as a dependency root for all the UI managers. Also handles editor-specific lifetime events (such as resizing, changing of target infographic, etc.)

3. **Entities**
   A set of core entities – Presentation, Slide, Element, DataTable, etc. These are, to a large extent, pure models and encapsulate no business logic.

4. **Modals**
   TypeScript representations of a UI pieces (modals). Not shown on the diagram, but part of the design are smaller UI building blocks (special input types, data grid, etc.)

5. **Managers**
   A self-contained pieces of business logic, objects managing certain area of the Editor (for example, Composition manager is responsible for updating the infographic as changes are being made while Slide manager is responsible for adding/deleting/reordering/etc. of individual slides).

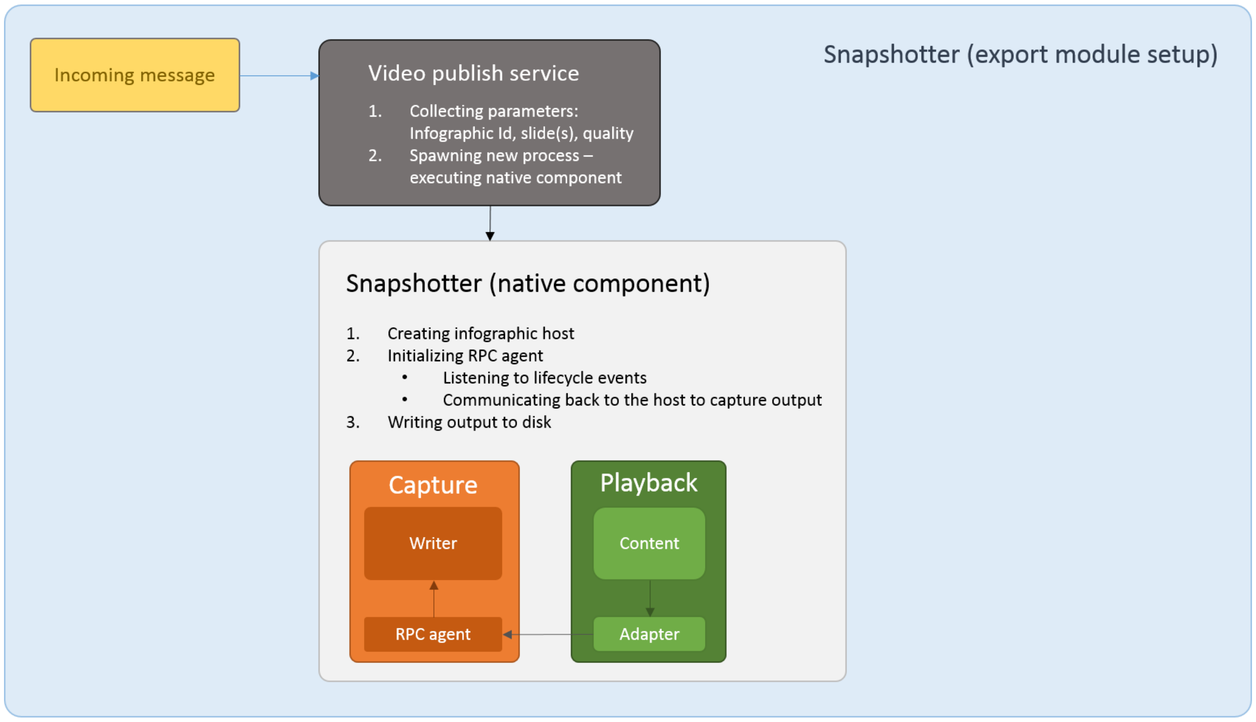
6. **Infographic.ts**
   An implementation of an infographic. Responsible for rendering elements of different types (charts, tables, maps, etc.) Also responsible for preparing the canvas area (for example, applying a theme) as well as for some of the interactions (conditional drag'n'drop, highlighting, etc.). Has no dependencies on the Editor experience.

   Rationale: Modularity and Maintainability. Having a core presentation logic decoupled from the rest of the UI opens up many different scenarios in which this logic can be re-used (for example, loading an infographic in a modal window when recording narration or rendering elements in different contexts such as Google Chrome plug-in)

7. **Player.ts**
   An infographic playback experience. Also responsible for communicating with an export agent when an infographic is being saved as an image or a video.

Video Export

One of the most-appreciated features in Sprites is an ability to export an infographic into an HD-video. Below is a diagram providing a high-level overview of the design of this component:



Snapshotter (export module setup)
Incoming message
Video publish service
1. Collecting parameters: Infographic Id, slide(s), quality
2. Spawning new process – executing native component
Snapshotter (native component)
1. Creating infographic host
2. Initializing RPC agent
• Listening to lifecycle events
• Communicating back to the host to capture output
3. Writing output to disk
Capture
Writer
RPC agent
Playback
Content
Adapter

Picture 3 – Export module setup

There're two main actors in the above setup:

1. **Video publish service**
   Responsible for receiving export parameters (which are passed from the user via the message queue) and spawning a new process for capturing image/video data from the given URL (which is composed based on input parameters).

2. **Snapshotter**
   A component responsible for loading the given URL into a "hosted environment" and listening to lifecycle events which, when triggered, cause the image/video stream to be forwarded to a file.