BACKBONE

Development Best Practices & Expresia

Author: Thanos Arnaoutoglou

Introduction

The current document highlights principals, best practices, ethics and tips that every developer should embrace in order to improve the quality of their work.

Ethics

Starting off, we will tackle the most important thing a developer should have. That would be certain work ethics.

Ownership : A developer must always be accountable for their work. Be proactive in making your code the best it can be. Ask questions, gather information and do whatever necessary to carry out the work.

Raise Flags : A developer must raise red flags when they see them. Not everyone is capable of understanding the technical implications a feature may have. Be sure to notify as early as possible.

Measure twice cut once : Everyone should always count their words, review their estimates, review their emails, run every test possible before deploying code.

Remember Murphy's Law : "Anything that can go wrong will go wrong." Come prepared.

Keep your word : Try to always be “a man of your word”. Try to provide accurate estimates. Try to meet deadlines and if everything goes wrong, take the blame and fix it.

Do no harm : A developer must make sure that their code would cause no harm to the client, user, company or others. We are not making bombs here, but errors that cause damage in cost or reputation must be avoided.

Say No : Not everything is possible / reasonable. It is OK to say No, but make sure to explain why.

Be profitable early on : Try to build a raw final implementation and add bells and whistles afterwards. A messy code that works in time is better than a clean “by the book” code that doesn’t.

Don’t create a mess : A developer should avoid leaving messy code, despite what’s suggested above. Always clean up once you’re done.

Leave things better than you found them : A developer should always improve his work when he revisits it, even by a space or a comma. Correct indentations, add spaces, write a comment, rewrite something.

If it ain't broke don't fix it : Do not overdo it with improvements suggested above. If something works as is, there is nothing wrong with it.

Development Best Practices

Regardless of stack, there are things that differentiate a good code from a bad one. In general everything listed below could fall under one rule: "Think of the future developers that will have to work on your code".

File Specific : Never do inline css, use script and style tags within the html.

Naming Conventions - General Control : Since all CSS and JS are going into their specific files, it is OK to have multiple files for specific reasons. Make sure you are using easy to understand naming conventions.

Readability : Again, naming conventions. Make sure that you are using English throughout your work. Make sure the code is clean and easy to read, it uses proper indentation etc.

Comments : Self explanatory code along with comments is the best way to describe what you are trying to accomplish.

Small Custom code > 3rd Party Library > Reinventing the Wheel : Developers should write high performance oriented code. Writing a small script is always better than loading an entire library, but using a library is better than doing a big implementation from scratch.
Use GIT : "If it's not on Git, it did NOT happen". Given that currently Expresia uses the dev and master branch, code should always be written in the dev branch and deployed only once completed.

Check markup performance : Markup should be tested for PageSpeed performance before being put in Expresia. If for whatever reason development takes place directly in Expresia then there should be an early PageSpeed testing along with /XprTrace (see below), so that issues can be addressed from early on.

Share ideas : Make sure to share ideas / implementations that worked well. Make a case study and share it with fellow developers. Also ask for suggestions, there is nothing wrong with asking for opinions.

Expresia Tips & Best Practices

Below there are a few Tips and Best Practices for developing in Expresia.

/xpr/apis : A great list of every api available in the current instance. Apart from available methods and documentation. There is a tool that can be used to hit the API on the fly and build queries. This is a great tool to experiment, without the need to start building infrastructure. See it as POSTman for your instance.

/XprTrace : A great tool for reviewing what is going on with your code, both performance and data wise. Use it.

{{{XprDump}}} / {{{XprJson}}} : Two very useful custom hbs tags that allow developers to review their data using a json parser.

/xpr/XprCQR : A great tool that provides suggestions of pages that something might be going wrong. Not the holy grail, as sometimes there is no better way to implement a feature, but a great indicator nonetheless.

Less of Everything : Try to use less Data Sources, less SSJS, less hbs logic, less Custom Fields and less JS. Using context to pass information between elements is a great way of reducing the number of API calls. Giving Custom Fields a more generic name helps in reusing them.

Remember Tag Manager : Tag Manager requires a piece of script to be pasted directly after the <body> tag. It is mindful to create an element for that and place it in every template. That way you eliminate having to go into multiple places to paste the script (in general Navigation works too, but there are cases where this wouldn't work).

Playlist is just another resource : Because we all use the generic playlist bundle, we might forget (or not learn) that there is more use in playlists than selecting the SKIN. For example it can be used as another layer of filtering articles within a section.

Use SimpleJsonAdapter : SimpleJsonAdapter essentially returns a defined json. This can be very useful. For example, instead of hardcoding translation strings everywhere, you can create a JSON structure and have everything centralized.

Use Element TTL : We do tend to cache websites behind CDN networks, but it is nice to also use the Element TTL provided, in order to set caching of elements to improve performance.

Use hidden Sections / DDMs : Another tip is using hidden Sections and/or DDMs to store information that is difficult to handle sitemap wise. For example a Hidden Section to store Quotes, for that Quote Slider.

Bundle Architecture

In general Bundles should follow one of the following architectures based on project needs and scale:

- **Page Specific** - used mostly on smaller scale sites.
- **Central Control** - used mostly on playlist websites. Every module sits in a bundle.
- **Domain Specific** - multi domain accounts should use a domain prefix to help organize bundles.

Apart from the above breakdown, there should always be a “General” or “Global” bundle where stuff like js, css and general mark up (header, footer, head, etc) is stored.

Development Security Best Practices

Below there are a few Tips and Best Practices for **securing** code in Expresia and its Instance..

Captchas : Captchas must be present in any forms and services that call important functions on the page internally, those that generate mail or other external responses for example. They also must be implemented and validated correctly according to **Google Developer Captcha Documentation**

- https://developers.google.com/recaptcha/intro
    - https://www.google.com/recaptcha/api/siteverify

External Endpoints : With Expresia, we have the ability to make 'internal APIs', if we make use of this functionality, we must know that at some point in the interaction, the external user can track that the application 'called' the endpoint we've made, so any external API must be properly secured since, anyone can call it.

URLs with IDs and similar : We can pass parameters though the URL, but we must not transfer any delicate data or identification, since both can be used to access unauthorized sources, or act as information that an external attacker can use.

Input Validations (Frontend) : Any input that we made for an external user has to be correctly validated, without this validation, users can input code which once readed from within Expresia, provides a security hole for any user or administrator. The two most common verification would be to limit characters that are valid for Users, Text and emails, which shouldn't need characters like '<>' and to validate that only numbers are input with a valid length in phones and identifications. This also refers to strongly validating other inputs of other types, even files.

Generating Documents : When generating a document, or saving something in the server, remember that by default any file is saved from within the **/media** directory. This means that you should avoid saving delicate data to the server without correct configuration, since it will mean letting anyone externally see files with delicate data for any user, invoices, generated PDFs with validations, etc. If you need to generate or receive documents with important information, you should forward it to another subdirectory and protect it through the infrastructure.

Security Headers : It is recommended that any client contains a correct security headers configurator for any response.

Weak Credentials : The Expresia login XPR is available externally, and anyone can try to access. Which is why double authentication should be used along with a strong password when possible.

External Files with Server information : Files like **.htaccess** should be correctly protected and shouldn't be seen externally, if possible, protect these with a **CDN** or **WAF**, since they contain some configuration information.

Correct CDN-WAF Configuration : A correct configuration prevents the avoiding of the CDN or WAF, by only allowing these IP addresses in the infrastructure and blocking any direct connections that want to bypass CDN performance improvements or WAF protections.