

Appendix

A1

Related patent list:

#	Code	Type	Application ID	Filled	Publication ID	Also published as
1	WE	US	13/596,146	28/8/2012	US 2013-0086496	
2	DL	US	13/771,119	20/2/2013	US 2013-0219263	
3	SH	US	13/779,798	28/2/2013	US 2013-0232403	
4	MP	US	13/786,488	6/3/2013	US 2013-0238977	
5	CA	US	13/959,759	6/8/2013	US 2014-0237429	US9513771
		US-Cont	15/339,984	1/11/2016	US 2017-0116761	
6	TASS	US	14/053,614	15/10/2013	US 2014-0108373	US9436765
		US-Cont	15/233,987	11/8/2016	US 2016-0350422	
7	TACA	US	14/176,166	10/2/2014	US 2014-0229821	
8	LA	US	14/207,761	13/3/2014	US 2014-0282218	
9	MAC	US	14/483,981	11/9/2014	US 2015-0074516	
10	TDC	US-CIP	14/559,943	4/12/2014	US 2015-0089354	
11	ACS	US	14/619,903	11/2/2015	US 2015-0227533	
12	ACD	US	14/207,930	13/3/2014	US 2014-0282130	
13	ML	US	14/699,828	29/4/2015	US 2015-0310124	
		US-Cont	15/653,568	19/7/2017	---	
14	EXP	US	14/926,007	29/10/2015	US 2016-0124839	
15	CBCM	US-CIP	14/619,145	11/2/2015	US 2015-0154164	
16	ABCO	US	15/168,295	31/5/2016	US 2016-0350794	
17	ACE	US	15/175,272	7/6/2016	US 2016-0357527	
18	WT	US	15/224,616	31/7/2016	US 2017-0032050	
19	SE	US-CIP	15/292,172	13/10/2016	US 2017-0046317	
20	MLX	US-CIP	15/224,579	31/7/2016	US 2016-0371312	
21	OB	US	15/607,586	29/5/2017	---	
22	SBOX	US-Prov	62/367,151	27/7/2016	---	
		US-Prov	62/531,897	13/7/2017	---	
23	AWC	US-Prov	62/437,324	21/12/2016	---	
24	WC3	US-Prov	62/516,682	8/6/2017	---	

Appendix

A2

WIXCode

Can you create a website without coding a single line?

YES

Can you add animation to the site?

YES

Add a registration form to the site?

YES

Show some text only if the user register to my site?

Kind of...

Build your own custom menu?

NO

Can you show a different promotion based on the weather?

NO

A Site for Organizing Riding Groups?

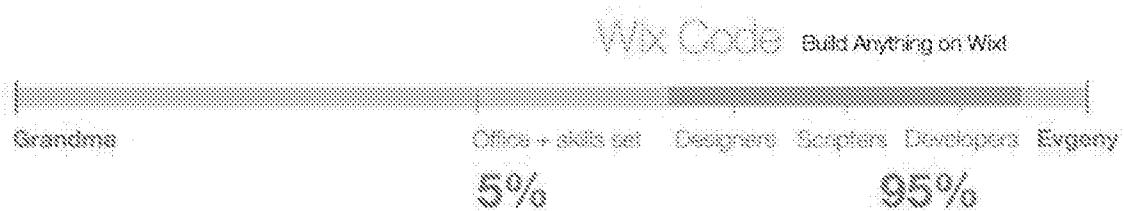
NO

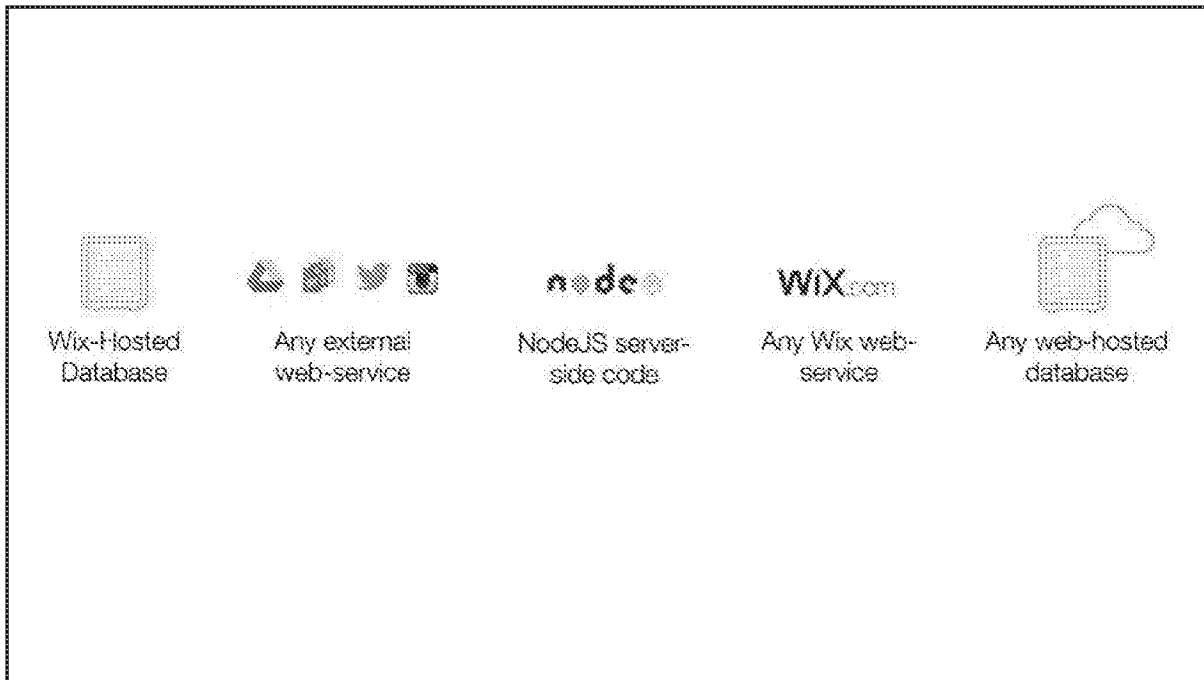
Wix code enables

To add missing functionality to the site

To develop a web application

Easy to use programming paradigm combining drag & drop with coding





Appendix

A3

Wix Code

Technological indepth overview

Index

Chapter 1: The cloud landscape

Chapter 2: What makes wix code (into a platform for developers)

Chapter 3: what's included - the features in teckish

Chapter 4: how it works -

- a. yaniv's slide cold bootstrap problem
- b. SEO

Chapter 5: WixCode X as a service / OS

Chapter 1: The Cloud Landscape

The Classic Cloud Landscape

Software as a Service (SaaS)
Finished application to rent and customize

Platform as a Service (PaaS)
Developer platform that abstracts the
infrastructure, OS and data storage

Infrastructure as a Service (IaaS)
Deployment platform that abstracts the hardware

Chapter 2: What makes Wix Code

What's in Wix Code

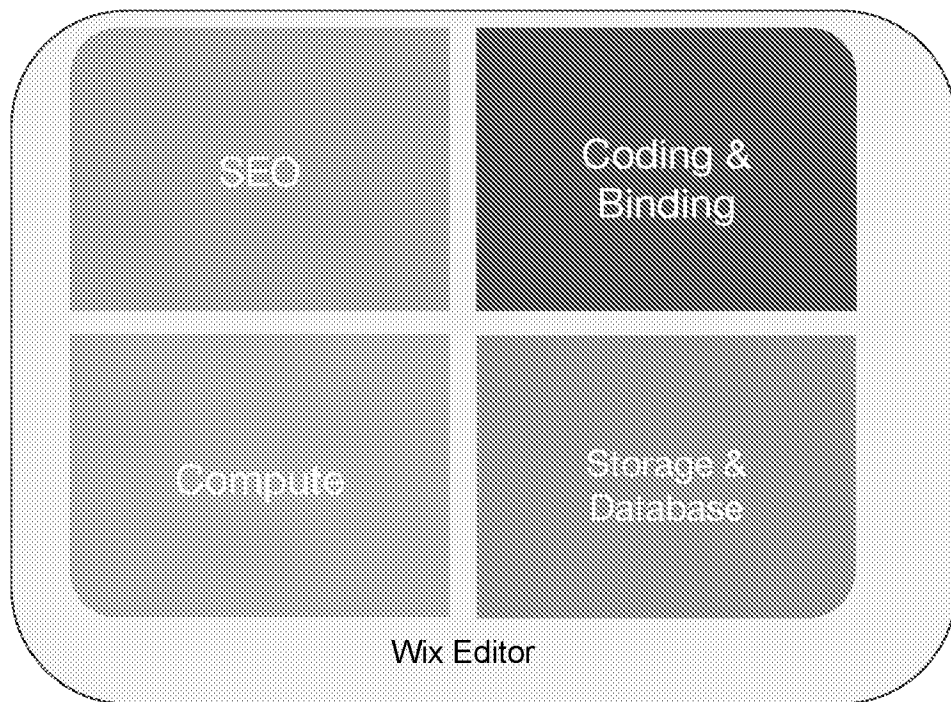


Wix Editor

What's in Wix Code



What's in Wix Code



Compute

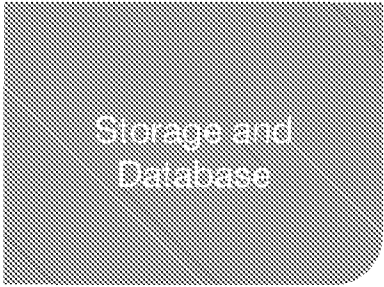
What it is

- Serverless, no hassle coding
- Javascript backend runtime

What Compute includes

- Web Methods
- Compute Functions
- Routers
- Dynamic Pages Hooks

Compute



Storage and
Database

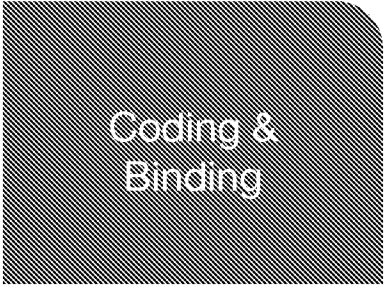
What it is

- Full managed database
- Soft Schema

What it includes

- Content Manager
- Wix Data API
- Media Platform
- Database Hooks
- Permissions

Compute



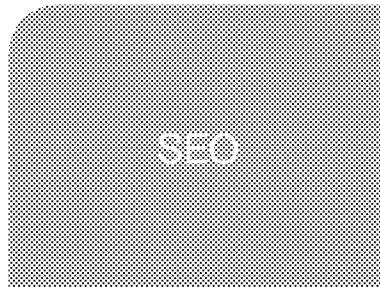
Coding & Binding

What it is

- Client APIs for the site
- No coding app to connect the site with data

What it includes

- \$w API & Page code
- Dynamic Pages
- Data Binding
- User Inputs & Submit Button



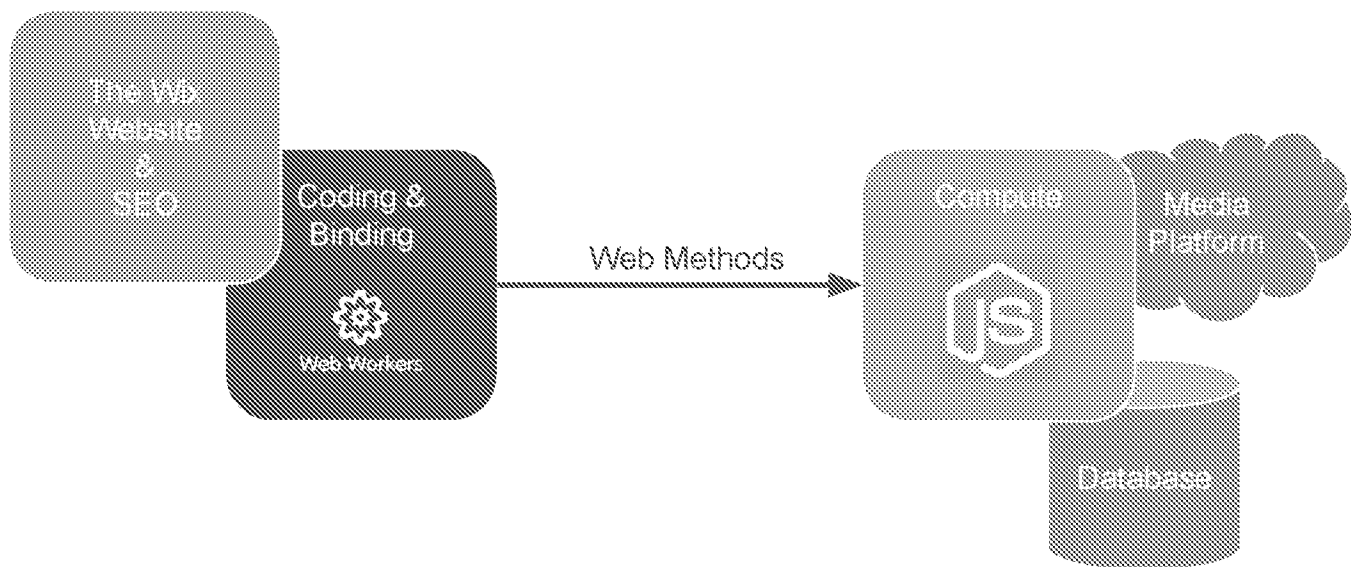
What it is

- Wix SEO
- Full support for Wix Code

SEO for

- \$w API Coding
- Binding
- Dynamic Pages
- Router

How the parts fit together

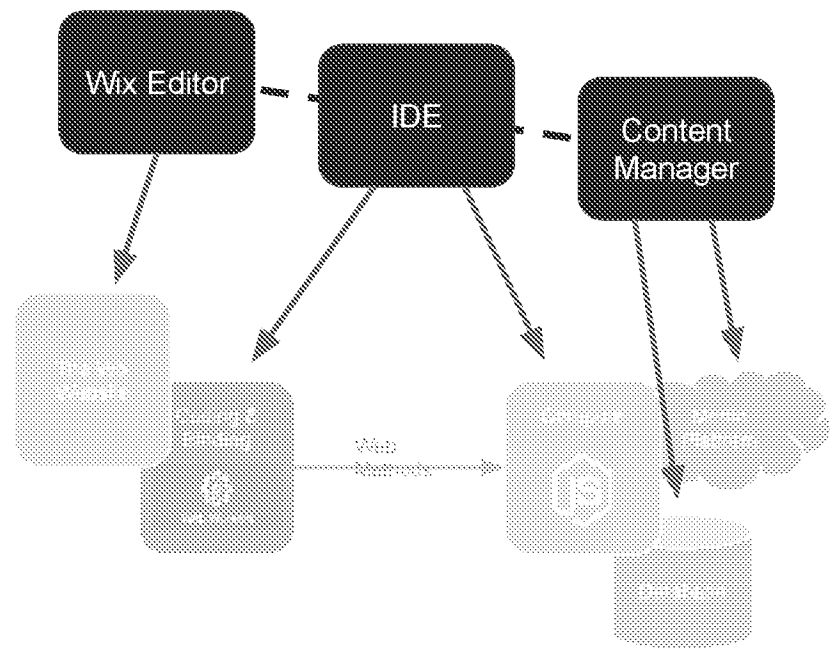


Chapter 3: What's included

Zero Setup

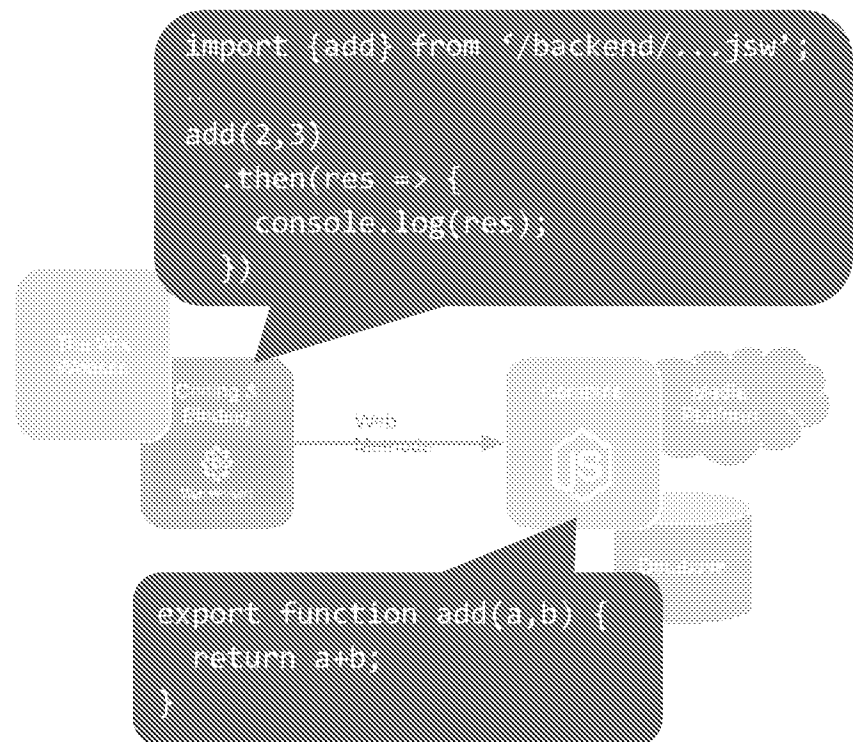
No setup to start building a web app

- Zero setup Database
- Zero setup JS Backend
- Zero setup development
 - Zero setup IDE
 - Zero setup Content Manager
 - Zero setup Wix Editor
- Zero setup Deployment
- Zero setup Javascript Build
 - ES6 Transpiling
 - Bundeling
 - Minification
- Zero Effort Uptime



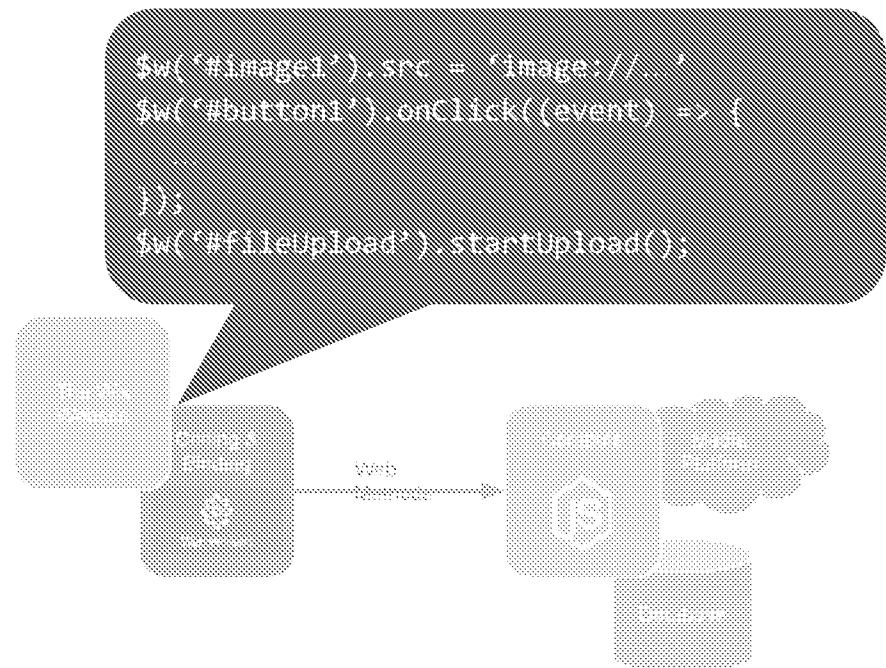
Web Methods

- Auto exposure of backend (compute) operations to the client
- Auto handling of console messages
- Auto handling of errors
- Permissions model - who can call which method



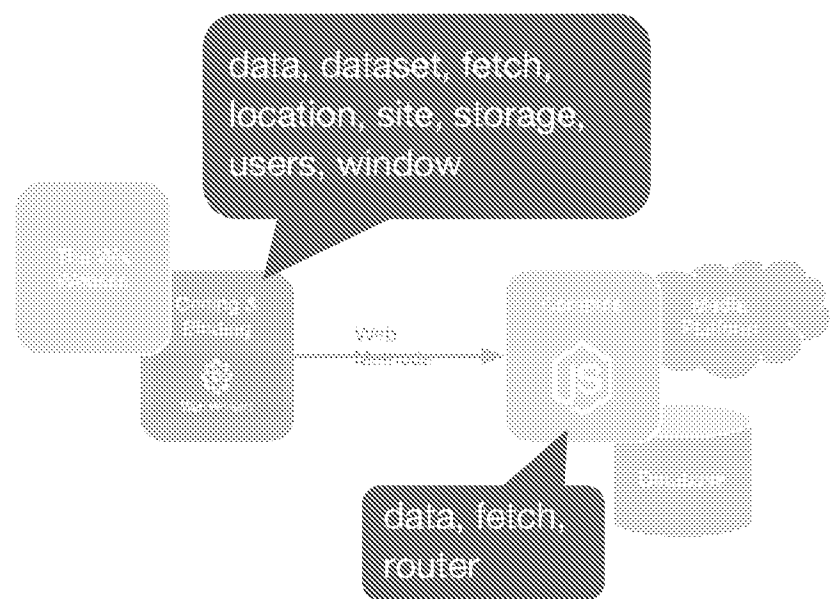
API to the Wix Elements

- \$w API - to select Wix Elements, change properties and listen to events
- Text, Image, Galleries, Shapes, Columns ... Elements
- Inputs & File Upload Elements
- Table Element
-



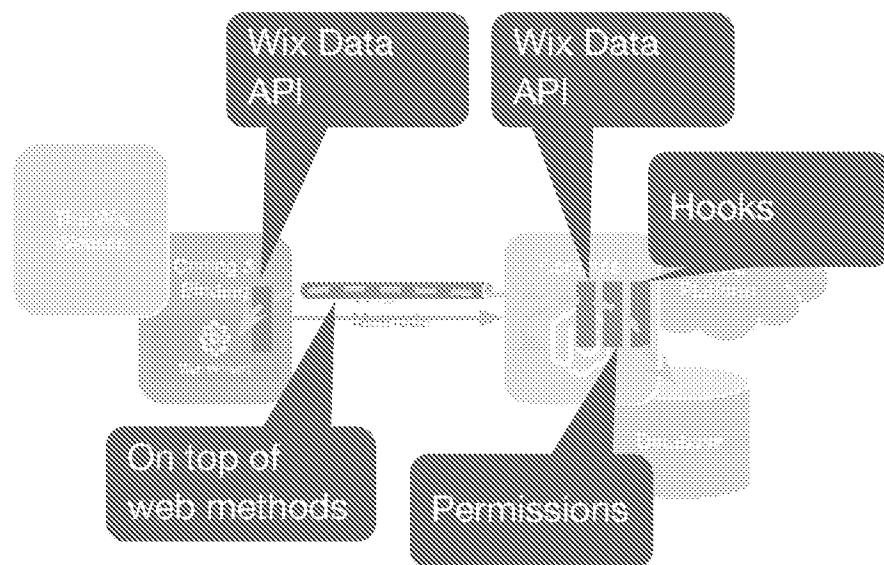
More APIs

- Wix Data - the database API
- Wix Fetch - the HTTP client API
- Wix Location, Site, Users, Window - to manage current browser and page view
- Wix Storage - local and session storage
- Wix Router - to manage custom routing logic



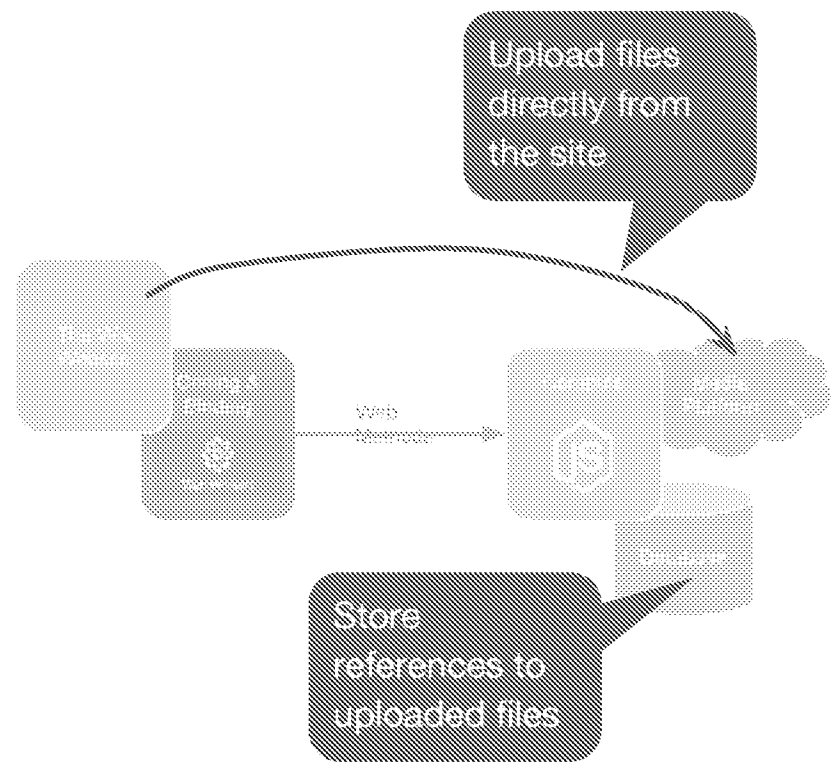
Wix Data

- Schemaless database API
- Accessible from client or backend
- Permissions
- Backend hooks
- Wix Data API
 - get
 - query
 - count
 - insert
 - update
 - delete
 - Includes (joins)



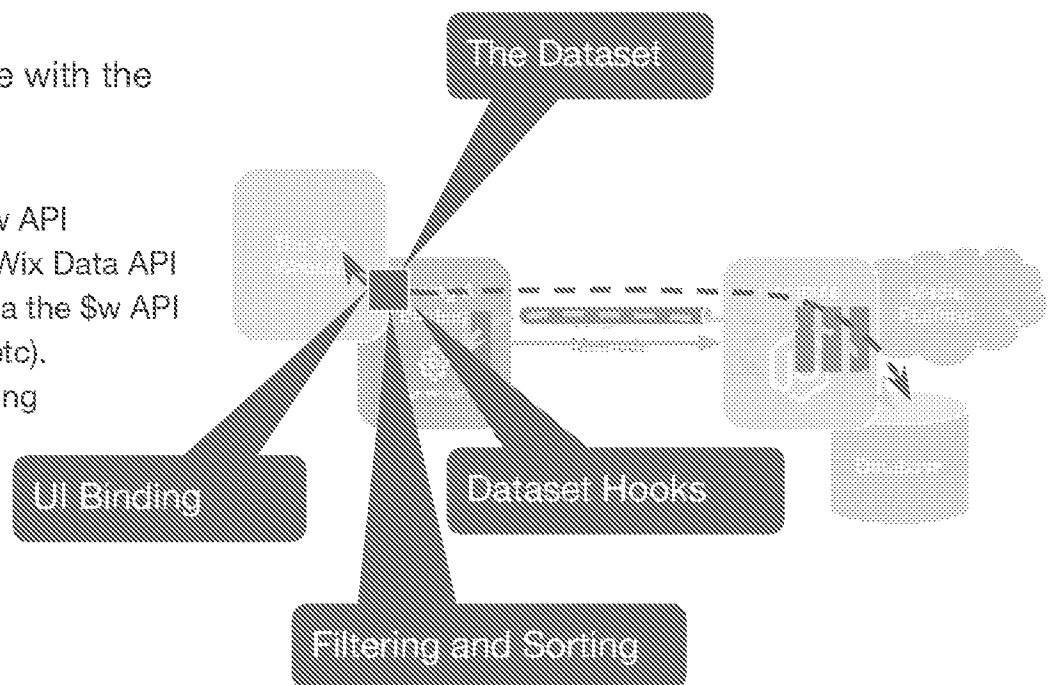
Wix Media Platform

- Site users can upload directly to the site storage in Wix Media
- Store Wix Media URLs (references to uploaded files) in Wix Data
- Full support in Content Manager
- Full support in Binding



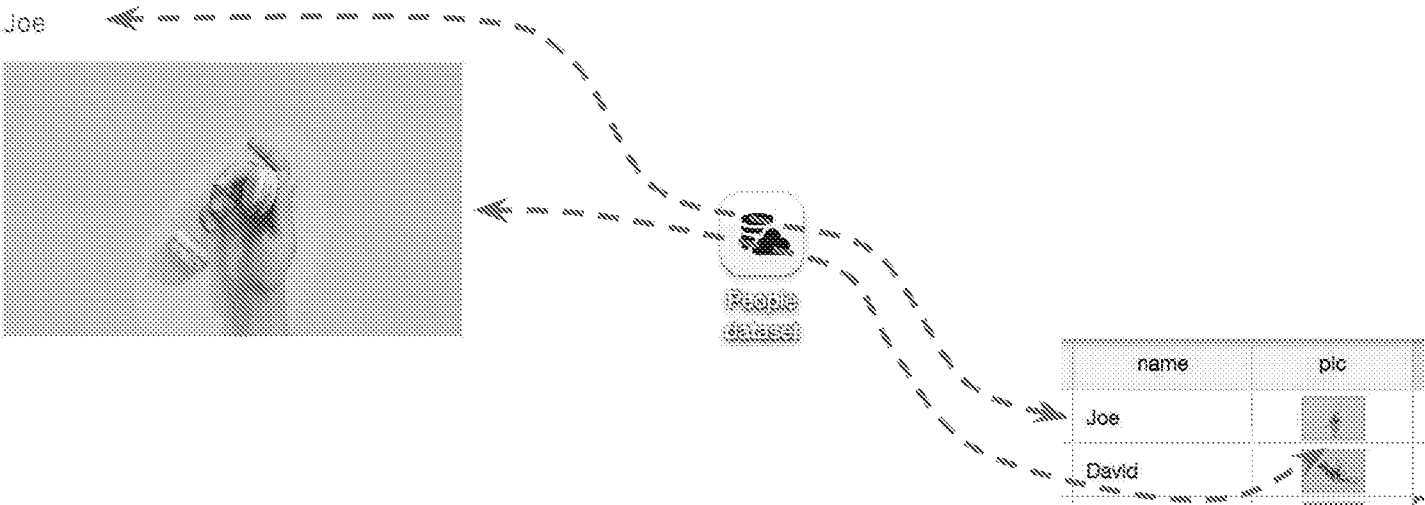
Data Binding

- Connect the database with the Wix site
- The Dataset
 - Binds UI via the \$w API and Database via Wix Data API
 - Binds UI actions via the \$w API (next, prev, save, etc).
 - Adds filtering, sorting
- No coding needed



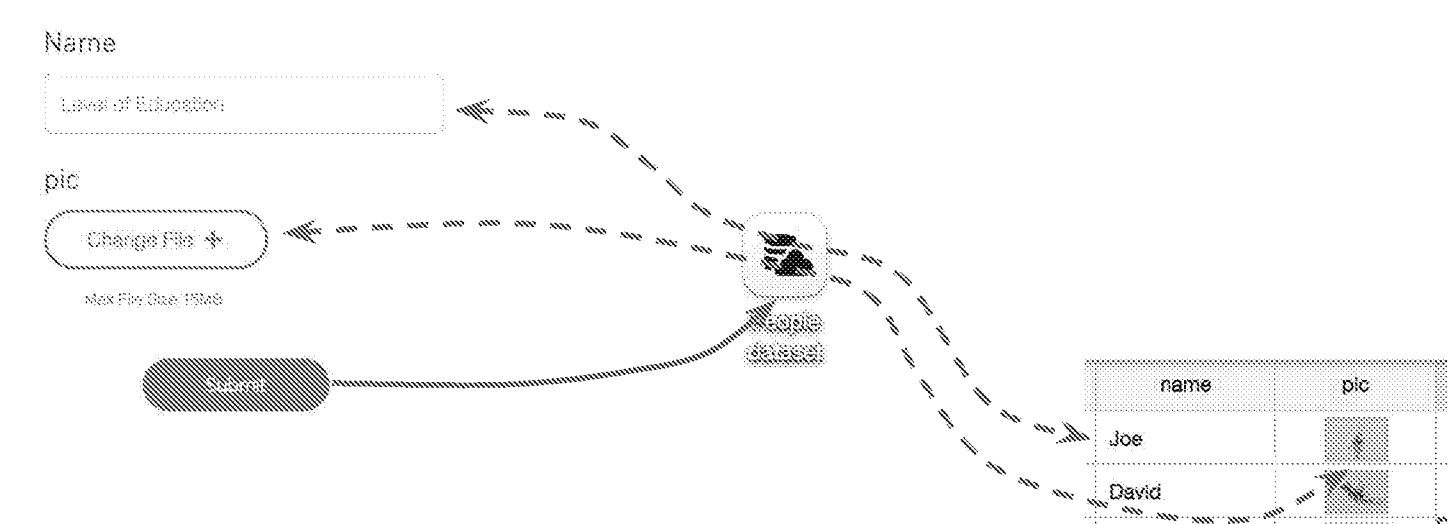
Data Binding

For displaying data



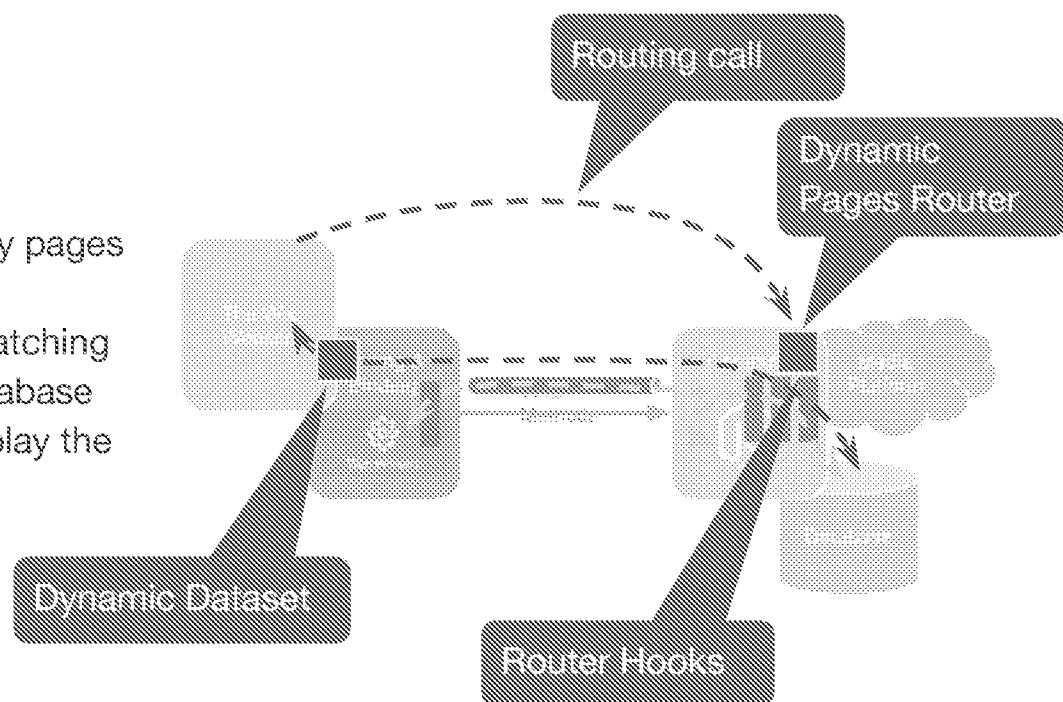
Data Binding

For submitting and editing data



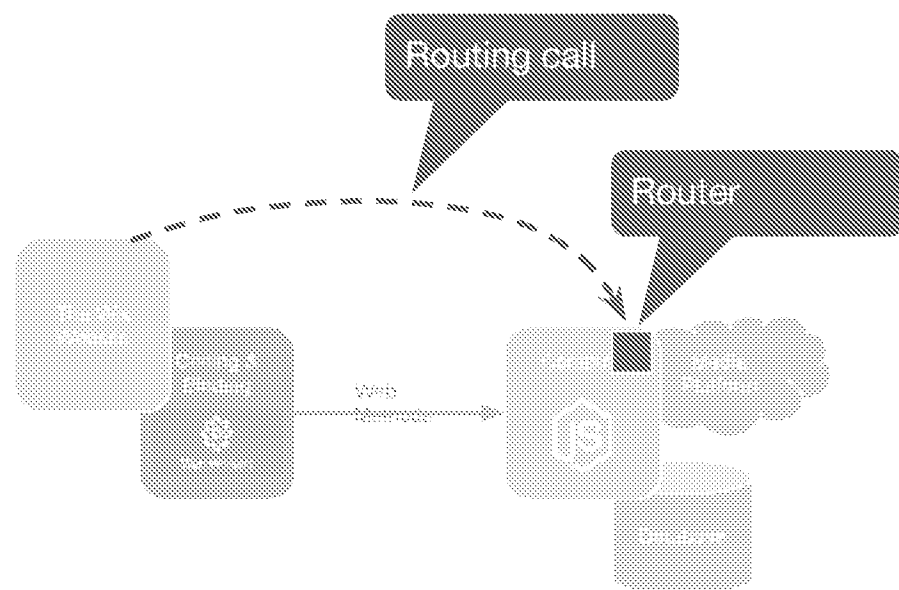
Dynamic Pages

- Design page once, get multiple site pages
- Create item and category pages with any URL pattern
- Given a URL, find the matching item (or items) in the database
- Use Data binding to display the item(s) on the page
- Can be customized via code hooks



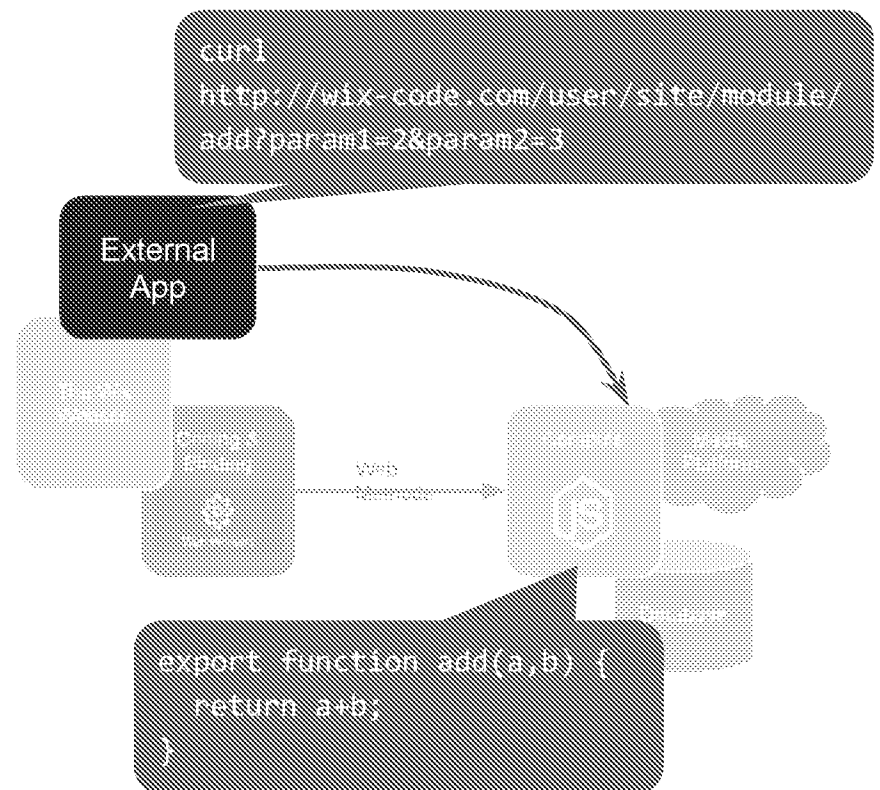
Router

- Even more control compared to dynamic pages
- Trades binding with the power of coding



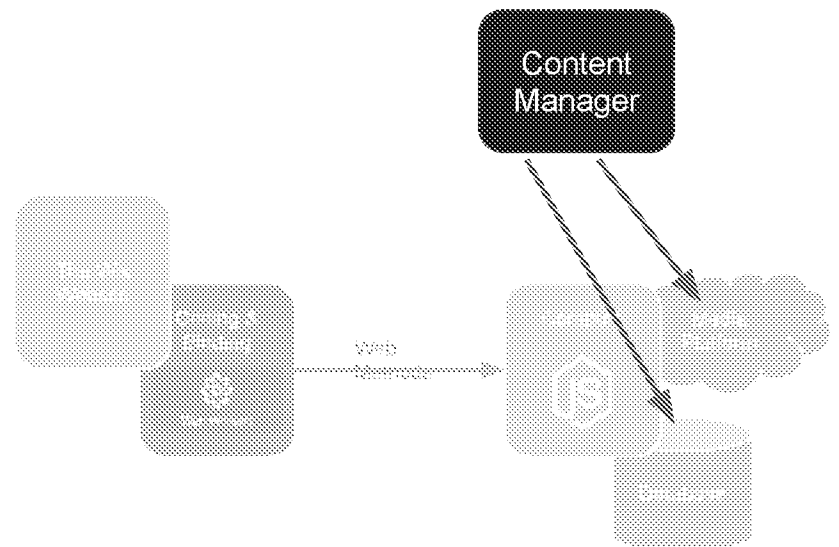
Compute Functions

- Allow arbitrary calls from the web to the backend
 - Such as curl
- Great for web callbacks
 - Such in the case of payment gateways



Content Manager

- To manage the live site data
- Excel like
- Update data without opening the editor
- Upload and Download files
- Built on top of wix code compute and database

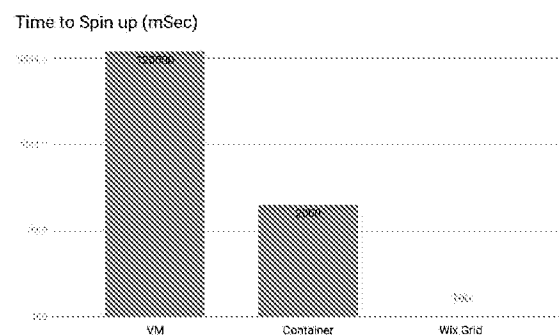


Chapter 4:

How it works - the compute grid

The Compute Grid

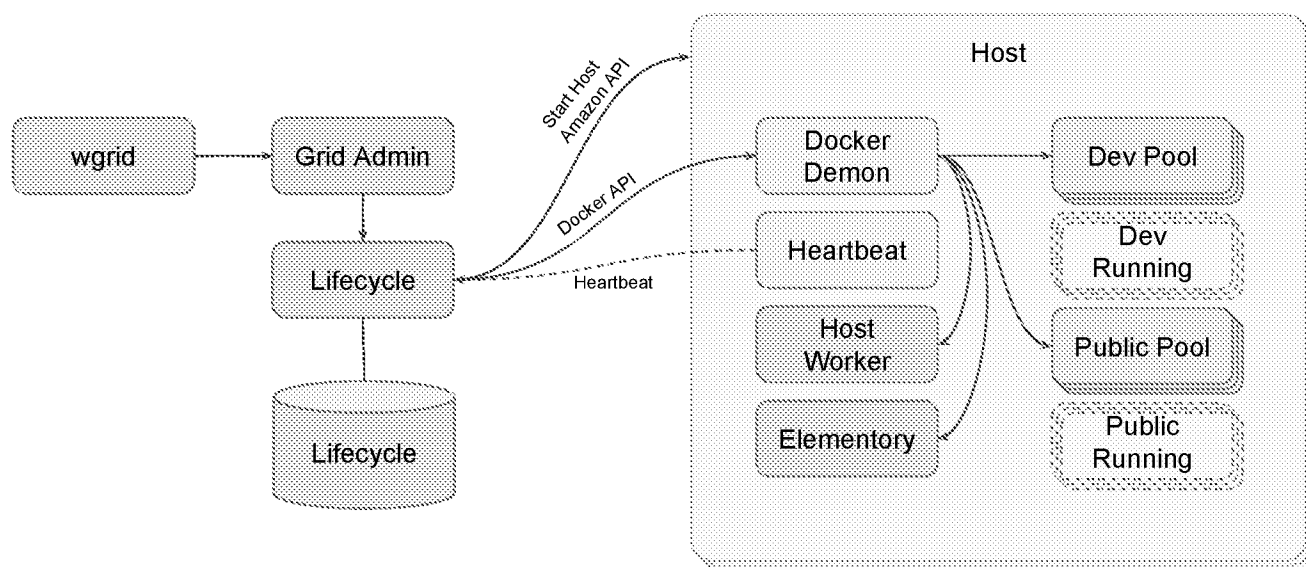
- Running user backend code requires strong isolation -
 - Requires dedicated host like VM or Container
- Dedicated backend for millions of users -
 - Dedicated host is not a practical solution
- Spin up a host on demand
 - VM takes ~2 minutes
 - Container takes ~2 seconds
- Dedicated hosts provide the required isolation and resource management, but none of the cloud providers can guarantee fast enough spin time
- **Wix Compute Grid**
 - Spin up time ~100 mSecs
 - Spin up a backend within web request time



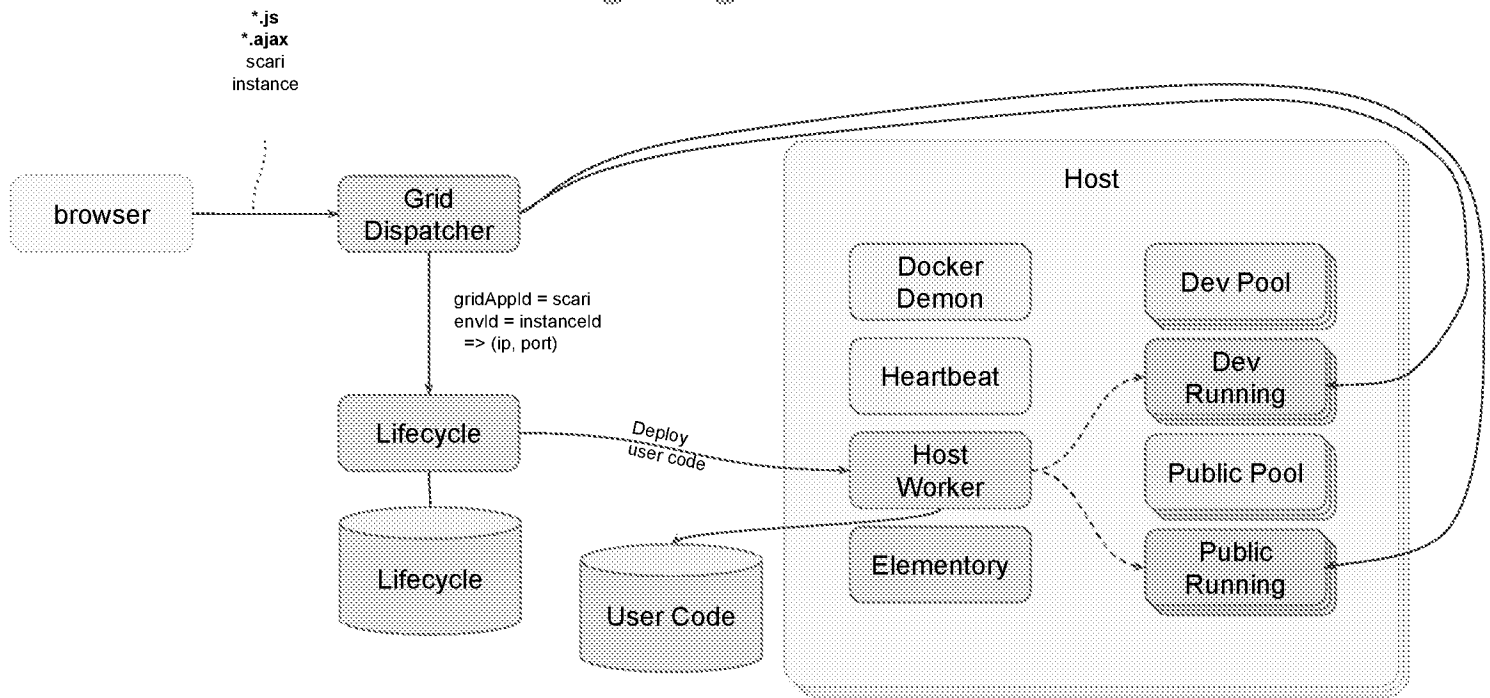
Search Engine Compatibility

- Web Applications are, by default, not search engine compatible
- Google runs Javascript
 - But Facebook, Twitter, Bing, Yandex and others do not...
- JQuery (market share 97%), the most used Javascript framework -
 - is not SEO compatible
- MooTools (market share 4%), the next most used javascript framework -
 - Is not SEO compatible
- Vue.js (0.1%), Angular (0.6%) & React (0.03%), the new kids on the block -
 - Are not SEO compatible by default
 - But are SEO compatible with a non trivial setup
- **Wix Code SEO**
 - Is compatible with all search engines and social networks
 - Including any \$w Javascript coding
 - Including all Wix Code features - binding, dynamic pages, router, etc.

Host Lifecycle



Assigning a Container



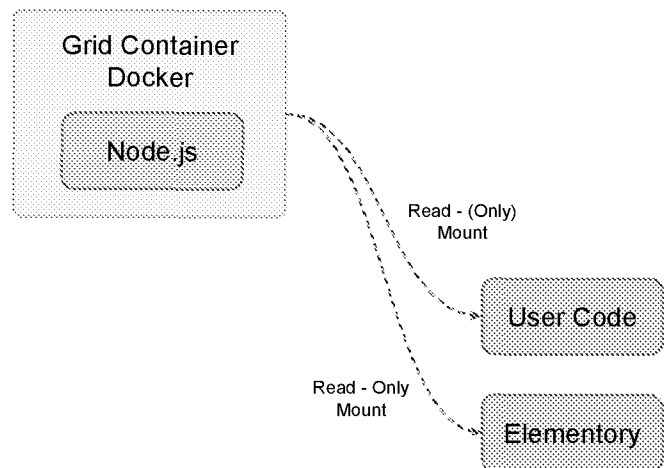
A Container

The Container

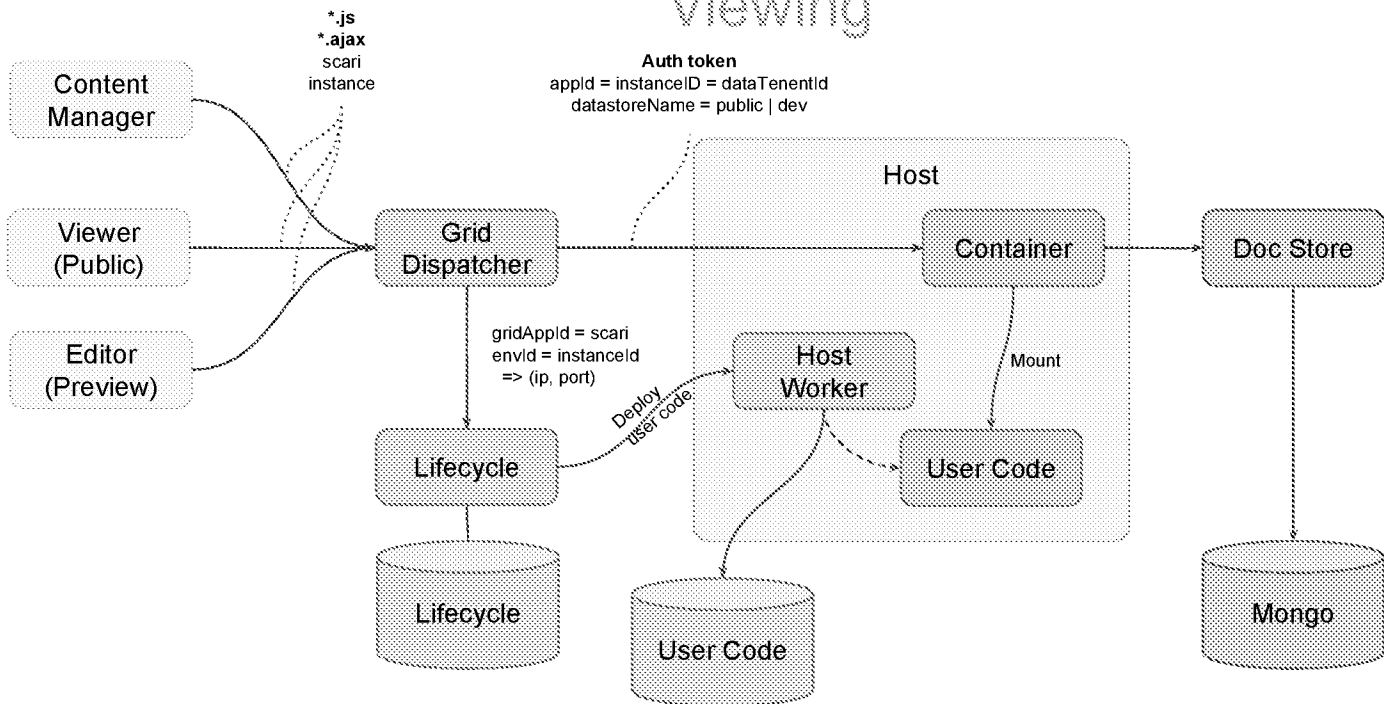
- Everything stripped out, including shell, gcc, gdb, etc.
- Does not run as Root
- Root FS of container is Read Only

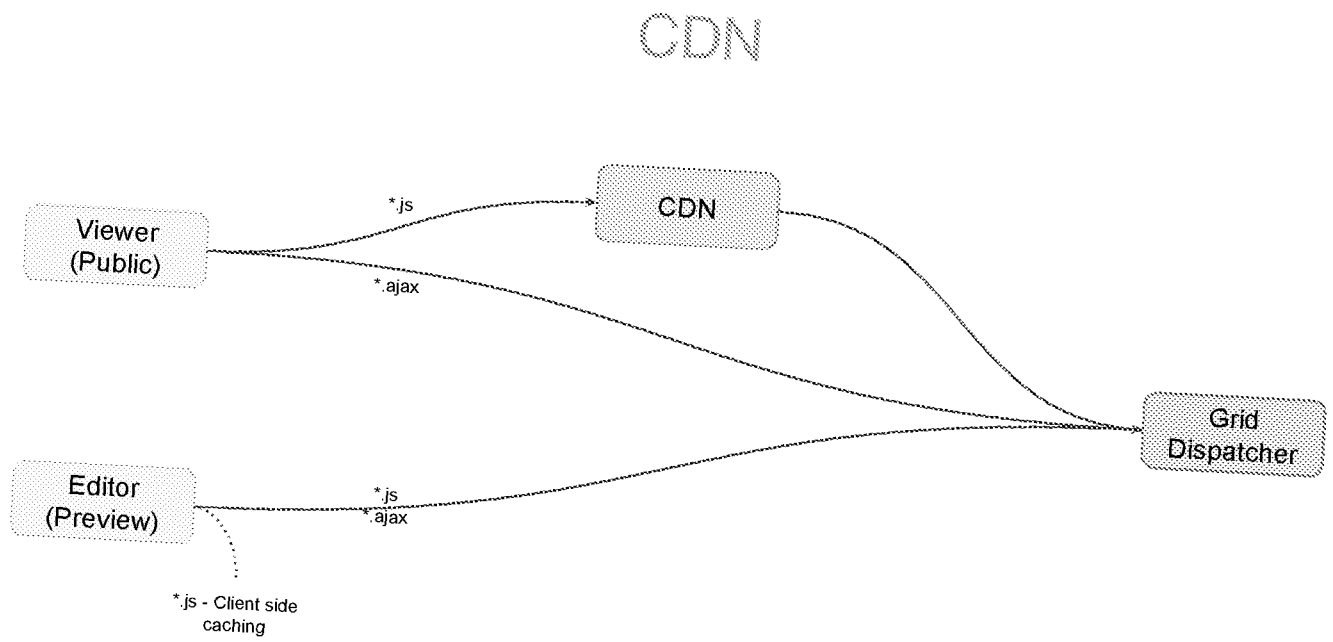
Internal Tool to create image with

- The container
- The mounts
- The operating system utils, like shell, less, ls, etc.

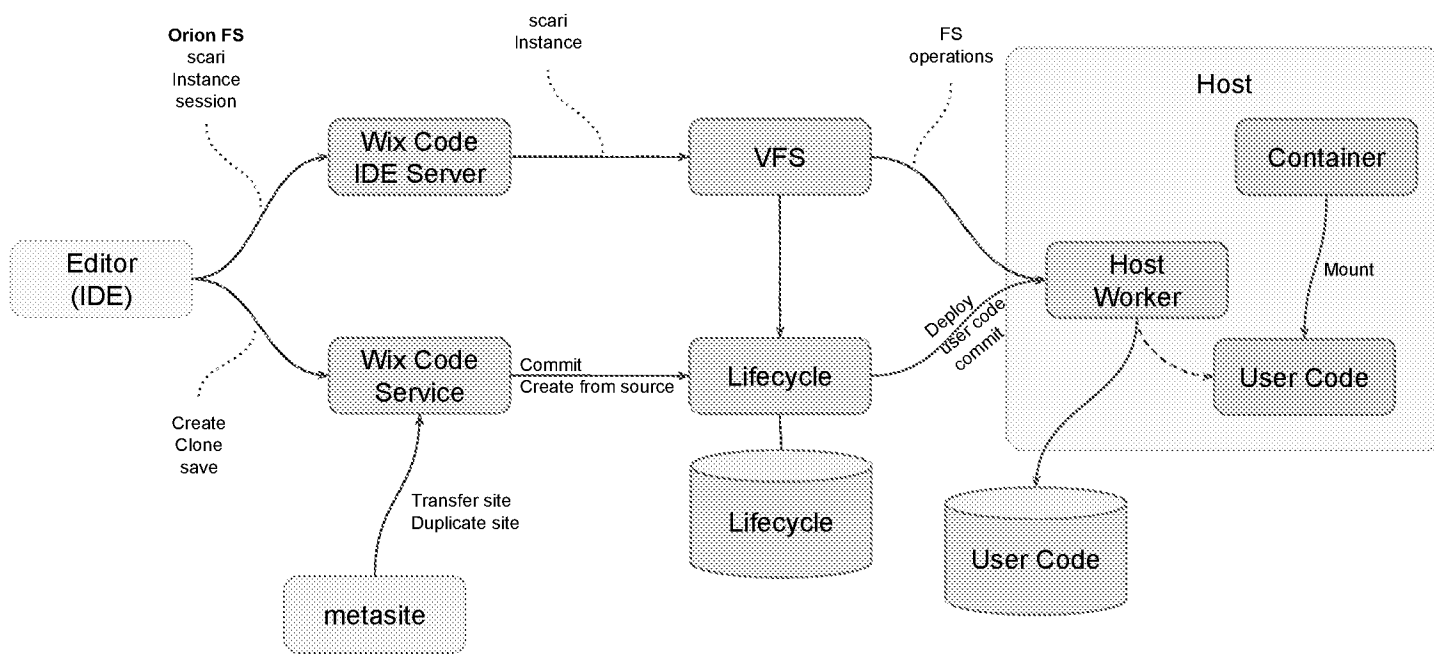


Viewing





Editing Code



Chapter 5: summery Wix as X as a Service

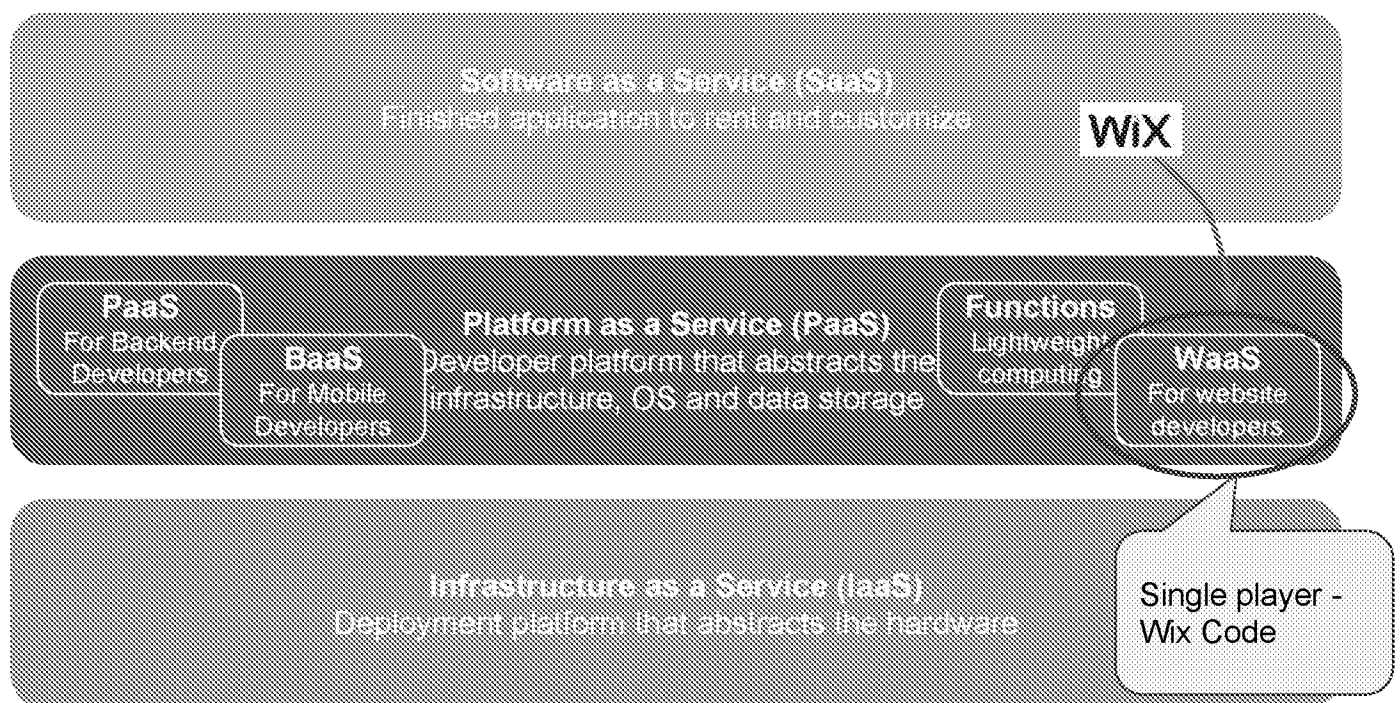
The Classic Cloud Landscape

Software as a Service (SaaS)
Finished application to rent and customize

Platform as a Service (PaaS)
Developer platform that abstracts the
infrastructure, OS and data storage

Infrastructure as a Service (IaaS)
Deployment platform that abstracts the hardware

The Classic Cloud Landscape



Appendix

A4

Hardcore Wix Code

Hackathon 2017

Session Goals

Demo of most flows

What's there?

Basic how to

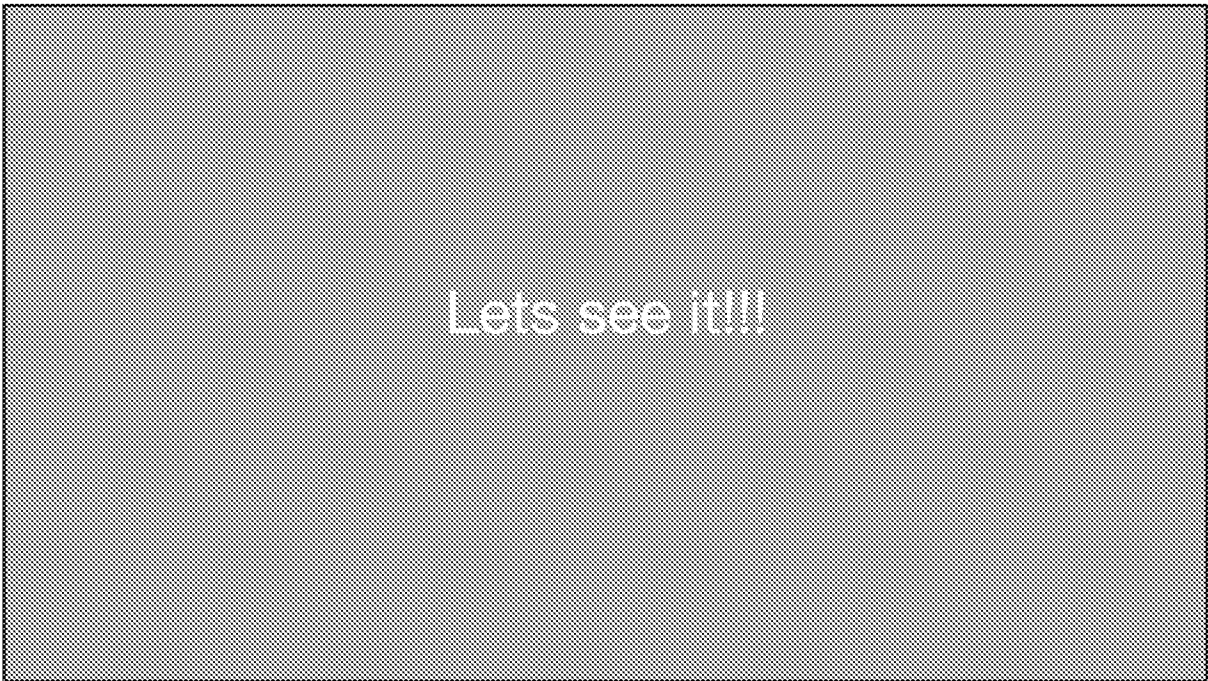
Provide links to videos,
documentation and samples

The hackathon

Q&A

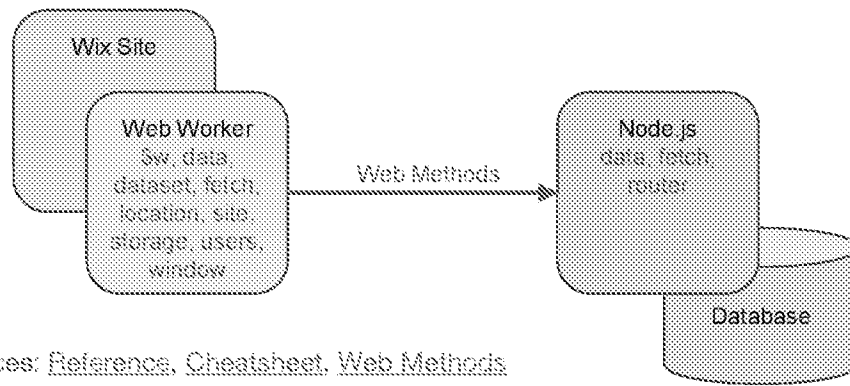


Be able to build any site
on the web with Wix



Lets see it!!!

Let's Code



Resources: [Reference](#), [Cheatsheet](#), [Web Methods](#)

Wix Database

- Content Manager
- Forms
- Display Data
- Editing Data

Resources:

- [Wix-data](#)
- [How to manage data](#)

The image displays three screenshots related to Wix database management:

- Top:** A screenshot of the Wix database table structure, showing columns like `id`, `name`, `email`, `password`, and `age`.
- Bottom Left:** A screenshot of the Wix database content manager, showing a list of data entries with columns for `id`, `name`, `email`, `password`, and `age`.
- Bottom Right:** A screenshot of a Wix form, showing input fields for `name`, `email`, `password`, and `age`.

Dynamic Pages

- One Design, Endless Pages
- Each page has a unique url, SEO friendly, Social friendly

	Title	Author	Category	Image
1	How to Cook	James P. ...	Recipe	
2	How to Cook	James P. ...	Recipe	
3	How to Cook	James P. ...	Recipe	
4	How to Cook	James P. ...	Recipe	
5	How to Cook	James P. ...	Recipe	
6	How to Cook	James P. ...	Recipe	
7	How to Cook	James P. ...	Recipe	
8	How to Cook	James P. ...	Recipe	
9	How to Cook	James P. ...	Recipe	
10	How to Cook	James P. ...	Recipe	
11	How to Cook	James P. ...	Recipe	
12	How to Cook	James P. ...	Recipe	
13	How to Cook	James P. ...	Recipe	
14	How to Cook	James P. ...	Recipe	
15	How to Cook	James P. ...	Recipe	
16	How to Cook	James P. ...	Recipe	
17	How to Cook	James P. ...	Recipe	
18	How to Cook	James P. ...	Recipe	
19	How to Cook	James P. ...	Recipe	
20	How to Cook	James P. ...	Recipe	

/recipes/panner-methi-pulau
 /recipes/cranberry-citrus-cocktail
 /recipes/gingerbread-french-toast
 /recipes/green-yuzu-baitall-cake-recipe

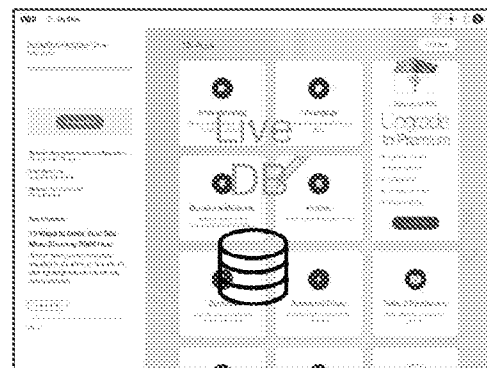


Sandbox & Live Databases

Editor



My Account



Permissions

Database

The screenshot shows the 'permissions' tab for a database table named 'customers'. On the left is a table structure preview. On the right, there are four rows of permissions for different roles:

Role	Permissions
Site Owner	Insert, Update, Remove, Read
Data Owner	Read
Site Manager	Read
Site Visitor	Read

Web Methods

The screenshot shows the 'permissions' tab for a web method named 'newCustomer'. On the left is a table structure preview. On the right, there is a table showing permissions for three roles:

Role	Permissions
Site Owner	Allowed to execute
Site Manager	Allowed to execute
Site Visitor	Allowed to execute

Wix Database APIs

```
var queryResults;
wixData.query("Customer")
  .gt("age", 20)
  .hasSome("labels", "Female")
  .eq("purchases.item", "ring")
  .skip(40)
  .limit(12)
  .find()
  .then(function(results) {
    console.log(results.items);
    queryResults=results;
  });
```

```
wixData.insert("Customer", newCustomer)
```

```
wixData.query("Customer")
  .gt("age", 20)
  .find()
  .then(function(results) {
    console.log(results.items);
  });
```

Resources: [wix-data](#), [working with the data API](#)

Customize my Code

- ◆ Dataset Events
- ◆ Wix Data hooks
- ◆ Dynamic Pages hooks
- ◆ Custom Router

PoCs

- ◆ Custom Database
- ◆ Vertical connectors
 - Frontend APIs via IFrame messaging
 - Backend APIs via Fetch
 - Backend APIs via Custom Database
- ◆ Web Callback

Custom Database

◆ Implement customDatabase hook

```
export function MyCollection_customDatabase(operation, params) {
  switch (operation) {
    case 'get': return doGet(params.itemId);
    case 'find': return doFind(params.filter, params.orderBy, params.skip, params.limit);
    case 'count': return doCount(params.filter);
    case 'insert': return doInsert(params.item);
    case 'update': return doUpdate(params.item);
    case 'remove': return doRemove(params.itemId);
  }
}
```

◆ Example -

<https://github.com/wix-private/cloud-runtime/blob/master/cloud-runtime/test/end-to-end/data-hooks-project/backend/data-custom-database.js>

Vertical IFrame Messaging APIs

Allows a TPA IFrame to open APIs for Page Code

Npm install the communication API

```
npm install santa-platform-super-apps --save
```

Export your APIs using the module

See at <https://github.com/wix-private/js-santa-platform-super-apps>

Install chrome extension

In the chrome extension -

- Select your app
- Select widget id

Web Callbacks

Allows web callback to a web module

The callback url:

```
http://wixcodewebhook.wixapps.net/v1/{user-name}/{site-name}/{web-module}/  
{function}
```

For example:

```
http://wixcodewebhook.wixapps.net/v1/13308aad-a1b0-48db-81d9-8d144e21ae19/wix-  
code/kfix-first-module/requestProxy
```

The function has to have the signature

```
function(request) : Promise[Response]
```

Resources

[Link to Marlowe's Videos](#)

[Link to the documentation center](#)

The [API site](#)

More examples

<https://docs.google.com/a/wix.com/document/d/1SqpfHViKslvnnNnZGOTr1bpqgpBJJ5FTHkaiOFV6Y4M/edit?usp=sharing>

Hackathon

Enable WixCode in your site

Add the following to the **end of the editor URL**:

```
&experiments=datepicker,dpages,sv_dpages,developerModeToggle,se__platform1  
,sv__platform1,se__grid,sv__grid,se__fileUploader  
&EditorSource=1.1339.0&ReactSource=1.2107.0
```

Getting support during the hackathon



Slack

[Join the official channel](#)

Can I do it? -> How can I do it?

Click to
provide your
feedback

FEEDBACK
?

Appendix

A5

01 – CGD

Compute grid

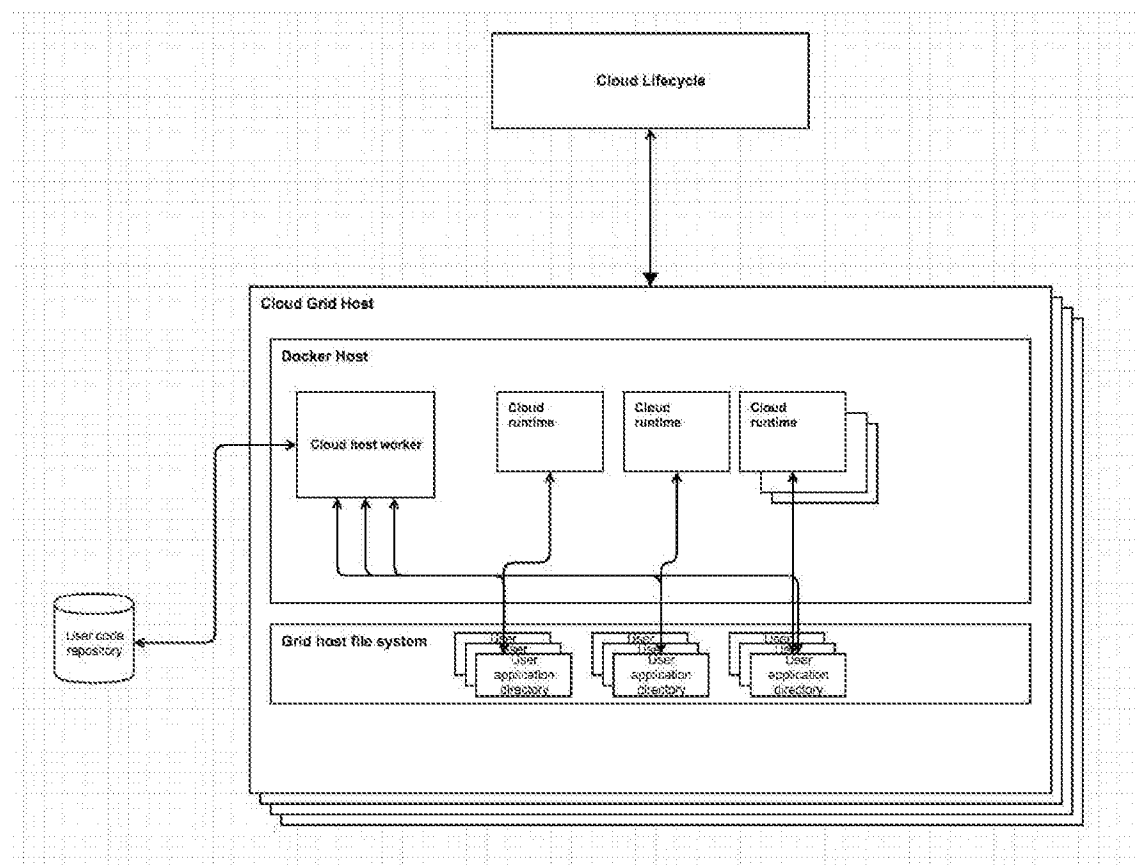
Wix code compute grid

Purpose and function

The grid has two main functions in wix code:

- When developing frontend and backend code, editor mode, it provides the underlying file system for the source code while providing front-end code bundling and backend code execution for preview purposes in a secure multi-tenant manner
- For public viewing, public mode, fast, secure multi-tenant reliable executors for user backend code.

Compute Grid architecture



Cloud Runtime FKA Elementory

Purpose

- Cloud runtime execution is the one single purpose of the compute grid existence
- Cloud runtime containers are designed to be reliable, fast and secure

Fast

- Cloud runtime container, are lightweight containers running a single node JS process.
- As it takes relatively long time to make a container ready to serve user code we maintain a pool of standby containers for that purpose
- It is the responsibility of cloud-lifecycle (see below) to make sure there are always enough standby containers to serve the on-demand capability

Reliable

- In public (non editing) mode
 - Containers are completely disposable
 - The system can provision a running container with user-code in less than 100ms
- In editing mode
 - User code is automatically archived periodically
 - The user can execute and preview her code immediately since code editing is done on the same file-system the cloud runtime container is running.

Secure

- A cloud runtime container only runs one application so no content leaks are possible
- Containers contain only a node JS binary, statically linked
- Root filesystem of the container is read-only
- Each container is executed under a unique user id
- User code is injected by Host Worker externally of the container
- A container can't access any code but the users whose code was assigned to it
- User code file system does not have execute permissions
- Disk quota, memory, cpu, network and other system resources quota limitations is enforced on the container

Cloud Lifecycle

- Although external to the grid host, cloud lifecycle it is worth mentioning in this context. Its function is to control and manage the wix code compute grid.
- Lifecycle uses two components on the grid host to manage the host and cloud-runtime containers
 - a. The docker daemon: To start, stop and control host-worker and cloud-runtime containers.
 - b. Host worker: To manage cloud-runtime container filesystem permissions, quota and archiving features.
- Relevant to this document is lifecycles function to allocate a unix UID per each user-code container cloud-runtime is using

Grid host

The compute grid is built of three main components which are managed by Wix-code lifecycle service. Each host on the grid has the following component:

1. Docker daemon
 2. Host worker (in a docker container)
 3. Cloud runtime fka "Elementory" (in a docker container)
- When a grid host starts running it registers itself to lifecycle by sending a message containing the grid host IP along with its defined resources: CPU and memory.
 - A grid host must have a working docker daemon installed and running for cloud lifecycle to manage it.

Host worker

Host worker is part of the components used by cloud-lifecycle to manage the host. It has two responsibilities:

1. Extend lifecycle ability to manage and manipulate system permissions, quotas, network, file system and code archive.
2. Provide file system like API for the editor while the user is editing her code in Wix editor.

Host worker lifecycle

1. Pre cloud-runtime container creation
 - a. Create directories for use code
 - b. Set appropriate permissions and disk quota
2. After cloud-runtime container is up "warm"
 - a. Set network traffic enforcement on the container
 - b. Periodically auto-archive user code while the user is editing her code
3. When cloud lifecycle decides to stop a container
 - a. Save and upload user code to the database (in editor mode)
4. Run periodic tasks scheduled by lifecycle
 - a. Located and kill "zombie" containers
 - b. Clean up disk after code has been archived

Host worker file system API

As mentioned above, the other responsibility of host worker is to provide virtual file system (VFS) API for the editor. Its responsibilities in that sense is to:

1. Provide basic file system manipulations such as:
 - a. Create, Updated, Read and Delete operations for file and directories
2. Make sure the above operations are made only on-behalf-of the permitted user while maintaining quota and permissions restrictions.

02 – ELM Elementary

Wix Code – Cloud Runtime / Elementary

Overview

Cloud Runtime (aka elementary) is a node.js based service which handles user requests both for edit, preview and view mode. The service is responsible for executing user code in multiple scenarios and for serving and transpiling user client code.

Main endpoints

- Web methods – specified in sub project #4 document.
- Dynamic routers – specified in sub project #8 document.
- Wix data - specified in sub project #3 document.
- Wix endpoints (compute functions) - specified in sub project #13 document.
- Static files – transpile, bundle and serve user code.

Internal/external modules separation

The service is split into internal and external modules, the user code is mounted into the external module and the user can access the wix libraries from there (wix-data, wix-routers, wix-endpoints, wix-fetch).

In order to limit the user to access the external modules only, the service uses scoped-require module (open source project developed by wix).

User code transpilation

Wix code allows the user to write code using the latest ECMAScript features (till es2017 for now), one of cloud runtime responsibilities is to transpile this code into a supported version of JavaScript by the runtime environment.

- Client side code – cloud runtime has an endpoint for requesting .js and .map files, the service is sharing the same core library with cloud-bundler-server for transpiling and bundling client side code.
- Backend code – the latest features of ECMAScript are not supported by the current LTS version of node(v6.11.0), the service uses babel-register in order to lazy evaluate the transpilation for required files.

User code execution

Via web methods, wix-data hooks, custom router, data-binding router hooks and compute functions, the user can write backend code which eventually needs to be executed by cloud runtime service.

Cloud runtime uses node domain in order to wrap the user code with context and it enables to capture the user code stack trace and logs.

03 – WDT

Wix Data

Wix Data

Wix Data API

Wix Data API - used to access the database. Wix Data API is used both in client side browser code (see CSWW) and in user backend code (see ELM). The API documentation can be found [here](#).

Wix Data API module also performs data transformation and error handling.

Wix Data API communicates with Wix Data Provider using Web Methods Protocol (see WBM).

Wix Data Provider

Wix Data Provider - is the implementation of the API. If user has no backend code of his own, the implementation runs and serves API requests in Cloud MultiTenant Server, otherwise it runs in user Docker container (see ELM). Wix Data Provider performs necessary Wix Data logic:

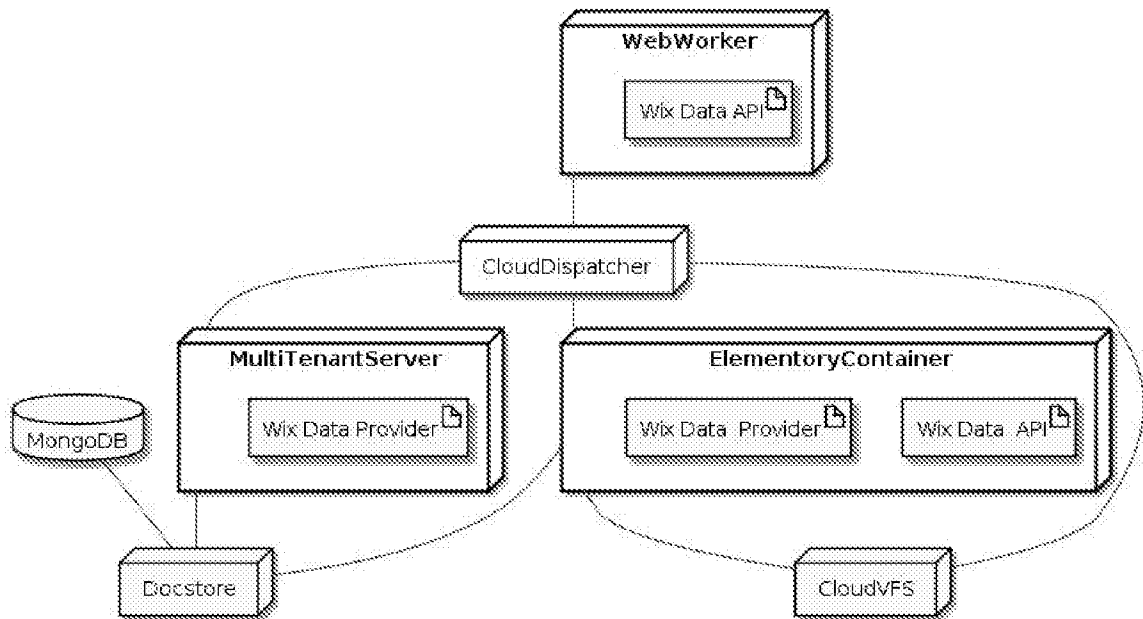
- Handles authorization. Permissions configuration defines what permissions does each role (site owner, data owner, anyone) have (read, insert, update, delete).
- Handles hooks. Hooks allow user code to intercept and modify Wix Data requests performed through API (see docs [here](#))
- Performs necessary query and data transformations to support advanced Wix Data features
 - Hides deleted fields from results and filters
 - Enriches resulting objects with computed fields (computed field is a special field defined in schema, that allows generation of new fields based on existing field data)
 - Joins referenced collection items
 - Enriches documents being saved with owner identification

Wix Data Provider is aware of the Schema and permissions stored in VFS.

Docstore

Docstore - service that acts as a bridge between MongoDB and Wix Data Provider. It

- Performs query translation from Wix Data to Mongo format
- Includes database copying and data syncing capability (allowing us to sync data from sandbox to live and from template to users site)
- Performs data enrichment by generating ids and storing document creation and update times



User libraries

To the best of our knowledge there is no direct use of any Open Source code in any of the projects (as in copied).

Open source libraries used in Wix Data API and Wix Data Provider:

Library	Link	License
bluebird	https://github.com/petkaantonov/bluebird	MIT
es6-promise	https://github.com/digitaldesignlabs/es6-promise	MIT
lodash	https://github.com/lodash/lodash	MIT
uuid	https://github.com/kelektiv/node-uuid	MIT
isomorphic-fetch	https://github.com/matthew-andrews/isomorphic-fetch	MIT

04 – WBM

Web Methods / RPC

WBM Web methods / RPC

Web methods are functions exported from ECMAScript modules that can be invoked over the network (such a module is called a web module). It is a mechanism to enable the code running in CSWW inside the end user's browser to seamlessly interact with code running in ELM. This mechanism handles authorization of which roles are permitted to invoke which web methods, injects request scoped information into web methods, propagates errors and logs to CSWW to be logged to the browser console. Additionally, it enables the same code to be invoked from user code in both CSWW and ELM using the same ECMAScript module import syntax. This approach allows for a "single image system" feel when debugging and developing applications.

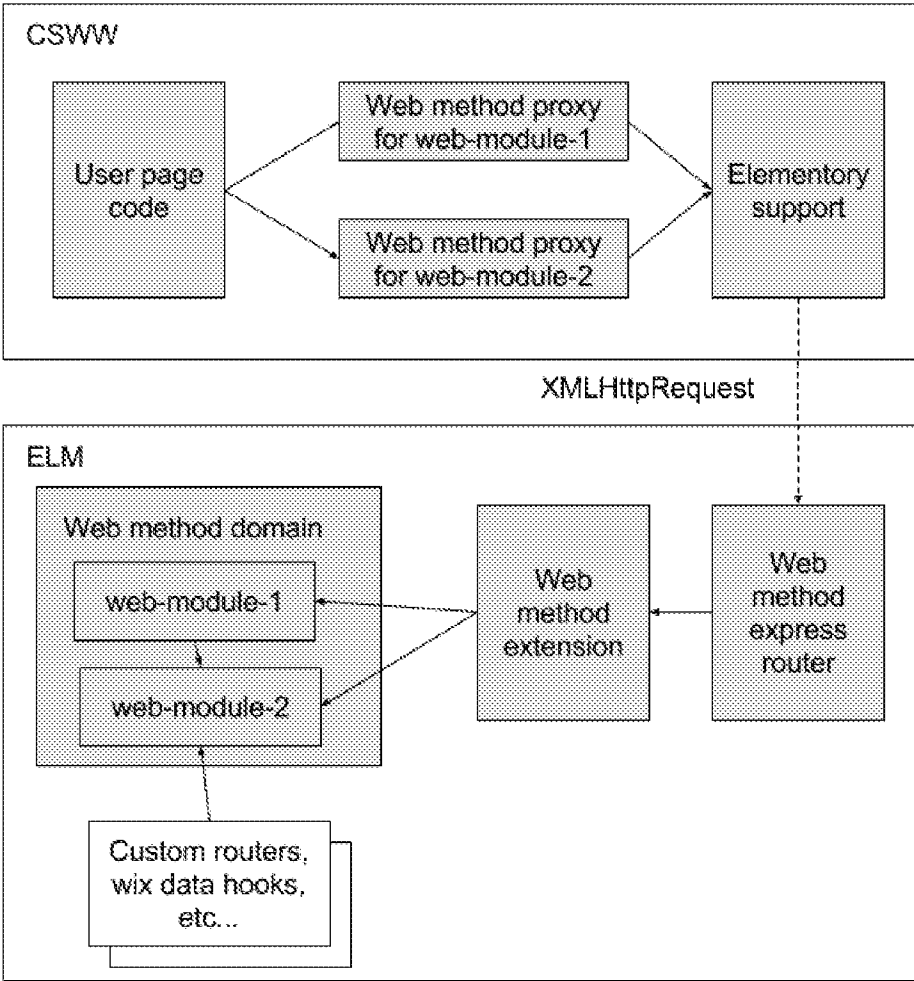
The server part of web methods runs inside ELM and can be freely invoked as regular functions that return promises from user code running in ELM, such as RTR Hooks and Custom Routers, WDT Hooks and CFN. WBM is also used as the RPC mechanism for WDT, to make it possible to be used in the same fashion in backend code and in page code. JavaScript proxies for invoking web methods as regular functions that return promises are bundled with the user code and run inside CSWW.

The mechanism is implemented by the following components:

- ❑ web-method-domain - attaches request scoped information to node.js domains, which is later retrieved by functions generated by web-module-extension and additionally overrides node.js default logging functions to collect logs to this domain.
- ❑ web-module-extension - uses Babel register to override the behavior of the global require function to transpile ECMAScript 2015 code to target the version of node used in ELM and hooks into the require mechanism to wrap functions exported from user web modules with.
- ❑ web-method-express-router - is the entry point to ELM for requests coming from elementary-support. It handles authorization of web method invocations from CSWW. It also forwards any logs collected during the execution of the web method once the web method execution has finished.
- ❑ elementary-support - provides the client side of the web method over the wire protocol. Runs in CSWW and is invoked by webmethod-proxies. Also outputs any logs received from elementary to the user's browser console.
- ❑ webmethod-proxies are generated by performing static analysis of web modules to determine which functions are exported. These are bundled with the user code to run inside CSWW.

The web method loading and execution is implemented using babel-register with babel-preset-es2015-node6. Scoped-require is used to alter the file system locations of modules that can be imported from user code. Webmethod-proxies are generated by performing static analysis using the acorn parser and bundling the generated proxies together with the user page code using browserify. The over the wire protocol used by elementary-support and web-method-express-router is based on JSON with extensions for handling native JavaScript dates.

Component diagram



05 – ZSI

Zero Setup IDE

IDE

The IDE component is an online code editor embedded in the Wix Editor. It can be used to edit code of existing site pages, or edit files that exist in the Wix Code file system (public and backend files). The IDE also provides rich contextual code completion, Wix media manager integration, code formatting and custom warnings and errors. All these custom features are built on top of an existing open-source IDE called Orion.

Code Completion

The IDE provides to the user rich code completion, both for the standard JavaScript environment and the Wix Code API.

Code completion can suggest completions for the following:

1. Wix Code component API (\$w) - provides suggestions depending on which elements exist on the current page, for example "\$w('#button1')". Meaning the user will see suggestions only for the elements that exist on the current page, and not for elements from other pages. These suggestions won't appear in regular public/backend files, only when editing page code.
2. Wix Code general APIs, such as "wix-data", "wix-window", which don't depend on the context of the code, meaning they are available when editing code for a page or in regular public/backend files.
3. JS objects and functions, such as "console.log()", also available in all file types.
4. Templates - provides template suggestions both for module imports (for example "import location from 'wix-location' ") and for JS common templates, such as for-loops.
5. Function parameters - provides the parameters and their names for functions

The code completion widget also provides a brief description for every selected suggestion, along with a link to the full documentation.

Media Manager Integration

A user can add code to a site that changes an image element source, meaning replacing the actual displayed image. She can set the source property of the image to any image URL, or to an image from the Wix media manager. When trying to set the source property, the code completion widget will suggest to open the media manager. There, the user can select an image, and its URL (custom Wix Code URL protocol) will be pasted to the code, for example:

```
"$w("#image1").src =
```

```
"image://v1/6e7958de786842eb811149ac9c25f73c.jpg/2362_4724/Cotton Candy Cone";"
```

The user can also hover over the URL, and a tooltip will open showing the image, and a link to open media manager to replace the image.

Technical Information

The IDE is built on top of the open-source project Orion [<https://wiki.eclipse.org/Orion>]. In order to provide our own custom experience and Wix specific bug fixes and behavior changes, we override specific modules from Orion with our own. Instead of modifying the original Orion code, we have a custom module loading configuration, that knows to load our overriding modules instead of the original ones, keeping our code separate from the original source code.

We don't use any other libraries or packages in the production code except for Orion and Orion's dependencies.

In order to build and test the code, we use a myriad of open-source packages, such as grunt, mocha, karma, selenium-webdriver, lodash and more. The code for these packages is used only in development time and is not shipped as part of the final product.

In order to provide code completion, we depend on the internal module "wix-code-reference-docs". This module depends on all the Wix Code libraries that expose API (such as \$w) and builds one big file that holds all the information needed to provide code completion. This file is copied during the build process and used in runtime by the IDE.

The entirety of code is written in JavaScript.

06 – EPT

Editor Platform

Editor Platform

The client side web-worker and SDK, enables custom code to run on the web application. The platform enables writing a code that is reusable, and therefore, can be distributed to multiple instances in the same web application and across web applications. The platform consists of the following components:

- **App Global Entity:** Shared state between its parts
- **Routers:** Controlling URLs and page resolution
- **Controllers:** Logic chunks of the app, Multiple instances
- **APIs:** Expose and mix everything

Application Global Entity

A global state for the application enables scenarios such as:

- Shopping cart, aggregating items during a multi-stage buying experience
- Multi-page wizard experience
- Multi-stage checkout experience (cart, shipping information, account information)
- Authoring experience that saves all information after finished

The platform allows the user-code to store application global state to enable the scenarios above. It also provide abstraction for the client web worker to implement page transition and cleanup without impacting the logic aspects of the application

Application Routers

The application can create routers (see section 8) and register it in the website. The application can store configuration information in the website, allowing the application developer to create a single server endpoint for the router and have multiple configurations per the distributed target. An example of a blog routing requirements can be:

- | | | |
|---|---|---|
| 1. domain.com/blog/2016/07/26/my-post | → | 1. Generic post page showing "my post" |
| 2. domain.com/blog/2016/07/26 | | 2. All posts from 26-Jul-2016 |
| 3. domain.com/blog/2016/07 | | 3. 10 first posts from Jul-2016 |
| 4. domain.com/blog/2016/07?page=2 | | 4. #11 - #20 posts from Jul-2016 |
| 5. domain.com/blog/tag/a-tag-i-use-a-lot | | 5. 10 first posts tagged "a-tag-i..." |
| 6. domain.com/blog/category/my-category | | 6. 10 first posts from category "my-cat..." |
| 7. domain.com/blog/2016/07/26/my-100th-post | | 7. Special post page designed for my 100th post |

By allowing the application to configure the router and save it per-instance/site, the router server code can be created once and support different URL formats (with/without dates) and specific pages for specific posts, although the URL structure is the same. Common usages are:

- Cascading page templates based on specificity (i.e. item → product → digital goods)
- Different URL formats based on SEO (Search Engine Optimization) requirements

- Addition of pages by adding content in a CMS (content management system)

Controllers

Controllers enable a creator of user code to package a piece of code with a list of components in a reusable way. In the normal (wix-code) method, the user accesses components based on their page-unique ID. This reduces the option to use it twice on the same page, and makes distribution hard. Controllers have the following properties:

- A **configuration**, save on the site. This configuration enables the code to be reusable as it provides a way to differentiate one instance of the controller from another. For example, if a controller code connects to external data source, fetches information, and presents it using an image and a text, the configuration would be the item ID in the remote system. This way, 2 separate controllers on the same page can represent 2 items in the remote system using a different image and text components
- **Connections** to components, using **role** (a string) as the name, with connection configuration. This enables the controller code to be written in advance, and then to be distributed, regardless of the IDs the components will receive on the page. This is especially useful for 2 or more controllers on the same page that access 2 different components, which can not have the same ID. In most cases, the controller will select (see selectors in section 15) the components to manipulate using the role. Each component can have multiple roles defined, but multiple controllers, to enable complex data binding scenarios and behavioral programming models (i.e. each controller is responsible for a different aspect of the component, for example a drop-down component can get the list of options from one controller and save the selected one through another)

APIs

Each controller can expose its own APIs. This makes the controller appear as any component, and has similar behavior. It enables other code-creators, on a site level or of other controllers reuse the controller code as a black box, turning the controller to an extension block of the runtime environment

07 – CSWW

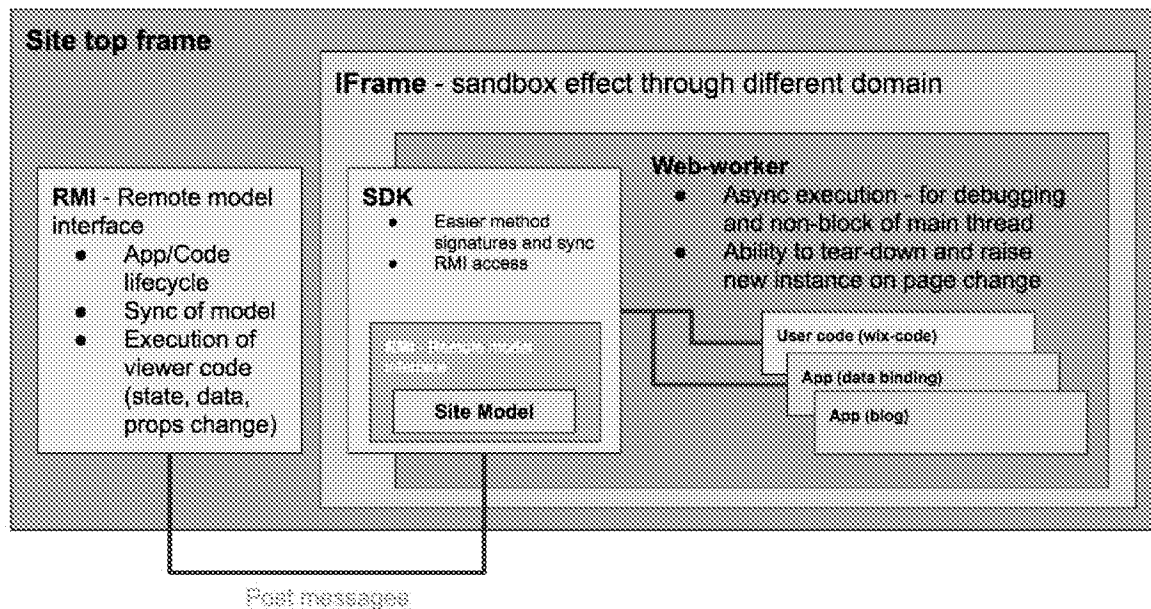
Client Side Web Worker

Client side web worker

The client-side web worker enables extending a website with custom functionality. The most common usages of that extension are:

- Dynamic content enrichment of the site
- Custom user flows
- Custom interactions

In order to enable this, while maintaining SLA, security, and the ability to host these extensions in a unified cloud, the following architecture is used:



Below is a list of logical components of the system:

- **Site top frame:** This is the main page of the web application. It is served on a user's domain or the general domain of the hosting service (Wix). This is the main engine that runs the web application, AKA "the viewer" or "the runtime environment"
- **IFrame sandbox:** In order to preserve the security of the top frame from malicious code (cookie hijacking, etc.), an IFrame with a different domain is erected. The domain difference is sandboxing system resources (window, cookies, etc.) and memory (all communication can be done using postMessage only and not by memory sharing)
- **Web-worker:** A web worker is a standard browser capability that allows running asynchronous code. The web worker creates another layer of protection to the top frame, as all communication is possible via postMessage, and the API surface globally available in the web-worker is much limited than the top frame. For example,

window global object is not available, DOM (document object model) access is forbidden, meaning that web-worker code cannot manipulate UI elements. In addition to the security benefit of the web-worker, it's async nature also ensures that the top-frame will not stop responding even when the web-worker is CPU intensive. Another benefit is the ability to debug and pause on user-code (code running in web-worker) without blocking the top frame execution context, and allowing the user to see propagation of the user-code in the UI layer, even while in breakpoint

- **RMI (Remote Model Interface):** This component is responsible to create and maintain a logical model of the top frame, and share it with the web-worker, via postMessage transport layer. Changes in the model will have effect on the web application. For example, a button's text and link are represented in the model. Changes of these values in the model will affect the visual button component, and its functionality. In addition, the RMI layer enables calling predefined methods in the top frame by the web-worker. This is used to enrich the functionality of the web-worker, and perform actions that are not related to the state, as represented by the model (i.e. animation). The remote model bridges the async barrier, allowing the user-code the perception of synchronous code, which is simpler to write, although the actions are async. For example, user-code changes an image URL from A to B. The next line of code reads the value to determine the next actions. The model already have the value changed from A to B, so it returns B, which is the consistent value, although the async behavior of the model does not guarantee that it is already presented in the UI of the web application
- **SDK:** The SDK is a wrapping layer on top of the RMI model. It is used to expose a simpler API to the user-code, simplifying the model into a consistent and documented method calls and property setters and getters. It exposes lifecycle methods and the \$w object as the base context for the user-code. Detailed description is provided in section 15

08A – RTR Router

RTR Router

Routers enable the site owner to define a custom URL structure for their site. Routers are used by dynamic pages to resolve a specific URL to a page and by SEO and the editor to get all the pages available in the site via the sitemap request. The routers also expose additional SEO information to be associated with a page. There are two types of routers: data binding and custom routers. The routers are backend components running inside ELM.

Data binding routers automatically connect a URL structure based on one or more database fields and custom text fragments provided by the site owner. This can be done completely through the UI in the editor and requires no user code. Additionally, these routers can be customized by hooks, allowing the user's code to modify the parameters passed to the router, modify the response and modify the query issued to WDT.

Custom routers give full control of the URL structure to the site owner. From the editor, the site owner can create a router for a specific prefix, where the rest of the URL will be interpreted by his custom router. The router requests are handled by two functions provided by the user: router and sitemap. The router function handles resolving single URL's to specific pages. The wix-routers API describes the data that is passed to this request. This function is expected to return one of the predefined responses in the API (e.g. OK, not found, forbidden, etc) along with any additional SEO information. The sitemap function is similar, except that it's expected to list all the pages available under the route.

The dynamic routing is achieved by two collaborators: one running in the user's browser and one in the backend. When a page is loaded via a URL that is handled by dynamic router, the default HTML page is loaded in the browser and an AJAX request is made to the backend to determine what page should be displayed. The backend part of the code is not aware of what prefixes are handled by what routers. All the data necessary to resolve a URL, including the pages the router can choose to display, is sent in this request and the response from the backend is used to open the appropriate dynamic page for the page and populate it with data. The router can choose to populate the data that will be exposed to the page when it will be opened, so that another round trip to the server can be avoided.

08B – RTR

Router – detailed

- Dynamic Routing
- Data-Binding Router
- Custom Routing

Introduction

Dynamic Routing Model

Basic components of the dynamic routing model

A more detailed explanation of routing

Router endpoints

The Resolve-URL endpoint

Resolve URL endpoint inputs

Resolve URL endpoint outputs

Page Roles in depth

Sitemap Endpoints

Implementing sitemaps correctly

Accounting for user credentials

Accounting for URLs that result in a 404 response code

Accounting for routes

Accounting for "no-index" flags on a dynamic page

Sitemap endpoint inputs

Sitemap endpoint outputs

Side note: the Preview Picker

Side note: link creation

Dynamic routing and SEO meta-tags

HTTP Cookies and Headers

404 and 403 pages

Cross domain issues when implementing a dynamic router

Complete example

The Data-Binding Router

Brief recap of data-binding concepts

The data-binding router

Transforming a URL into a wix-data query

Returning data to the client

Using a dataset to bind fetched data to UI elements

Generating links from URL patterns

Schema extensions

Ephemeral computed fields

Reference fields

Other considerations for data-binding router

Page roles in the data-binding router

Specifying SEO meta tags

The site-map endpoint

Hooks on the data-binding router

Open issues

The Custom Router

Introduction

This document outlines the architecture and different subsystem behind the **Dynamic Routing** capabilities that we plan to add to the editor/viewer stack. The document then

outlines the new **Data Binding Router**, and the way it is built on top of the dynamic routing model. Finally, the document discusses how developers can build **Custom Routers**, to create their own dynamic routing implementations.

Dynamic Routing Model

Dynamic Routing enables a wix site/app to define and manage **Dynamic URLs**. This allows users and editor apps to create sites that have more than just static pages with URLs that are known at publish time.

Ultimately users and apps can:

- Define URL patterns that map to concrete rendered pages
- Create **Dynamic Pages** whose content depends on the incoming dynamic URLs
- Expose these URLs to bots (search engines and similar automated consumers)

Basic components of the dynamic routing model

The basic components of this model are **Apps, Routers, Routes, Router endpoints, URLs, Dynamic Pages, Page Roles** and **Sitemaps**. Below we explore them and the way they interact to create the dynamic routing model.

The basic construct that allows for dynamic routing is the **Router**. **A Router is owned by an App**. Each app may have more than one router if it so needs.

A router is responsible for resolving a specific incoming URL to a specific dynamic page that the viewer should render.

Incoming **Dynamic URLs** have the following structure: **<site>/<prefix>/<suffix>**:

- **<site>**: has the form **<my.site.com>** for premium sites, or the form **<user_name.wixsite.com/site_name>** for free sites.
- **<prefix>**: this part is what determines which router will handle the incoming URL. Each router registers a set of **unique prefixes** in the site header. This information is managed by the editor runtime. **The editor ensures no two routers define the same prefix.**
- **<suffix>**: whatever comes after the prefix. This part, together with the prefix, is later used by the router to resolve the incoming URL.

Each router supports one or more **Routes**. A route is a template for the **<suffix>** part of a URL. Examples of routes (the routes are in bold):

- my.site.com/blog/
- my.site.com/blog/post-**{id}**
- my.site.com/blog/{year}/{**month**}/
- my.site.com/blog/featured-**posts**

An app may always support all of its possible routes, or it may let the user select which ones are enabled and which ones are not. For example, in a blog app, it may be up to the user to decide if he wants to use the featured-posts capability or not.

Note that it is up to the app and the specific router to make sure it can tell the difference between different routes. Consider the following two routes for a blog app:

- my.site.com/blog/{year}/

- my.site.com/blog/{author}/

It may be hard to tell which route to use to decode the incoming URL - there might be an author named "2016"... in such a case it may be better to create these routes instead:

- my.site.com/blog/{year}/
- my.site.com/blog/authors/{author}/

A **Dynamic Page** is simply any page that a router can route to. **Dynamic pages are owned by routers** - i.e. every router has a set of dynamic pages that it and only it can route to. Hence a router cannot route to a "legacy" static page.

Each dynamic page is assigned one or more **Page Roles**. Page roles help the router decide how to resolve an incoming URL into a concrete page to render.

The (simplified) algorithm works as follows:

1. [viewer runtime] Incoming URL is parsed to decide which Router will resolve it, based on the <prefix> part
2. [viewer runtime] calls the chosen router's resolve_url endpoint, passing in the info the router needs in order to resolve the URL to a specific dynamic page ID.
3. [chosen router] Computes an **ordered list** of Page Roles for the incoming URL
4. [chosen router] Selects the dynamic page ID to resolve to, based on the list of dynamic pages and their associated page role. For each page role in the ordered list, the router looks for a dynamic page that matches that role. The search is terminated when a dynamic page that has that role is found. If no page is found, the URL resolves to a 404 response code.
5. [chosen router] Returns the selected page ID to the viewer, alongside auxiliary information needed to render a dynamic page (most importantly, an optional data payload)
6. [viewer] Renders the selected dynamic page

Note that it is perfectly OK for the router to resolve different incoming routes to the same page ID. For example, consider a Zoo management application. This application registers a Router under the <zoo> prefix, and supports two routes:

- /zoo/animal_of_the_day
- /zoo/animals/{animal_name}

In this case, both routes will resolve to the page role {animal_page}. If the incoming URL is </zoo/animals/tiger>, and if tiger is currently the animal of the day, both routes will resolve to the dynamic page that renders an animal, and both will render a beautiful tiger on screen.

A more detailed explanation of routing

As we've seen, the viewer uses a router to resolve the page to render. However, in order to correctly render a dynamic page, there are a few more considerations to take into account.

See a diagram that explains this in detail [here](#).

Router endpoints

Each router must support the following two endpoints:

- **Resolve URL endpoint:** used to resolve an incoming URL to a page ID

- **Retrieve Site-map endpoint:** used to retrieve the complete list of URLs this router can resolve

These endpoints and their behavior are detailed below.

The Resolve-URL endpoint

This endpoint is used when an incoming URL needs to be resolved to a specific dynamic page.

Resolve URL endpoint inputs

In order to resolve an incoming URL, the router potentially needs the following pieces of information:

- **URL parts (site, prefix, suffix).** Note that this should include the **URL query string** (the part after the '?' and before the '#' characters).
- **App signed instance:** The router endpoint needs to know the application instance ID so it knows which site the incoming request is from. For example consider a Blog application. In order to resolve the incoming URL for a specific blog post the router has to know the the blog app instance.
- **User credentials:** Part of the signed instance. The router needs to know the user credentials of the user currently issuing the request. This is needed, for example, to know if that user is allowed to view the information in the specific page requested. For example, a Blog router may want to forbid anonymous users from seeing the page behind *my.site.com/blog/admin*.
- **Routes array:** [This is part of the router configuration]. An array of the routes this app instance supports. Usually, each route will provide information about its URL pattern, about the Title and other SEO meta-tags for this specific route, etc. See more in the SEO section below.
- **Router configuration:** Extra information that is app and router specific. For example, consider a Blog App. Assume this app provides a dynamic page with all the posts for a given author. The URL for such a page might look like this: *my.site.com/{{author_name}}*. In this case, there's a router that handles these URLs, and it needs to be able to map the author name to a specific author ID in its posts database. In order to do that mapping, the router configuration will hold the following: {author_ID:123}. When the router endpoint is invoked, the posts for author_ID 123 will be retrieved.
- **List of dynamic pages and their associated roles:** Used by the router to select the page ID to render based on role matching like explained above.

Resolve URL endpoint outputs

- **Page ID to render:** The primary result of the router's resolving of a URL.
- **HTTP response codes:** Since we're dealing with URLs, we need to account for the possible HTTP response codes. Specifically response codes include: {200, 301, 302, 403, 404}. The viewer runtime uses these codes to behave correctly.
- **Data Payload:** The router can potentially return a data payload that is later used to render the specific dynamic page. This payload can for example be fed into a dataset that is used to bind to components on the page.
- **Title:** The title for the page that will be rendered. The viewer runtime uses this to set the page's title for the browser. The title can be computed based on the content retrieved for the specific dynamic page. **Typically, different routes may result in**

different titles. For example, the route `</zoo/animal_of_the_day/>` may resolve to the dynamic page for showing an animal, with the title “*Our animal of the day - the Tiger!*”. The route `</zoo/animals/tiger/>` may resolve to the same dynamic page, but the title will be just “*Our Tiger*”. Hence, the template for creating the title is typically part of the router configuration for that specific route.

- **SEO meta-tags:** SEO related information, more on this later.

Page Roles in depth

As explained above the system uses page roles to determine which actual dynamic page to resolve to. Page roles allow apps to create **Fallbacks**. By this we mean that an app's user can create specific dynamic pages for specific content, that may differ from the standard dynamic page used for other pieces of content.

For example, in a blog app, there may be two different dynamic pages:

- Standard post page, with the role `<post-page>`.
- Specific post page, with the role `<post-page-id-123>`.

The router resolves an incoming URL of the form `<site>/<blog>/<post-id-xxx>`. It will compute the following page roles ordered list: `{post-page-id-xxx, post-page}`.

For all incoming URLs that have an id different than 123, the router will resolve to the standard post page. **However, for the `<site>/<blog>/<post-id-123>` URL, the router will resolve to the specific page for that 123 post.**

Through this mechanism we have created a **fallback strategy** for resolving dynamic pages. If the user later on decides to add a specific page for post id 321, all he needs to do is add a new dynamic page with the right page role, and the router will know to resolve to that page.

Sitemap Endpoints

A **Sitemap** is a list of the URLs exposed by a given site. In the context of the dynamic model, this list contains the list of URLs, and for each one added information that the consumers of the site map need.

Site maps serve two different scenarios:

- Firstly they are used for the classic intent of exposing the list of pages in a site to a bot request. In this scenario, the router sitemap endpoint computes the full list of URLs it can resolve, and returns them to the incoming bot request.
- Secondly, sitemaps can also be used in the editor. In this scenario the editor issues a request to the sitemap endpoint, and uses the returned results to let the end user select one of the returned URLs. This is useful in two distinct cases:
 - **When the user creates a link to a dynamic page.** In order for the link to be created, the editor needs the full URL to link to.
 - **When the user wants to preview a dynamic page.** In this case, we need to know the exact URL that resolves to this page so we know how to provide the page with the data it needs in order to render correctly.

Implementing sitemaps correctly

The following aspects must be taken into account.

Accounting for user credentials

The list of URLs must take into account the user credentials for the call. For a bot (anonymous) request, the sitemap should only contain URLs that are publicly available. A bot should not see URLs that are forbidden to an anonymous user. For example, assume we have an app that has a router with prefix `<my_app>`. If that app has pages under `<site>/<my_app>/<admin/salaries>` to manage the salaries of employees, this URL should not be returned to the caller.

Accounting for URLs that result in a 404 response code

It is possible that a router can resolve certain URLs, but that the site has not defined a dynamic page with the right page role to render that URL. In these cases, the router would resolve to a 404 response code. It is the router's sitemap endpoint responsibility to remove such entries from the results it returns to a bot request.

Accounting for routes

A router may potentially support routes that are not used by a specific site. For example consider a blog router that supports routes such as `</posts/post-id>` and also `</posts/featured-posts>`. However, in a blog instance in a specific site, the user might have decided not to use the featured-posts capability, and hence didn't mark any posts as featured and didn't even create a dynamic page for featured posts.

In such a case, the sitemap shouldn't return this URL, otherwise a bot might try to look for it and will get a 404 response, or worse, get another dynamic page with an empty list of posts.

The way to avoid these problems is for the sitemap generation to take into account the routes supported by the app instance in a site, and only generate URLs for supported routes.

Accounting for "no-index" flags on a dynamic page

A user might mark a dynamic page as "no-index". This means that it should not be indexed by a search bot. When the sitemap endpoint computes the list of URLs, it should also compute the page ID each URL resolves to. If that page ID is marked as "no-index", that URL should be removed from the list.

Sitemap endpoint inputs

In order to compute the sitemap, the endpoint relies on the following inputs:

- **Signed instance:** Used by the endpoint to know which app instance's sitemap to generate
- **User credentials:** Encoded in the signed instance, so the router can generate the sitemap that contains the URLs that user is allowed to see
- **"Is_editor" flag:** So the endpoint can generate a sitemap that is suitable for the editor's use. The editor needs information to display the sitemap to the end user, which the bot doesn't require.
- **Router configuration:** Information about the way this specific router generates sitemaps.
- **Routes in use in this site:** (as part of router configuration) List of supported routes. The sitemap generated should only include routes that are in use in the specific instance of this app in this specific site.
- **List of this router's dynamic pages and their page roles:** Used to make sure the sitemap only includes URLs that actually map to a dynamic page. Any URL that resolves to no page should be excluded from the list.

- ***"No-index" flag per dynamic page for this router (part of route configuration):*** URLs that resolve to a page that is marked with "no-index" should not appear in the sitemap generated for a bot. However, the same URLs should be included if the request comes in from the editor, since in that case it does not matter if the resolved page is indexed or not.

Sitemap endpoint outputs

For bot requests, the following should be output:

- ***List of URLs*** for the bot to scan
- Per URL, the following (all optional) parameters:
 - ***Last modified date***
 - ***Change frequency***
 - ***Priority to index***

Note that a single sitemap request should not return more than 50,000 URLs. At this stage we do not plan to support paging of results - if you have more than 50,000 URLs, omit the extra ones.

For a complete specification of the sitemap protocol for bots, see [here](#).

For Editor requests, the following should be output:

- ***List of URLs***
- per URL, the following parameters:
 - ***Friendly title***. Will be used when creating links in the editor to a specific dynamic page, and by the preview-picker box in the editor.
 - ***Page ID this URL resolves to***. Will be used by the Editor's link creation dialog to let the user select a URL that resolves to a specific dynamic page.

Side note: the Preview Picker

When the user wants to preview a dynamic page, the editor must know which URL suffix to use for this dynamic page. The preview picker element invokes the sitemap, and filters the results for URLs that resolve to this dynamic page ID. The user then browses the list, selects the URL by its friendly title. The editor now knows the right URL suffix to use in order to render the dynamic page.

Side note: link creation

Assume the user wants to create a link to a dynamic page. The editor retrieves the sitemap, and looks for URLs that map to this page ID. The user browses the URLs by looking at their friendly title, and selects one. The editor now creates a "router link" to that specific page ID, using the selected URL's suffix.

Dynamic routing and SEO meta-tags

A router should provide meta-tags as part of its resolve-URL endpoint. The main ones include: "title", "no-index", "description" and "image". It is up to the router to provide them per resolved URL, in whatever way makes sense for the app and that router. I.e. the viewer/editor infrastructure is agnostic to meta-tags. Each app can supply an iframe to define its meta-tags behavior. This iframe is rendered as part of the page settings dialog for a dynamic page. The information can then be persisted as part of the router's configuration and be used later to generate the correct response.

HTTP Cookies and Headers

Cookies are not sent to the various endpoints. As for headers, the final list of headers that are important to support is still TBD.

404 and 403 pages

These pages have built-in support as part of the viewer/editor infra. The user can create such special pages, and if they exist, the infra will know to route to them in case the appropriate HTTP response code is received.

Cross domain issues when implementing a dynamic router

Still TBD

Complete example

Still TBD

The Data-Binding Router

In this part of the document, we explain the **Data Binding Router** and the way it is built on top of the dynamic routing infra detailed in the first part of this document.

Brief recap of data-binding concepts

We assume the reader knows the basics of the data-binding model.

Still, a short summary of data-binding components follows:

- **Collection**: a mongo-DB based set of records.
- **Schema**: the meta-model of data in a collection. A collection's schema is persisted in a schema file (one per collection).
- **Content manager**: a spreadsheet-like application through which the user can manage the content of collections
- **Wix-data**: a library that provides data access, as part of wix-code. Wix-data has a client interface as well as a backend layer that's responsible for executing user-provided hooks, enforcing permissions, performing validations, etc.
- **Dataset**: an editor/viewer element that enables users to bind UI elements to data provided through the wix-data APIs.
- **Data-binding**: the ability to bind a visual component to data through the use of a dataset.
- **Dev DB**: the database instance the editor works against.
- **Prod DB**: the database instance a live site works against.

The data-binding router

In order to complete the offering of our data-binding system, we wish to provide a router that bridges the gap between a dynamic URL and the data needed to render a dynamic page.

Essentially, the data-binding router is responsible for parsing an incoming URL, and transforming it into a query using wix-data. Once results are obtained, they are fed into a special type of dataset, that in turn provides data to the element on the rendered page.

Let's review the different elements that make up this system: creating the right query, returning data to the client, and using a special dataset to bind the data to components on the page.

Transforming a URL into a wix-data query

Assume we have an **animals** collection with animal record. Also assume we want our animal page to have the route: `/animals/{{animal_name}}`. How can our router generate the correct wix-data query?

The router needs to perform the following:

- Know which collection to query $\Rightarrow$ we'll keep the collection name as part of the router's configuration when defining the routes.
- Know which field(s) to query on $\Rightarrow$ we'll maintain a mapping in the router's configuration. In this case, we map `{{animal_name}}` to the "name" field

- Generate the right query syntax ⇒ this is a bit more tricky. The value in the URL is always in lowercase, and with no spaces. However the value in the collection is usually in a more human-readable form. For example, the animal name might be "Grey Wolf" in the collection, but "grey-wolf" in the URL. To overcome this, the router needs to generate a "regex-query" - searching the collection for all possible permutations of "grey-wolf": {"Grey Wolf", "Grey-Wolf", "grey wolf", etc.}

Returning data to the client

This is rather straightforward. The router provides its payload per the `resolve_url()` API and that's it.

Using a dataset to bind fetched data to UI elements

Now that the data is available to use on the client side, we need a dataset to push in into the UI components on the page. This dataset has a few special characteristics, hence we'll call it a **Router Dataset**. This dataset has the following special behaviors:

- At runtime, it expects a data payload and initializes itself with it instead of issuing a wix-data query by itself.
- At runtime, when it needs to fetch more data (e.g. when a grid is scrolled, etc.), it needs access to the filter defined by the router for this URL. Hence, it knows how to generate a wix-data query that will return similar results as the query created by the router.
- At edit time, it does not allow the user to override its filters on the parts that are bound to the URL elements. Note that the user is allowed to add more filters and to specify sorting and page size, like for a normal dataset.
- At edit time, it does not allow the user to move it to be on the master page (as it is bound to the URL, which implies a specific page).
- Other UX-related behaviors (e.g. special notification if it's going to be deleted, or special behaviors when it's duplicated).

Generating links from URL patterns

Let's now turn our attention to the generation of links. Assume that we want a very simple index page, that shows all animals in a grid, with a link to each animal's unique page. How do we generate these links, to be used in the grid?

The basic idea is to use the well known URL patterns, or routes, defined for dynamic pages for a specific collection. Once these patterns are known, we can generate links from them.

For example, let's say we have three different URL patterns in our system:

- **[main route]** `\zoo\animals` ⇒ collection page of all animals in the zoo
- **[habitat route]** `\zoo\animals/{{habitat}}` ⇒ shows animals that live in the {{habitat}}
- **[animal route]** `/zoo/animals/{{animal_name}}` ⇒ specific animal's page

Once we've defined these patterns, given a record in the animal collection, we can create all three of these URLs. The way to make this work involves a few new concepts in our system.

Schema extensions

A **schema extension** is a segment in a collection's schema file that is provided by an application. This extension typically adds schema elements to an existing schema file.

In our case, once a new route is added (or changed, or removed) for a given collection, the system modifies the schema extension provided by the data-binding app for the relevant collection. For each route, we create an **Ephemeral computed field** (see below) that tells the system how to generate a URL for the route.

Ephemeral computed fields

These fields are (a) computed and (b) ephemeral. This means they are never persisted in the actual record in the collection; rather they are computed on data retrieval per the computation specified. **These computations all take place at the wix-data backend layer.**

In our case, for the example routes given above, we create three such fields:

- Main_URL: string. Val = "/animals/"
- Habitat_URL: string. Val = "/animals/" + to_url_part(field_val["habitat"])
- Animal_URL: string. Val = "/animals/" + to_url_part(field_val["name"])

Note that the computation above doesn't specify the <site> part of the URL, since it is unknown to the wix-data engine that performs the calculations. **It is up to the router to complete the computation and add the missing <site> part to create full URLs.**

Reference fields

In future versions, we'll add reference fields from one collection to other collections. The existence of such fields will enable the creation of URLs that link from an item in collection A to items in collection B. i.e., if we have in collection A a reference field to collection B, in which we have an ID of an item in collection B, we can use that ID to generate the right URL links based on the ephemeral link fields defined on collection B.

However, not for the current version.

Other considerations for data-binding router

Page roles in the data-binding router

Note that at least in v1 of this system, we do not plan to use multiple roles per dynamic page, or to let the user specify custom roles for a page. This means that users cannot create "fallback pages" or more specific pages per the generic capabilities of page roles.

Specifying SEO meta tags

SEO metatags are managed through an iframe provided by the data-binding app to the page settings dialog infra provided by the editor. Through that iframe, the user can specify how to generate SEO metatags per dynamic page owned by the data-binding router.

This configuration is persisted as part of the router's configuration, and passed along to the router endpoints to generate the right meta-tags based on data retrieved from the database.

For example, the user can specify the following expression for the title of the "animal" page: **"meet our magnificent {{name}}!"**. The router will then generate the right title when retrieving data.

The site-map endpoint

Implementing the sitemap is relatively straightforward:

- Per collection, retrieve all its items.
- Since we have computed fields per URL route, we already have the desired information.
- All that is left is to get the **distinct set** of computed URLs, and return it to the caller.

Hooks on the data-binding router

We plan to support the ability to let user code run before and after the router performs its functionality. The basic mechanism is similar to backend hooks we have in wix-data today - the user can register a function that will be called before or after the router executes.

The before hook gets the same input parameters as the router itself, and may or may not pass them to the actual router implementation.

The after hook gets the output parameters from the router's execution, and may alter them before they are sent back as part of the HTTP response.

Open issues

- dev/prod db
- Permissions on page level

The Custom Router

- Creating a custom router
- Creating custom dynamic pages
- API for a custom router, including errors
- Using page roles with a custom router
- Using data binding with a custom router

09 – DYP

Dynamic Pages

Dynamic Pages

General description

When you create a dynamic page, you're designing one page layout that can be used over and over again, each time displaying different content from your database collection.

As opposed to a regular page, which has only one url that may lead to it, a dynamic page can have different URLs. The content that is shown on the page depends on the URL that was used to access that page.

Since a dynamic page displays different content each time it's viewed, it's not like a regular page. A regular page stores its content in the page itself, while the content for a dynamic page is stored in a database collection.

For example:

Let's say you have a recipes site and you would like to design a page that displays data for a single recipe. With regular static pages, you would have to create a separate page for each one of your recipes. With dynamic pages, you will only create one dynamic page, and assign it a URL pattern that can determine the specific recipe that should be displayed at runtime.

If each of your recipes (stored in a WixData collection) includes a field named "title", you may define the dynamic page's url to be "www.../recipes/{title}".

When a user browses to "www.../recipes/pizza", the page will display the content for recipe titled "pizza", etc...

Main components / Related WixCode elements

Content Manager (& WixData database)

Used by the user to create collections & manage their data

Data-binding editor application

Enriches the Wix editor with panels that allow a user to create and manage dynamic pages and to connect page components to the content parts relevant to them.

Data-binding router

A backend application that accepts 2 types of requests:

1. Routing request - when loading a dynamic page that belongs to the data-binding application, the wix viewer requests the router for page and content to display for a given input URL. The router will try to match the given URL with patterns defined by the site's owner and will look for relevant content in the database. If found, the router responds with the relevant page identifier and content.

2. Sitemap request - when a bot requests a Wix site's sitemap, and the site includes dynamic pages that belong to the data-binding application, the router will be requested to return a list of all possible URLs that it can resolve.

Data-binding viewer application

Executes when a dynamic page is loaded. Receives the content returned by the router and binds it to the components on the page according to their configurations.

Technical information

The entirety of code is written in JavaScript.

We use different open-source libraries that are shipped with the code, such as react, redux, lodash, co and more...

Other open-source libraries are used to build and test the code but are not shipped with it: webpack, babel, mocha, eslint, and more...

Third-party services

Sentry

The code uses a third-party service called Sentry (<https://sentry.io>) to log problematic flows that happen at runtime (both editor and viewer time).

Each time some error happens, the code sends information about the error and the current context to sentry. This information includes things like:

- Application version
- Editor configuration
- Description of the error
- Site / editor URL
- Referrer URL
- User-agent (browser type and version...)
- Operating system
- User's database content
- Code written by the user in the IDE
- User actions that lead to the error

10 – DBG

Data bindings

Dataset & Data binding

The core element of the data-binding system is the **Dataset**. A dataset is a client side provider of data, which visual constructs (i.e. editor components) can two-way-bind to. It is also the channel through which the client application manages data, by implementing a data-retrieval and data-mutation protocol with the backend logic layer and eventually with the datastore itself.

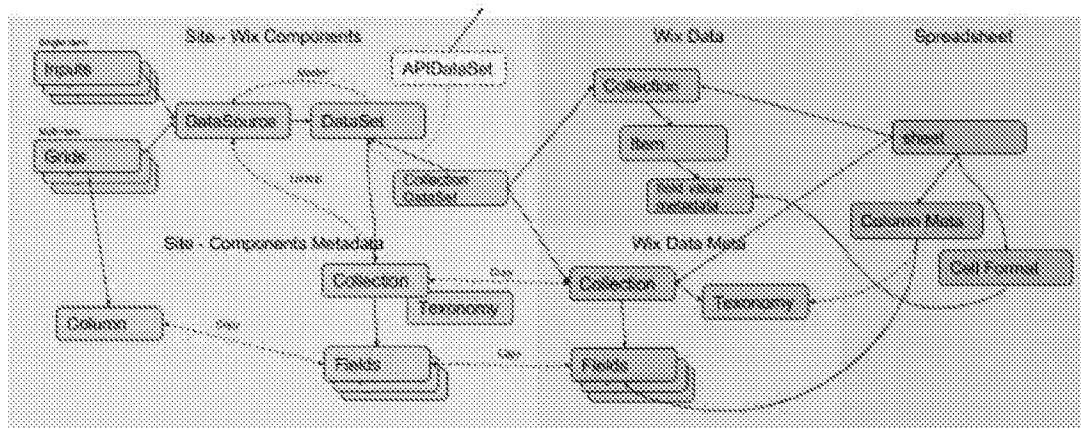
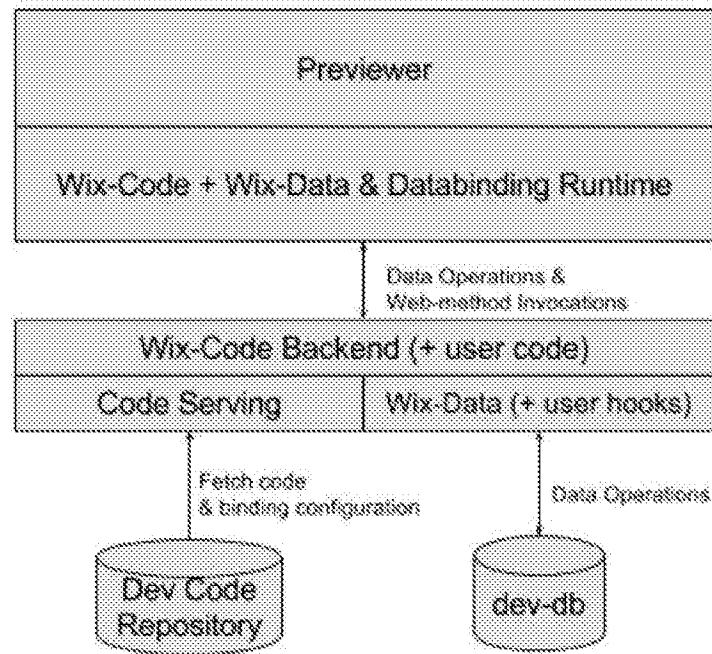
The simplest type of dataset is a **collection dataset**. Such a dataset simply retrieves data from a specific collection, enables sorting and filtering it based on the field metadata of the underlying model, and allows its consumers to iterate over records and potentially create new ones and update existing ones. The dataset is a **query-based dataset**. Such a dataset is driven by a query over a single collection.

Editor elements can be bound to a dataset. In particular, a property of the editor element is bound to a collection field representation in the dataset. Multiple elements can be bound to the same dataset field.

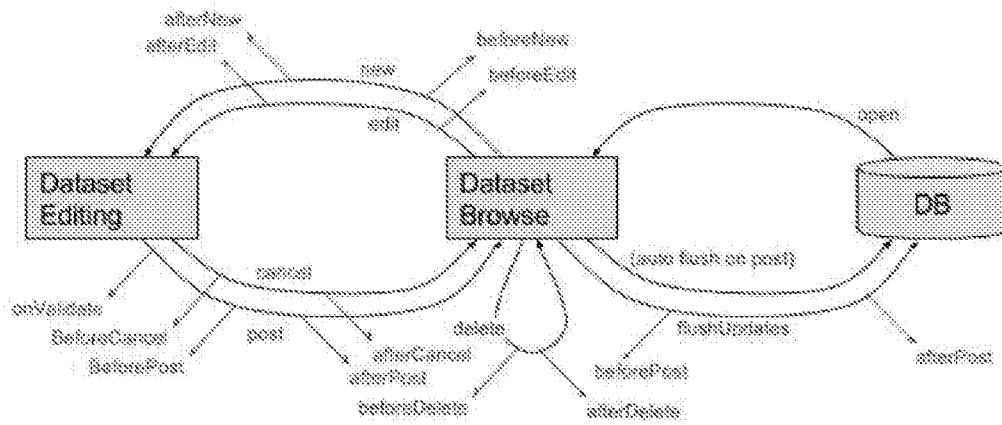
At runtime, the following (simplified) scenario takes place:

- A page loads
- Page code gets executed. As part of that, one or more datasets are created (more on how datasets are attached to pages below).
- The dataset now wires itself to UI constructs. This happens as follows:
 - Locate page components that are bound to this dataset.
 - For each, create a matching **adapter** that facilitates the interaction between the dataset and the component.
 - The adapter registers itself as a listener on change events from that component, so it can relay the changes back to the dataset.
- Now the dataset is populated by executing its data retrieval capability.
- Once initial data is available, by using the relevant adapters, data is set to the relevant data-bound controls.
- Now, the dataset's state may change as a reaction to one of many events:
 - Calls to "next record", "prev record", etc.
 - Calls to "next page", "prev page", etc.
 - Changes to the current record via user inputs
 - Etc.
- On each such change to the datasets state, it fires update commands via its adapters to all its bound components
- Also, the dataset may "**commit**" changes to its backend store when requested to. This can happen either automatically when the "current record" pointer moves to a new record, or explicitly through a "commit changes" API call. More on this later.

High Level Preview Time Architecture



Dataset states



11 – CMGR

Content manager

Content Manager (CMGR)

CMGR is Excel-like data management application which enables the user to manipulate data that is stored in Wix Data collections. The application consists of two major parts: CMGR Editor application and CMGR My Account application, residing in respective parts of Wix site.

Content Manager in Editor (CMGR Editor)

CMGR Editor is opened by Client Side Web Worker. It gets Editor SDK which is mediator between Editor API and CMGR and enables CMGR to manipulate the state of Wix site.

CMGR Editor enables to:

- view and edit test/sandbox data of a site (using Wix Data API)
- edit collection permissions (using Editor SDK to modify files in Cloud VFS)
- view and edit Wix Data hooks (editing happens in an external panel in Santa Editor, modifications are stored in Cloud VFS)
- sync collection data from sandbox to live and vice versa (using Wix Data API)
- preview live data (using Wix Data API)
- manage schemas: create new ones, add/edit/remove fields (using Editor SDK to modify files in Cloud VFS)

Content Manager in My Account (CMGR My Account)

CMGR My Account is opened by Wix Dashboard. It enables to:

- view and edit live data of a site (using Wix Data API)
- Add temporary (undefined) fields to data which are not persisted in schema

Content Manager grid

It is primary component both in CMGR Editor and CMGR My Account. It uses collection's schema to construct columns of the grid. Schema is stored in Wix Code filesystem associated with a particular Wix site. It uses user's data loaded from Wix Data to construct rows of the grid. It enables to:

- Manage collection items (rows): add, edit, remove, duplicate (using Wix Data API)
- Toggle visibility of fields
- Filter data (using Wix Data API)
- Sort data (using Wix Data API)

Data types

CMGR handles multiple data types: strings, numbers, Booleans, dates, page links, images, JavaScript arrays and objects, references.

Strings, numbers, Booleans, dates, arrays and objects are primitive types (natively supported by JavaScript).

Pagelinks are special strings that are not stored in a collection and are constructed by wixData on runtime given a pattern for a Dynamic Page. A pagelink can be used to link to a dynamic page in a Wix site.

Images are either strings to external resources or special Wix Media strings referencing files stored on Wix Media servers. User has ability to upload images via Wix Media Dialog.

References are pointers to other collections, used to enable joins between Wix Data collections.

Implementation

The application is rendered by open-source framework React:

<https://github.com/facebook/react>

Application state is handled with open-source library redux: <https://github.com/reactjs/redux>

The grid is rendered by open-source library ag-grid consisting of three components:

1. ag-grid (<https://github.com/ceolter/ag-grid>).
2. ag-grid-enterprise (<https://github.com/ceolter/ag-grid-enterprise>)
3. ag-grid-react (<https://github.com/ceolter/ag-grid-react>)

12 – DSEO

Dynamic SEO

Wix Code - SEO subsystem description

The “regular” SEO renderer for Wix sites takes the static JSON data that represents the site and uses it to render a stripped-down version of the site for search bots (without using the regular rendering logic which is implemented on the client side).

Since Wix Code introduces the ability for the user to change the site dynamically, we need a different/additional mechanism that can render the site in a way that includes those changes, which can include dynamic page routing and arbitrary user logic.

Main sub-components of the SEO subsystem

- A dispatcher service (“Cloud SEO Dispatcher”) that receives RPC requests from the Wix SEO renderer and dispatches them to available workers for processing
- A grid of “Cloud SEO Workers” that process rendering requests.
Each worker runs a headless Chrome browser that is used for running client code (a.k.a. “Mini Santa”) that does the actual processing.
(including running user code, etc.), and produces the result as a set of changes to apply on the static data structures that represent the site.
- A management service (“Cloud SEO Lifecycle”) that keeps track of available workers and synchronizes them with any configuration changes.

Rendering flow

The regular SEO renderer, when it recognizes that the site is WixCode-enabled, calls the “Cloud SEO Dispatcher” with the relevant data for the rendering, and expects as a response:

- Diffs to apply on the site’s JSONs (for masterPage + current page)
- In case of dynamic page routing, the pageID of the page to render + SEO metadata to render

The “Cloud SEO Dispatcher” hands the rendering task to one of the available “Cloud SEO Workers”.

The Worker tells its managed headless Chrome instance to render the site by calling Mini Santa’s rendering function with the parameters of the site to render.

The rendering function runs the code that renders the site, but instead of showing the result on the screen it computes and outputs:

1. The differences between the initial data structures of the static site and the runtime data structures after running the user code.
2. Dynamic page information - which page to render and what SEO metadata to add to the rendered result.

These resulting differences travel back up the chain of calls, up to the Wix “regular” SEO renderer, which applies them to the initial static data and renders the final HTML that is sent back to the search engine crawler.

13A – CFN

Compute functions

Wix Code - Compute Functions

Abstract

For any Wix site, a user can have an ability to write a function which will be exposed as a web interface for the site via standard HTTP protocol.

This ability is valuable for server to server integrations, for example registering to 3rd party applications via a webhook, or exposing a REST API for the web site.

Adding a Function

Via the Wix Code IDE a user has the ability to add a *compute function* - which will open a dedicated page for writing the code for it. The name of the function represents the web path prefix relative to the web site address, via which the function can be called using HTTP protocol.

For example, the function named **compute** will be accessible via the URL:

`http[s]://mysite.com/_api/endpoints/compute/...`

User can also optionally define which HTTP methods (GET / POST / PUT / DELETE) the function will be available for. The code is stored in a multi-tenant service, called VFS under the user's site dedicated space.

The user will have the following HTTP request information available for processing:

URL, query parameters, headers, payload, HTTP method.

The user can generate an HTTP response with the following parts: status code, payload, headers.

Execution of a Function

After the function is available for HTTP access, and an HTTP request is called for a wix code web server (dispatcher), first validation is performed for testing the existence of a function definition. If the function does not exist - status 404 (Not Found) is returned. If the function exists, the code will be executed in an isolated docker container via the *elementory* application server (ELM) running there.

Testing Before Publishing and Debugging

During the development phase of a *compute function* it is possible to test the function via a test HTTP URL, which is supplied via the Wix Code IDE. It is also possible to see the logs of the function execution be it already published or currently developed.

13B – CFN

Compute functions - detailed

Wix Code - Compute Functions

Overview

This document will outline the functional requirements and the technical specification proposal for exposing Web Endpoint as an API via HTTP.

This document is based on the [research paper](#)

Functional Requirements

Ability to expose a method for web access without authorization

- Ability to define supported HTTP methods: POST, GET, PUT, DELETE (on the same path)
- Ability to extract (read) the request's HTTP method, path, headers, query parameters and body
- Ability to write the response's HTTP status code, headers
 - Nice to have: Ability to define the HTTP *Accept* header for the requests
 - Nice to have: Ability to define the HTTP *Content-Type* header of the response
- Unauthorized access to *data* should be handled (enforced) by the *data* layer

Web API HTTP Specification

When a method is exposed for web access, the method will be exposed with the following specification.

URI

Free Sites

Published site: https://{user_name}.wixsite.com/api/endpoints/{site_name}/*

Development: https://{user_name}.wixsite.com/api-dev/endpoints/{site_name}/*

Premium Sites

Published site: https://{user_domain}/api/endpoints/*

Development: https://{user_domain}/api-dev/endpoints/*

The `/_api/endpoints/` (and `/_api-dev/endpoints/`) relative path to the site root will be the root entry point to the whole user defined APIs. We will call this entry point as the *API root*.

From user perspective, this URL will be permanent one. The use cases when the URL is going to change is:

- In free site - user changed the site name
- In premium site - user changed the domain
- When user changes internal sub paths after the *API root*.
- Transferring site - domain name can be changed
- Upgrade to premium - buying a domain / assigning an existing domain

All to all - seems that backward compatibility of the API is solely user's responsibility.

Security

The URLs are on the same domain as the user's site. The potential security risk may arise if those URLs are accessed via browser, and the user writes some unintentionally non-secure code.

We can address CSRF and XSS attacks using the following:

- CSRF - employ cookie / header validation for browser user agents
- XSS - employ Content-Security-Policy functionality by default on the user's domain, with option to opt out or to define the whitelist

We could further improve security by using SSL.

Request

- **Allowed HTTP methods:** according to the definition of the *endpoint*
- The function's code is responsible to extract the request's body, headers, query params, IP

Response

- **Response's status code:** responsibility of the *endpoint's* code
- It is the responsibility of the *endpoint's* code to supply a response's body with the appropriate format
- Response headers can be supplied also

Wix Code Design

Preface

General direction is to differentiate the Web Modules used by the *public* site scripts from the definition of the *methods* which are exposed via web publicly for any HTTP consumer - let's call them *compute functions*. It will allow simpler design and implementation, when a shared code is needed to be used from the *public* script and at the same time it's result default format cannot be used as is for the web *compute functions* exposure and needs some massaging.

It will also allow to define web *compute functions* which are totally unrelated to the site's presentation layer (the *public* scripts).

For example, a web compute function which listens to github webhook, parses the data and stores some info in a database collection, and there is a dynamic page which present the data from the same collection. No public site script is involved here.

The Location

We will store the compute function's code in a dedicated JS file. User will have an ability to define *compute functions* - and the UI will open the file where the code of all the *compute functions* will be located.

Maybe we need to revisit this aspect, if the code in that file can becomes big.

How to Define a Compute Function?

There are 2 aspects of a *compute function*.

1. **The design time aspect** - how the user defines a *compute function* and configures its various aspects, such as the HTTP method, the request's content type it accepts, the response content type it generates, etc.
2. **The runtime aspect** - once such a *compute function* is defined - an appropriate HTTP *endpoint* should be exposed and when called to be redirected to running the *compute function's* code, and when an existing *compute function* is deleted - an existing HTTP *endpoint* should be removed. Of course, this should be available only after the publishing of the site, or before publishing for testing purposes of the *compute function* by the user.

The Design Time Aspect

- We will define exported functions for each compute function prefix
- We will have the following prefixes for the functions: *get_*, *post_*, *put_*, *delete_*, *use_*
- The function name after the prefix will represent the web endpoint's path prefix
- **TODO: still need to close the UI aspect of how to define new compute function**

```
export function get_example(request) {  
  // all GET requests for the URL  
  // http[s]://{user_domain}/_api/endpoints/example/* will be routed to here  
  ...  
  return response  
}  
  
export function post_kuku(request) {  
  // all POST requests for the URL  
  // http[s]://{user_domain}/_api/endpoints/kuku/* will be routed to here  
  ...  
  return response  
}  
  
export function use_jojo(request) {  
  // all HTTP requests, no matter which HTTP method is called for the URL  
  // http[s]://{user_domain}/_api/endpoints/jojo/* will be routed to here  
  ...  
  return response  
}
```

For some other *rejected* alternatives, please see *Appendix A*.

Pros:

- Simple definition, can be easily assisted by UI
- Configuration is close to the code
- No reflection is needed

- Using the current publicly available spec of JS

Cons:

- Dynamic binding of functions to HTTP calls - no predefined configuration exists
- Ambiguous definitions can occur - e.g. `get_` and `use_` on the same endpoint
 - Can be solved by simple precedence algorithm, which should be documented. Specific methods take precedence over `use_*`
- Some characters cannot be part of the prefix (e.g. the '-' character)
 - All characters which are not supported to be located in JS function name

Review: it might be that on some levels of redirection at wix infrastructure, some additional internal headers are added along the way. This could be resolved by avoiding supplying request headers. On the other hand, if there is such a case, a sensitive header is transferred to a service, then it probably security issue of the communication protocol, which should be avoided anyways, and could be resolved by redesigning the relevant protocol.

User runtime is considered a non-secure environment today as well, so we don't keep or pass any sensitive info into user containers. If there's anything we want to prevent user access to, we can strip it down during dispatching.

Request and Response Data Structure

We will strive to have a unified data structure for the request and response objects as for the routers feature.

```
request = {
  baseUrl: String, // the domain of a premium Wix site or the url of a free Wix
                  // site
  path: String[], // the relative path to the API root decomposed by parts
  url: String,    // the full url the function was accessed by
  ip: String,     // the original client IP
  query: Object,  // this object has a key/value of a query param name
                  // and its value respectively
  headers: Object, // this object has a key/value of a header and its value
                  // respectively
  body: Buffer,    // the body of the request
  httpMethod: String // "GET", "POST", ...
}

response = {
  status: Number, // an integer number represents the response's status code
                  // We might reuse here the routers Responses functions, e.g.
                  // ok(), forbidden(), notFound(),...
  body: String,   // the body of the response
  headers: Object // this object has a key/value of a header and its value
                  // respectively
}
```

Factory for Response

// There are ready responses from a response factory.

// Include wix-compute-functions.js - which exposes the following functions

// Generic response

response({ **status** = 200, **body** = "", **headers** = {} } = {})

// 200 OK response

ok({ **body** = "", **headers** = {} } = {})

// 201 Created response

created({ **body** = "", **headers** = {} } = {})

// 404 Not Found response

notFound()

// 500 Internal Server Error response

serverError()

// 400 Bad Request response

badRequest()

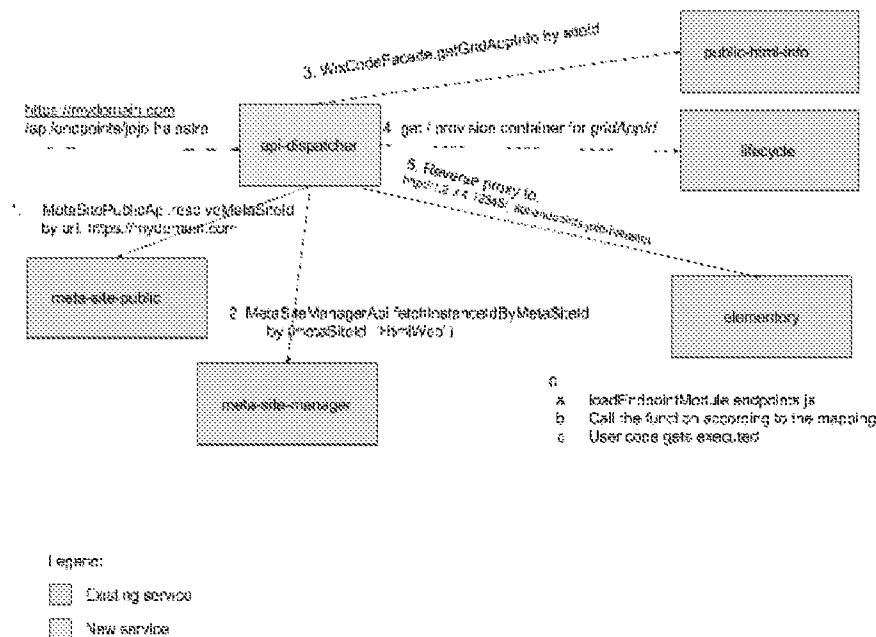
```
// 403 Forbidden response  
forbidden()
```

The Runtime Aspect - Mapping for Execution

Reminder - so we have a file called *compute-functions.js* for each site, where the endpoint's URL to function mapping is located together with the function's code itself.

Mapping Site to Wix-Code Compute Function

- Upon creation of a compute function via UI - a file under the *relevant site context* will be created in the elementary in the path */compute-functions.js*



It seems, that there is no easy way to get the *siteId* from the domain. There are 3 RPC calls in order to understand which *gridAppId* we should launch.

We probably want to cache those 3 RPC calls, however, in case of a change of *metaSiteId* / *siteId* / domain - we need event to update the cache, or update the cache by some periodic poll, which will imply eventual consistency (in any alternative of cache update, since we cannot rely that the events are bulletproof). From our understanding, there is no event of *metaSiteId* change or domain change.

The proposal is to cache *domain* -> *gridAppId*. *gridAppId* can potentially be changed after a save, so we will invalidate the cache on each save - this event is already handled in wix-code. It would be better, if we had listened to the publish event, if such exist.

If there is a change in domain name / free site name - the new domain will not be in cache, so first it will perform the whole flow for this new domain again and cache.

The cache should have some eviction policy based on last touch time.

The Runtime Aspect - Preview vs. Published

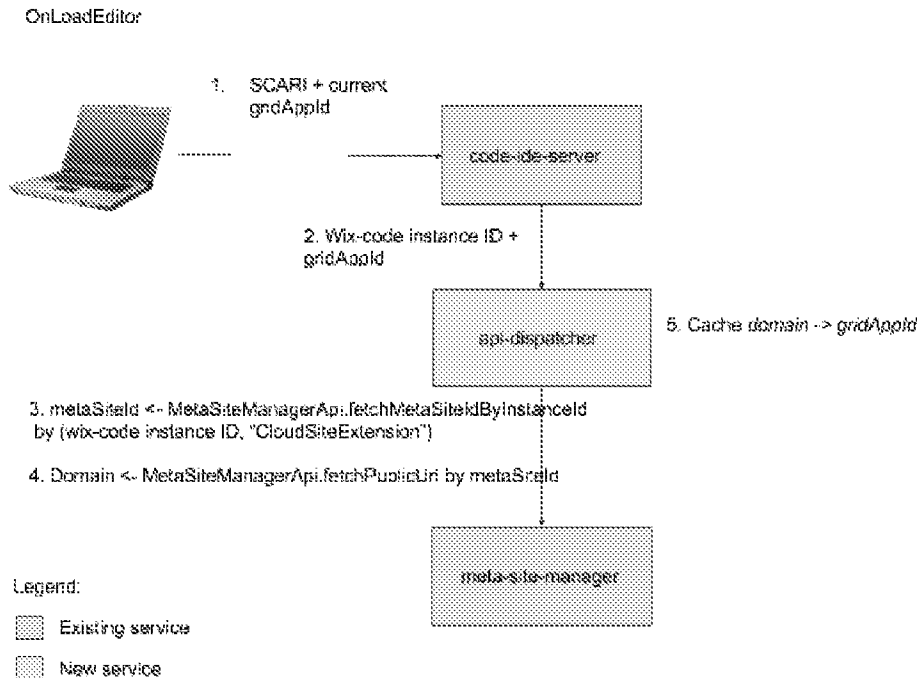
When user develops his compute function - should we supply some "dev mode" for the web endpoint before publishing?

For new functions - it is useful if the function's code manipulates data. User wants to make sure the new code doesn't corrupt existing data, thus wants to test the code prior publishing. In addition, when a compute function is already published, and user needs to update it, for instance to support some new protocol, he doesn't want to break the existing functionality of the function, until he is sure that his newer function's code is running as expected.

How this capability should be supplied? Meaning - expose the updated function to the web which will work in parallel to the currently published web endpoint?

The Algorithm

- When editor client loads, it sends its SCARI to the api-dispatcher service (via ide-server)
- The api-dispatcher service will have a mapping *domain* -> *gridAppId* for testing
 - We might want to separate the editor segment (testing) and the public segment (published sites api access)
 - This *gridAppId* will represent the current developed version in editor, it can be the published one, if no changes occurred since publishing, or a newly developed one (the not saved one)
- Whenever there is a call to wix-code-service of *cloneCodeApp* / *duplicateCodeApp* / *createApp* - the newly generated *gridAppId* will be propagated to the api-dispatcher service
- Accessing currently developed version will be via the *development* URI as stated in the URI section



And then, when the request for *api-dev* is received by *api-dispatcher*, it will look the cache for *gridAppId* by domain. If not exist, it can fallback to the similar protocol as for published sites, and cache the last saved version's *gridAppId*.

NOTE: Problems can occur, if domain is changed when the editor is opened. We need to listen on such an event if exists, and update the cache accordingly, or declare a limitation.

Another alternative can be that on editor load - we will cache the *wix-code instanceId* -> *gridAppId* which was sent by the client (without resolution of the domain), and then on actual *api-dev* request similar flow to resolve the domain as on the above diagram, and cache also the *domain* -> *gridAppId*.

And then, when *cloneCodeApp* / *duplicateCodeApp* occurs - update both of the caches by previous *gridAppId* (we will have also inverse index by *gridAppId*).

This will also solve the "changed domain" problem.

This seems rather a complex mechanism. Can't we have a limitation, and require to save before testing? Or maybe some other "on demand" event for testing an API, say - *test API*?

Debugging

One of the pains in the hackaton with webhooks was the lack of debugging capabilities for the backend code. We will need to add some capability, it is a product open question.

Preferred Solution

User will have a web endpoint for continuous stream of *console.log* outputs, similar to [this](https://webtask.io/docs/api_logs).
[https://webtask.io/docs/api_logs]

We will need to consider this carefully, this can be very resource consuming.

Other Alternatives

- Supply some logging API, and have some rolling log file per site, which user will have the ability to download via the editor.
- Open a special UI in *dev-mode*, where those logs will be printed
 - Actually this can be an addition to the *preferred solution* above
- Developer can supply a header, say *x-debug* and in return will receive a response header with a unique URL holding the debug information about the request. Up to 50 URLs per site. The info will be held in memory for 5 minutes
 - There should not be any problem memory wise, if this is kept in-memory of elementary, given, that we have 200 elementary per machine.

Appendix

Rejected Design Time Aspect Alternatives

Alt. 1: Almost a copy / paste from JAX-RS API

It would be nice if we could use *@decorators*, however, it seems that it will be available for the next spec of JavaScript, ES8.

Babel transpiler supports it in some experimental mode.

We can then use reflection to search all those compute function methods and map them accordingly to the configuration.

```
@get("example/path") // can be also @post
// a web endpoint will be exposed for GET HTTP method
// at the following url for example:
// https://mydomain.com/_api/endpoints/example/path
@consumes("application/json")
// expects application/json content type for the request
@produces("text/plain")
// defines the content type of the expected response
function webEndpoint(request, response) {
  ...
}
```

Pros:

- Clear and easy declaration

Cons:

- Seems like such mapping of *request's Accept* and *response's Content-Type* headers will be difficult for the dispatcher, if for example wildcards are used. Maybe should be dropped
- Using non-finalized spec (currently stage 2 - <https://github.com/tc39/proposal-decorators>)
- Reflection is needed to find the relevant mappings

Alt 2: Configuration of the Compute Functions is Stored Separately from Code

We can supply a special method, by predefined name, where the mapping will be declared by the user.

```
export default function wixConfigComputeFunctions(config) {  
  // config has the following method:  
  // config.register(method, path, functionName)  
  // maps HTTP method, path and a function to execute  
  
  // the function is expected to be of the following signature:  
  // function someFunction(request, response)  
  
  // exact path mapping  
  config.register("GET", "example/path", "examplePath");  
  
  // wildcard path mapping - all subpaths of /kuku will be mapped to this function  
  config.register("POST", "example/kuku/*", "kukuPath");  
}  
  
function examplePath(request, response) {  
  ...  
}  
  
function kukuPath(request, response) {  
  ...  
}
```

Pros:

- Pretty simple declaration
- No reflection is needed
- Using the current publicly available spec of JS

Cons:

- Function name is declared as string - typos can occur
- Configuration is "far" from the method itself

Alt 3: Express-like Configuration

We will have some global object, which will be act as registrar for the compute functions, e.g. *WixComputeFunctions*.

```
// exact path mapping
WixComputeFunctions.get("example/path", function (request, response) {
  ...
});

// path and all its descendants
WixComputeFunctions.post("example/kuku/*", function (request, response) {
  ...
});
```

Pros:

- Simple definition, express like API
- Configuration is close to the code
- No reflection is needed
- Using the current publicly available spec of JS

Cons:

- Exposing global object
- Ambiguous definitions can occur - need some precedence algorithm

Alt 3.1: Configuration Function with Express-like Syntax

An express like pattern for configuring the HTTP method, path and the function to execute.

```
export default function wixConfigComputeFunctions(config) {  
  // config has the following methods: get, post, use  
  // config.get(path, function)  
  // maps HTTP method, path and a function to execute  
  // when used use - any HTTP method will call the function  
  
  // exact path mapping  
  config.get("example/path", function (request, response) {  
    ...  
  });  
  
  // wildcard path mapping - all subpaths of /kuku will be mapped to this function  
  config.post("example/kuku/*", function (request, response) {  
    ...  
  });  
}
```

Pros:

- Simple definition, express like API
- Configuration is close to the code
- No reflection is needed
- Using the current publicly available spec of JS

Cons:

- Ambiguous definitions can occur - need some precedence algorithm

14 – WDB

Wix DB

Overview

DocStore is a generic multi-tenant document-style data service that exposes a RESTful API for managing data. Every tenant can have several separate virtual databases. The service also has an administration server-to-server (not user facing) API for managing tenant's databases and possibly disabling a tenant.

Storage Model

The underlying database used to store data is MongoDB and the service supports multiple physical MongoDB databases for sharding purpose.

Virtual/Physical Database Mapping

Upon first request and after it passes authentication check, each tenant virtual database is assigned to one of the physical MongoDB databases the service is working with and from that moment on, all incoming requests of that tenant's database executes against that physical database.

Virtual tenant database to physical database mapping is maintained in an administration database and hot mapping records are kept in an LRU based in-memory cache.

Virtual Database Model

Each tenant virtual database can contain one or more collections of documents. Each virtual collection is mapped to one physical MongoDB collection. Collection mapping is done using a simple namespacing system. Each physical collection is named <tenantId>@<vdb-name>@<collection-name>. Accordingly, when a request comes in, after it passes authentication check, the tenant ID, target database and collection name are resolved, the physical collection name is constructed and can be accessed.

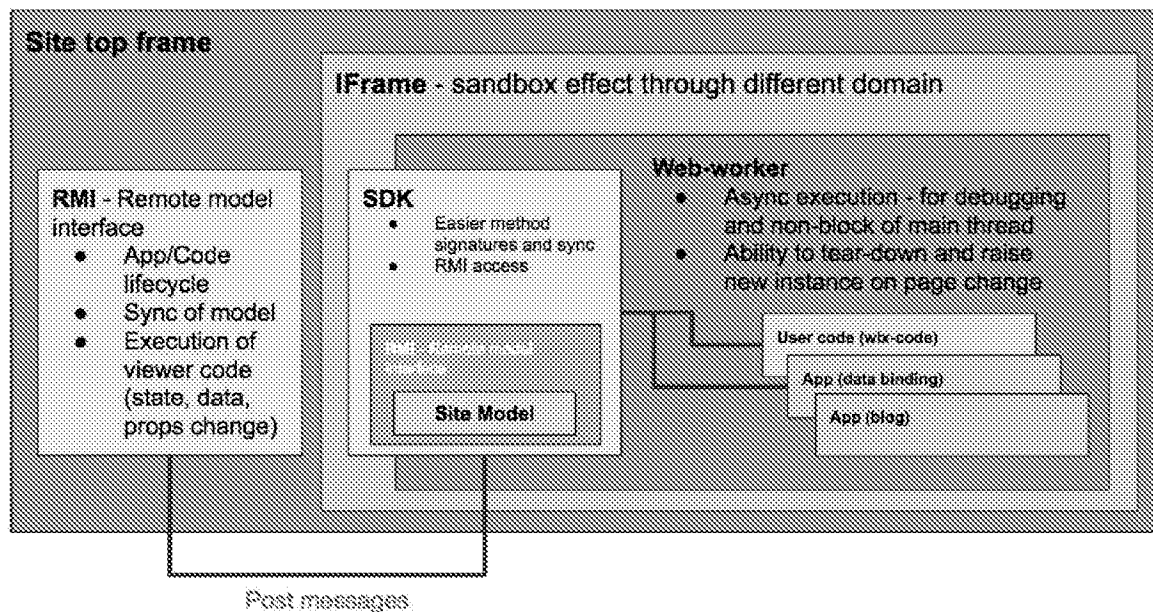
Physical Database Allocation

When a new tenant database is introduced it needs to be assigned with a permanent target physical database. Physical database selection is done based on database size right now. A background task is scheduled to run periodically and collect statistics from all the active databases the service is configured to work with. Statistics are stored in an in-memory data structure, so that when it's time to allocate a new database, the smallest one can quickly be selected as a target.

15 – \$WAPI
\$W API

\$w API

\$W API is the way we allow users to interact with viewer components using code. The SDK is used to wrap the internal communication layer used between the worker, where the user code / extension is running and the runtime (AKA the viewer). Following is a diagram that represents the logical architecture:



The need:

The internal model of RMI is too complex for users that want to control and modify only parts of the system, and is undocumented and contains proprietary information needed in order to run. The \$w SDK wraps this, and turns it into a more common pattern of locating elements and manipulating them.

Coding pattern:

The coding pattern supported by the \$w SDK contains of 2 parts:

1. **Page lifecycle events:** Using lifecycle methods, a code using the SDK is triggered at predefined stages. During these stages, the SDK is available, and the code can perform various tasks, such as external data loading, initialization of components or event registration. Currently a single method is available, which happens before the page is presented, and called `PageLoad`. By returning a javascript promise, the code can defer the presentation of the page until all asynchronous information is loaded (such as external data coming from wix-data or external services)
2. **Selection:** \$w is representing the global context of a page. Similar context objects can exist, i.e. an \$item when drawing repeater items. Using the context, the user selects elements using a string query. Selectors are similar to regular CSS selector, with small differences. Selection can return a single element, or multiple elements. For example:

\$w('#myElement') - this would select the element with myElement ID

Selectors have the following types:

- ID selectors - selecting using a page unique ID
 - Type selectors - selecting using a type of a component
 - Array of types selector
 - Role selector - this is used by controllers, to select using their own pre-defined names. An element can have an ID, and multiple roles, given by the controller
3. **Element manipulation:** once a reference to an element is acquired, the element is providing with methods to manipulate it. Some methods are available on all elements, such as hiding and element and showing it. Some methods are available only on certain elements. The full list is provided in the [SDK documentation](#)
 4. **Event registration:** each element supports a pre-defined set of events. The code that is using the SDK can register to these events, and provide a callback methods that will be called when they occur. During the callback, selection and manipulation is possible.

Appendix

A6

Database overlay is an algorithm to have a diff database for staging that is over a live production data, such that it appears like a full staging database with the following properties

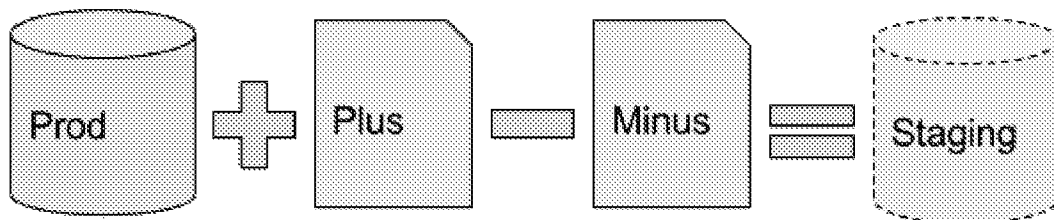
1. The staging data is based on the live data
2. Any update in live is visible in staging, if not overwritten in staging
3. Any write goes to staging
4. Any delete goes to staging
5. Any update goes to staging
6. Queries retrieve data from staging or live, such that any updates, delete or insert in staging effects the query result without affecting the live data
7. The staging database can be dropped at any point
8. Creating a new staging database is an atomic operation without any data copy
9. O(1) operations

The algorithm

For the algorithm, we define the following

1. Prod - the original production database
2. Plus - table for new items in staging. Plus has the same schema as Prod
3. Minus - table for items removed from staging. Minus has the same schema as Prod
4. Staging - the effective database (virtual)

The concept



Inserts:

When inserting a new record in staging, it is inserted into Plus.

Updates:

When updating a record in staging, we check if the record exists in Plus.

If the record exists in Plus, we update it.

If the record does not exist in Plus,

we copy the record (by primary key) from Prod to Minus

And we insert the updated record to Plus

When updating a record in Prod, we check if it exists in Minus

If the record exists in Minus, we update it there as well

Delete:

When Deleting a record from Staging

We insert the deleted record into Minus

If the record exists in Plus, we delete it from Plus

Get:

When getting a single record by id, we check

if it exists in Minus, we return no record found

If the record exists in Plus, we return the record from Plus

Else we return the record from Prod, if one exists there

Query:

We perform three queries - all with the same filter and sorting. To make sure we have unique sorting, we add also sorting by primary key as a last sort.

Q_{Prod} -> on Prod

Q_{Plus} -> on Plus

Q_{Minus} -> on Minus

We traverse the cursors in parallel walk, such that for each record

If it exists only in Q_{Prod}, we return the record

If it exists only in Q_{Plus}, we return the record

If it exists in both Q_{Minus} and Q_{Prod}, we skip it.

parallel walk of result sets assumes that all the result sets are sorted the same. Then we find which of the result sets has the lower record (according to the current sort) - can be one or more result sets. We process this record, then move forward only the result sets that are on the lowest record.

Example

The data in the tables:

Prod				Plus				Minus		
A	B	C		A	B	C		A	B	C
1	q	a		3	zz	d		2	w	s

2	w	s		5	r	zz		3	e	d
3	e	d		7	w	w		5	t	g
4	r	f								
5	t	g								
6	y	h								

Query, order by B:

Q _{Prod}				Q _{Plus}				Q _{Minus}		
A	B	C		A	B	C		A	B	C
3	e	d		5	r	zz		3	e	d
1	q	a		7	w	w		5	t	g
4	r	f		3	zz	d		2	w	s
5	t	g								
2	w	s								
6	y	h								

When doing parallel traversal of the result sets:

Prod				Plus				Minus		
A	B	C		A	B	C		A	B	C
3	e	d						3	e	d
1	q	a								
4	r	f								
				5	r	zz				
5	t	g						5	t	g
2	w	s						2	w	s
				7	w	w				

6	y	h								
				3	zz	d				

Merging the results we get

Result		
A	B	C
1	q	a
4	r	f
5	r	zz
7	w	w
6	y	h
3	zz	d

Query with paging - option 1:

by doing client side traversing of the merged query data.

Problem - not a O(1) operation

Query with paging - using random search:

Suggested algorithm (need to validate)

Given query on staging with skip N rows.

Step 1 - find the record at position N on prod

First stage, to figure out the skip parameter on Prod, Plus and Minus.

R_{prod} = find the record on prod that is at position N.

Select * from Prod where <filter> order by <sort> skip N limit 1

Step 2 - find the offsets on plus and minus for that record

Find the offset of this record in Plus in Minus

$P_{Plus} = \text{Select count(*) from Plus where (items} \leq R_{prod} \text{ by current sort)}$

$P_{Minus} = \text{Select count(*) from Minus where items} \leq R_{prod} \text{ by current sort}$

The index of R_{prod} in staging is now

$$P_{Staging} = N + P_{Minus} - P_{Plus}$$

Step 3 - walk to position N on staging

We have three cases now - going forward ($P_{staging} < N$), going backward ($P_{staging} > N$) or a match ($P_{staging} = N$)

Going forward ($P_{staging} < N$)

We perform queries, with the filter, and sort

- Prod with skip N
- Plus with skip P_{plus}
- Minus with skip P_{minus}

When we do parallel walk $N - P_{staging}$ steps forward. We record the number of steps on each table (denote those as O_{prod} , O_{Plus} , O_{Minus}).

Our skip numbers for prod, plus and minus to get effective skip of staging.

$$S_{prod} = N + O_{prod}$$

$$S_{plus} = P_{Plus} + O_{Plus}$$

$$S_{Minus} = P_{Minus} + O_{Minus}$$

Going backward ($P_{staging} > N$)

We perform queries, with the filter, with reverse sort

- Prod with filter records $\geq R_{prod}$
- Plus with filter records $\geq R_{prod}$
- Minus with filter records $\geq R_{prod}$

When we do parallel walk $P_{staging} - N$ steps backward. We record the number of steps on each table (denote those as O_{prod} , O_{Plus} , O_{Minus}).

Our skip numbers for prod, plus and minus to get effective skip of staging.

$$S_{prod} = N - O_{prod}$$

$$S_{plus} = P_{Plus} - O_{Plus}$$

$$S_{Minus} = P_{Minus} - O_{Minus}$$

Match ($P_{staging} > N$)

Our skip indexes are N , P_{Plus} and P_{Minus}

$$S_{prod} = N$$

$$S_{plus} = P_{Plus}$$

$$S_{Minus} = P_{Minus}$$

Step 4 - do the queries and process the query

We do select from each of the 3 tables with current filter and sort, adding skip values S_{prod} , S_{plus} , S_{Minus}