Présentation

Le projet Sillon **peut être** une base de connaissances et de recommandations dans les choix de mise en place d'un projet au sein d'une administration française (mais pas que). Ce projet n'est ni sponsorisé par une administration ni par un incubateur de beta.gouv. Il est à l'heure actuelle une initiative cherchant à combler un manque ressenti.

Ce "guide" tend à faire un tour complet des concepts importants et à justifier le "pourquoi" telle ou telle technologie/méthode est mise en avant. Il y a beaucoup de notions concernant la technique mais nous tentons aussi d'aborder des aspects design et produit pour couvrir le panorama de la réalisation d'un outil numérique. Ce guide devrait couvrir les besoins de **90%** des projets. L'important est de garder un regard critique au vu du projet que l'on a à réaliser, et qu'en cas de doute, l'on se rapproche de ses confrères. *Par exemple vous aurez sûrement des prérequis différents si vous relevez du "secret défense".*

Notez que de manière générale les choix sont influencés par les effets de mode (hype & marketing), la communauté sur les outils, les retours d'expérience… Le bon

choix d'aujourd'hui pour votre projet sera peut-être regretté demain. Il est normal d'avoir de la dette au sens large, il ne faut juste pas l'ignorer.

Bonne lecture 🙂 ! *(une version complète PDF est disponible dans le menu du haut)*

> **NOM DU PROJET**
>
> Le sillon est une rigole faite pour préparer la terre afin d'y semer des graines facilement. Un sillon peut être rectiligne, courbé, écourté… et il est parmi plein d'autres sillons. Tout cela pour dire que ce guide présente justement "un sillon", mais que pour faire "pousser" votre projet vous êtes libre d'en choisir totalement ou partiellement un autre 🌱.

Les contributeurs à ce guide ne sauraient être tenus pour responsables d'un mauvais choix dans votre projet 😀.

Préambule

Validité de ce guide

Nous avons rassemblé dans ce guide la façon dont nous voyons la vie d'un produit. Même si cela est basé sur des expériences diverses, cela reste subjectif. Nous ne prê…

À chacun son expertise

Dans les chapitres qui suivent, il faut toujours garder en tête que les étapes décrites sont à faire par la personne de l'équipe la plus appropriée.

Nous ne fournissons pas une stack technique "code en mains"

Pour la partie technique s'est posée la question de fournir ou non une stack de code (comme un template) à copier/coller très rapidement pour démarrer.

Le lexique

Pour chaque mot ou acronyme qui sort un peu de l'ordinaire nous essayons d'y apporter du contexte voire d'y joindre un lien explicatif. Pour rendre la lecture digeste (v…

Validité de ce guide

Nous avons rassemblé dans ce guide la façon dont nous voyons la vie d'un produit. Même si cela est basé sur des expériences diverses, cela reste subjectif. Nous ne prêchons pas la sainte parole.

De plus, il est important de remettre cela dans votre contexte. **Piochez les informations qui vous semblent pertinentes, mais ne court-circuitez pas les équipes transverses existantes au sein de votre entité (design, marketing, technique…).** Pourquoi ? Car avoir un produit "dissident" du reste est compliqué à reprendre le jour où vous partirez, et vous risqueriez au passage de vous faire des ennemis.

Si votre entité fait différemment, ils doivent avoir leurs raisons. Et si vous estimez qu'ils doivent changer quelque chose, libre à vous de les "challenger".

L'idée du guide est d'apporter la solution la plus simple à un projet qui n'aurait pas trop de ressources à disposition, ou de cadre technique imposé.

> **ATTENTION**
>
> Comme cela reste subjectif, il est possible que ce guide vous irrite car mettant en avant des choses que vous n'approuvez pas 😣😣🤬. Essayez alors d'y repenser avec la perspective "faire simple avec peu de ressources" pour relativiser… Et si vous pensez encore que l'on est dans le faux, n'hésitez pas à nous proposer des améliorations en les partageant via la messagerie.

À chacun son expertise

Dans les chapitres qui suivent, il faut toujours garder en tête que les étapes décrites sont à faire par la personne de l'équipe la plus appropriée.

Si cette personne n'existe pas et qu'il y a d'autres équipes dans votre entité, n'hésitez pas à leur demander conseil voire à leur demander de passer un peu de temps à vous aider. Même si votre organisation est en "silos", n'hésitez pas car ce temps de partage transverse sera précieux pour vous et votre entité.

Dans le cas où vous êtes le porteur de projet et que c'est assez nouveau pour vous, nous vous conseillons de trouver un mentor qui a déjà entrepris des projets "de bout en bout". Il est sain de prévoir des échanges réguliers avec cette personne pour parler de vos problématiques et vos choix stratégiques.

Nous ne fournissons pas une stack technique "code en mains"

Pour la partie technique s'est posée la question de fournir ou non une stack de code (comme un template) à copier/coller très rapidement pour démarrer.

Sachez qu'il existe déjà des stacks de certaines administrations en faisant le parti pris de telles ou telles technologies. Notre retour d'expérience concernant cette pratique est un peu particulier puisque pour vous faciliter la vie il faudrait brancher tous les concepts présentés ci-après afin que ça marche "out-of-the-box". Mais dans les faits, quand on commence un projet nous avons besoin du strict minimum… Pire encore, chaque développeur a sa propre expérience et ses propres choix de développement.

Sauf que quand vous devez nettoyer 80% des concepts dans une stack, vous vous retrouvez avec un découpage qui n'est peut-être pas le plus judicieux, et avec lequel vous ne vous sentez pas forcément à l'aise (vous héritez de pratiques étrangères). On vous recommande donc de créer vous-mêmes votre stack en ajoutant brique par brique (même si ça prend quelques heures ou quelques jours au total). Ainsi, **votre stack** vous resservira pour d'autres projets !

> **CONTEXTE TECHNIQUE**
>
> Dans la suite pour le développement d'un produit, nous allons mettre en avant et argumenter la stack suivante :

- Frontend : applicatif web ;
- Backend : Node.js ;
- Langages : JavaScript / TypeScript ;
- Frameworks : React & Next.js ;
- Intégration responsive du frontend ;
- Base de données RDBMS (PostgreSQL) ;
- ORM de la base de données : Prisma ;
- Communication client-serveur : tRPC (comme du GraphQL mais "nativement" avec des schémas TypeScript) ;
- UI : DSFR (Design System FRançais) ;
- Hébergeur simple et souverain.

Gardez en tête que même si vous utilisez des technologies différentes, la lecture de ce guide peut vous être bénéfique.

INFO

Si vraiment vous tenez à partir d'un code existant, lisez le chapitre mentionnant code.gouv.fr afin de retrouver des projets publics utilisant les technologies que vous désirez. Ensuite regarder leur rendu "live" ainsi que l'organisation du code pour voir si cela vous sied.

(Le guide n'a actuellement pas pour objectif de référencer des "projets exemples". Cela évoluera peut-être si c'est une demande répétée de votre part)

Le lexique

Pour chaque mot ou acronyme qui sort un peu de l'ordinaire nous essayons d'y apporter du contexte voire d'y joindre un lien explicatif. Pour rendre la lecture digeste (vu que Sillon est un peu long 😄), nous avons fait ce travail de détails seulement sur leur première occurrence.

Si vous lisez les chapitres dans l'ordre, aucun problème… Sinon n'hésitez pas à utiliser la fonction de recherche pour remonter aux introductions de certains concepts.

Amorce du projet

Réfléchir aux coûts liés au temps

Outils métiers

Est-ce que votre projet est inédit ?

Il est important de prendre quelques heures voire quelques jours pour sonder le marché afin d'être sûr qu'une solution existante n'existe pas déjà :

S'assurer de répondre à un besoin

Avant même de commencer à recruter des personnes sur le projet, et bien que la "user research" (phase d'investigation) soit un domaine à part entière, en tant qu'initi...

L'investigation, ça se travaille

Il faut garder à l'esprit que la phase d'investigation est une tâche de fond. Plus vous avancerez sur le projet et plus vous aurez d'éléments pour ajuster la direction du pr...

Attention ! Difficile d'aller plus loin sans budget

Ça y est, deuxième barrière qui peut mettre en péril le projet 😨... La bonne nouvelle c'est que jusqu'ici l'investigation peut normalement se faire en glanant quelques h...

Réfléchir aux coûts liés au temps

Outils métiers

Cela peut paraître bizarre de commencer le guide par la thématique des coûts, mais c'est une notion essentielle qui vous permettra d'arbitrer de manière lucide les choix à chaque étape de votre projet.

En imaginant que j'ai besoin d'un produit numérique pour faciliter le quotidien d'agents publics qui gèrent des demandes de médiation, et que je me retrouve dans la situation où il existe déjà une solution française sur le marché qui répond à mon besoin. J'ai le choix :

1. Soit je prends la solution tierce, et elle va me coûter 1000€ par an ;
2. Soit je développe mon propre outil en interne et le coût d'hébergement sera de 30€ par an.

Il n'y a pas de réponse parfaite, mais il faut se méfier de la (2). Le fait de prendre l'initiative de faire soi-même induit forcément de mobiliser du temps, il est donc important de définir à grosses mailles quel est le coût de ce temps. Normalement cela va toucher les domaines suivants :

- Idéalement juste pour la phase de développement :
  - Le légal ;
  - La gestion de projet ;
  - Le développement informatique (+ audit de sécurité) ;

    - Le design ;
- Sur le long terme :
    - Le support utilisateur ;
    - Le support technique (mises à jour des librairies ou des serveurs à cause des failles de sécurité découvertes, correction de bugs…).

Sans même savoir combien de jours ou mois cela me prendrait de développer l'outil, je sais que j'aurai "sur le long terme" du temps à mobiliser en interne (même si ce n'est que ponctuellement). Si on imagine que le coût toutes charges comprises d'un ingénieur informatique (qu'il soit agent public ou prestataire externe) est de 350€ par jour minimum, et de même pour quelqu'un du support utilisateur ; en 3 jours travaillés sur l'année on se retrouve déjà dans l'ordre de grandeur du coût de la solution existante.

Sachant qu'un développement de projet numérique prend rarement moins de plusieurs mois… Le coût de développement peut devenir assez vite important, et vous perdez aussi du temps avant de répondre réellement à votre besoin. Il peut donc être stratégique de partir avec la solution existante si elle répond à votre besoin.

L'idée n'est pas ici de dire qu'il faut éviter de développer en interne. Au contraire, il faut juste y apporter un regard critique et juger à court, moyen et long terme si cela se justifie.

Outils fonctionnels

Dans l'exemple ci-dessus on parle d'un outil "métier", où il peut être judicieux de le développer en interne puisque c'est notre cœur de métier et nous avons des besoins très spécifiques. Par contre quand il s'agit d'outils "fonctionnels" comme pour :

- Héberger un site internet ;
- Avoir un outil de design collaboratif ;
- Utiliser de l'intégration continue (CI/CD).

Il paraît peu légitime de gérer nous-mêmes des serveurs physiques pour déployer ces outils, sachant tous les coûts de maintenance associés, plutôt que de se reposer sur des solutions pour grand public quand elles existent. Sinon vous prenez le risque de concentrer les efforts de votre entité sur des choses qui ne font pas partie de sa mission.

Est-ce que votre projet est inédit ?

Il est important de prendre quelques heures voire quelques jours pour sonder le marché afin d'être sûr qu'une solution existante n'existe pas déjà :

- Il faut être vigilant qu'un service public similaire n'existe pas déjà au sein d'une autre administration. Si tel est le cas, évitez de faire doublon et rentrez directement en communication avec leur équipe produit pour réutiliser le leur ou essayer de mutualiser le besoin ;
- Dans le cas où c'est un service privé, jauger si ça vaut vraiment le coup de développer une solution en interne comme expliqué dans les coûts liés au temps. C'est un arbitrage très souvent lié à vos spécificités métiers non présentes dans l'outil.

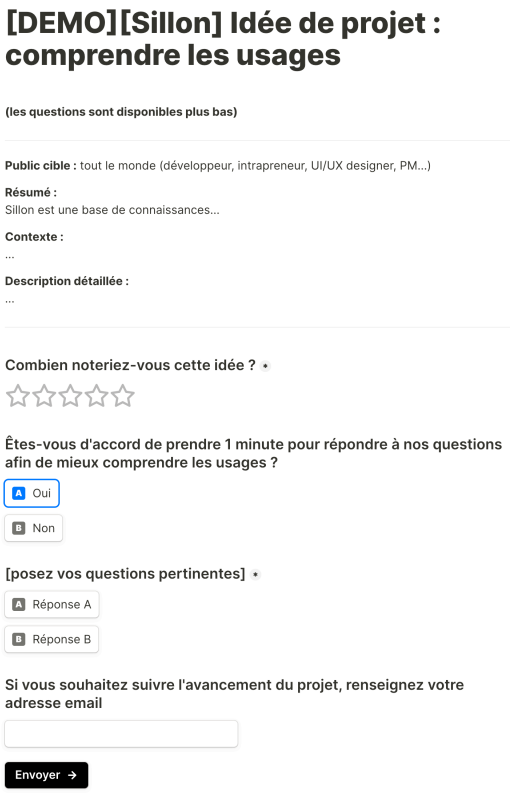
S'assurer de répondre à un besoin

Avant même de commencer à recruter des personnes sur le projet, et bien que la "user research" (phase d'investigation) soit un domaine à part entière, en tant qu'initiateur du projet vous devez avoir la confirmation de potentiels utilisateurs finaux que le projet leur apportera quelque chose. Même si l'erreur est permise, il faut éviter de réaliser un projet qui ne correspond pas à votre public cible.

Il faut absolument **identifier quelques acteurs** qui correspondent au profil type d'un utilisateur final de votre idée de projet. Il faut **co-construire avec eux** afin de voir ce qui va et ce qui ne va pas, c'est ce qu'on appelle la maîtrise d'usage. L'important dans toute cette phase d'investigation est de documenter tous les retours que vous avez. Ces notes vous permettront de quantifier l'intérêt et d'apporter un historique au projet pour retrouver pourquoi les choix ont été faits de telles manières.

Le plus simple pour les interviewer :

- Soit de manière libre aux cours de discussions ;
- Soit de manière structurée en faisant un sondage. Cela peut aider à récolter beaucoup d'avis avec le moindre effort, vous devez juste obtenir une liste de contacts. Des outils comme Tally ou Google Form permettent cela.



Exemple d'un formulaire fait avec Tally (*source*)

À vous de juger la taille critique afin de tirer des conclusions de votre échantillon de réponses. Pour donner un ordre d'idée, si votre projet vise des particuliers, essayez d'obtenir au minimum une trentaine de retours, et si vous visez des personnes morales (entreprises, collectivités...) essayez d'en avoir une dizaine.

Quelques exemples de questions ci-dessous :

- Suite à la présentation du projet, combien le noteriez-vous de 1 à 5 ?
- Seriez-vous prêt à l'utiliser ?
- Si "oui", combien seriez-vous prêt à payer (coût fixe, abonnement...) ? *(cette question est dans le cas où vous devez être rentables pour financer la pérennité du projet)*
- Partagez ouvertement votre avis sur le projet (critiques, suggestions)
- Souhaiteriez-vous faire partie de nos "beta-testeurs" ?
- Renseignez votre adresse email, prénom, nom si vous souhaitez suivre

l'avancement du projet

Mais gardez en tête qu'il faut y ajouter des questions spécifiques au projet (des questions "métiers") pour confirmer vos idées.

L'analyse des réponses devrait vous aider à conclure :

1. L'idée correspond parfaitement aux attentes ;
2. Je dois ajuster un peu l'idée, et "challenger" de nouveau celle-ci ;
3. Je suis à côté de la plaque, il n'y a aucun besoin utilisateur pour ce projet.

La conclusion (3) est difficile à admettre, pourtant il faut à tout prix arrêter l'initiative du projet. Le risque sinon est de se convaincre soi-même qu'un besoin existe, d'ignorer les retours du sondage, voire même d'essayer de créer un besoin chez les utilisateurs en tentant de les convaincre.

> **DANGER**
>
> **L'idée d'un projet a de grandes chances de répondre à un besoin si elle émane d'un potentiel utilisateur.** Donc restez vigilant aux demandes de réalisation d'un projet qui sortent de nulle part…
>
> **Notez aussi que si aucun des interviewés n'est prêt à être beta-testeur, c'est plutôt mauvais signe.** Cela peut vouloir dire que la plus-value de votre projet ne va pas transcender leur quotidien, ou que personne n'a de disponibilité pour faire le suivi (dans ce cas, essayez de trouver de nouveaux interviewés).

L'investigation, ça se travaille

Il faut garder à l'esprit que la phase d'investigation est une tâche de fond. Plus vous avancerez sur le projet et plus vous aurez d'éléments pour ajuster la direction du projet afin de mieux répondre au besoin. Il n'est pas étonnant de **pivoter** au cours d'un projet, ni de se rendre compte a posteriori que le projet n'a pas de raison d'être.

Ces signaux indispensables apparaissent si vous travaillez avec des "beta-testeurs" et que vous ne restez pas dans un tunnel seul avec votre équipe. *À proprement parler nous sommes loin d'une quelconque version "beta", pourtant c'est parmi les personnes qui ont participé au sondage qu'il faut que vous trouviez de réels partenaires pour toute la durée de développement du projet*.

L'autre bénéfice d'avoir des partenaires est qu'ils vous feront économiser énormément de temps. Pour chaque fonctionnalité susceptible d'être ajoutée, ils seront là pour communiquer leur avis (sachant que de base ces fonctionnalités devraient être définies par le besoin des utilisateurs, leur besoin).

Pour faire connexion avec le chapitre sur les coûts liés au temps, il faut garder en tête la gradation suivante qui définit aussi le temps nécessaire pour chaque étape d'un produit (du plus court au plus long) :

1. Émettre une idée (quelques secondes) ;
2. Spécifier des cas d'utilisations (quelques heures) ;
3. Faire des maquettes pour tester l'appétence (quelques jours) ;
4. Développer informatiquement (quelques semaines voire mois) ;

5. Maintenance et évolutivité (quelques années).

C'est évident que si vous allez en phase de développement en ayant négligé la phase d'investigation, vous prenez un gros risque pour la continuité du projet et le moral de votre équipe. Il vaut mieux savoir dès le (1) que l'on doit pivoter ou arrêter, plutôt que quand on est rendu au (4).

Il est donc important de garder les beta-testeurs motivés car en vous aidant ils donnent de leur temps. La moindre des choses est donc :

- De les tenir à jour de l'avancement du projet ;
- De leur offrir une réduction ou gratuité du service si celui-ci est payant ;
- De considérer leurs demandes d'évolutions même si elles ne font pas l'unanimité.

> **REMARQUE**
>
> Si vous travaillez en équipe, vous pouvez suivre la méthodologie Agile, ou même faire du pseudo-Agile… l'important est de vous sentir à l'aise sans que l'équipe perçoive cela comme une contrainte. Comme à ce stade votre projet a un avenir incertain, privilégier la "productivité" aux procédures peut accélérer votre investigation.
>
> L'important est de garder à l'esprit que l'on parle toujours sous forme de **cycles itératifs** (à chaque nouvelle proposition ou fonctionnalité il faut essayer de repasser des validations).

Attention ! Difficile d'aller plus loin sans budget

Ça y est, deuxième barrière qui peut mettre en péril le projet 😅... La bonne nouvelle c'est que jusqu'ici l'investigation peut normalement se faire en glanant quelques heures par-ci par-là sans empiéter sur vos tâches du quotidien. Mais pour la suite, il va falloir monopoliser plus de temps, d'énergie, afin de faire de ce projet une réalité.

Les conditions actuelles sont peu propices à la réussite du projet (sauf s'il est tout petit). Il va falloir que votre entité vous libère du temps pour vous y consacrer, voire même prévoir un budget pour recruter une équipe si nécessaire.

La bonne nouvelle c'est que vous avez prouvé que vous répondiez à un besoin. Votre projet a donc une utilité publique, cela devrait aider à ce que vous convainquiez votre hiérarchie.

Si vous peinez à officialiser une suite à votre projet et que vous restez persuadé qu'il aurait de l'impact, sachez que l'État a créé une entité à part entière nommée beta.gouv. Elle répond à des besoins qui ne peuvent être portés ou solutionnés par l'administration concernée. Contactez-les afin d'avoir plus d'informations. Concrètement si l'idée est retenue vous serez guidé et accompagné à toutes les étapes du projet (ils mettent à disposition des personnes ainsi que des ressources techniques).

AVIS AUX MOTIVÉS

Si vous êtes expert dans un domaine du numérique et que vous souhaitez contribuer à des projets publics, regardez les offres beta.gouv ainsi que le

concept d'entrepreneurs d'intérêt général (EIG).

Prototypage

Apporter les compétences manquantes

Durant cette phase vous allez maquetter votre produit afin de poser "les directives" quant à la réalisation finale du produit. Comme mentionné dans le chapitre investi...

Cadrer le projet

Là encore on va s'épauler de nos beta-testeurs. On peut penser savoir sur le bout des doigts à quoi la plateforme doit ressembler, mais il est important d'aller à la renco...

L'importance du maquettage

Vous arrivez à un point où beaucoup de choses abstraites ont été validées, cela semble prometteur. Il faut maintenant avoir une base de travail commune qui soit visue...

Prototype technique ou pas ?

Nous aurions tendance à dire non.

Apporter les compétences manquantes

Durant cette phase vous allez maquetter votre produit afin de poser "les directives" quant à la réalisation finale du produit. Comme mentionné dans le chapitre investigation, cette phase de prototypage est un peu plus coûteuse en temps et peut avoir de lourdes conséquences. Il est important de rallier à votre projet les compétences de "designer".

1. Le "UX designer" (user experience designer) dans notre cas est spécialisé en ergonomie d'application numérique pour que l'utilisation soit limpide et efficace. (*Il n'y a rien de pire que de ne pas savoir où chercher quand on est sur un site...)*
2. Le "UI designer" (user interface designer) va apporter la touche graphique à vos maquettes afin d'y créer une certaine harmonie pour rendre le produit attractif et répondre à certaines normes du numérique. *On verra plus loin qu'il est acteur de la mise en place de l'accessibilité au sein du produit, qui est légalement requis pour les sites publics.*

Il est important de garder en tête que la priorité à ce stade est le "UX designer", c'est lui qui va faire que votre produit sera utilisable sans s'arracher les cheveux. Même si le produit est un peu moche, on cherche dans un premier temps à ce qu'il réponde au besoin efficacement. Vous verrez d'ailleurs que l'État met à disposition un standard graphique afin d'harmoniser et accélérer la conception graphique des produits (voir la section sur le DSFR).

Cadrer le projet

Là encore on va s'épauler de nos beta-testeurs. On peut penser savoir sur le bout des doigts à quoi la plateforme doit ressembler, mais il est important d'aller à la rencontre des beta-testeurs afin de s'accorder tous ensemble sur un panorama du produit :

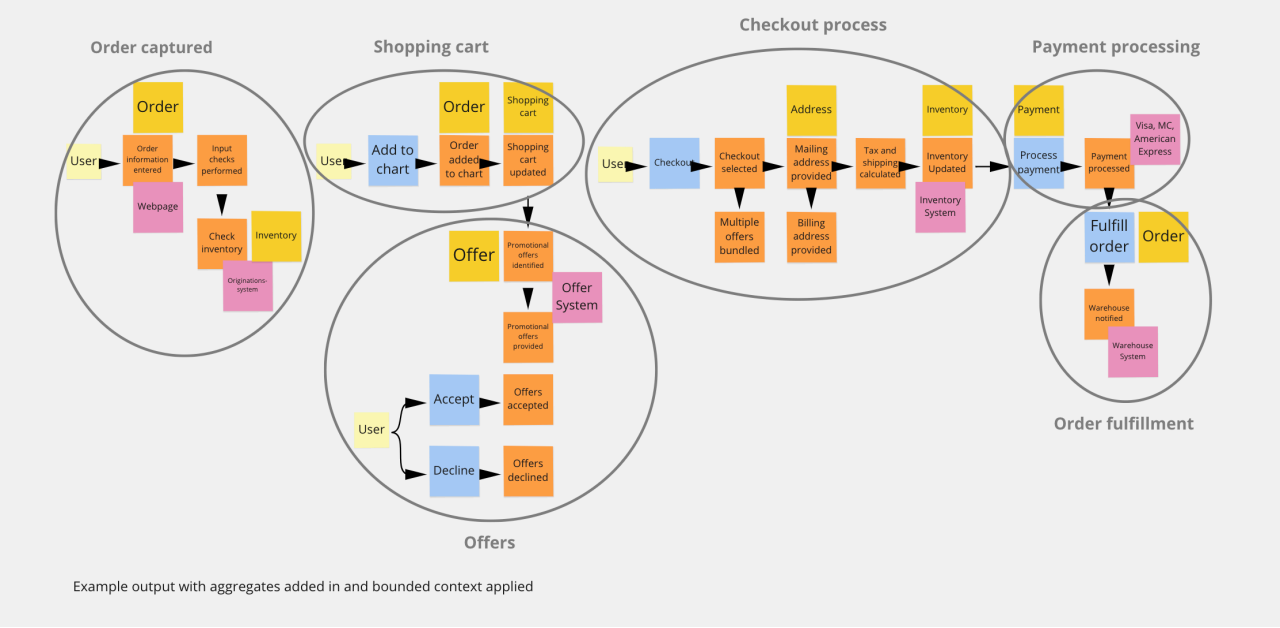
- Chaque type d'utilisateur au sein du service ;
- Les entités externes dont dépend le service ;
- Les cas d'utilisation (pouvoir s'inscrire, pouvoir faire une demande de médiation…).

Dites-vous juste qu'il faut réussir à définir tous les flows possibles au sein de votre produit numérique, et arriver à un consensus avec vos beta-testeurs. Cela permet d'identifier les points bloquants et aussi de tous s'accorder sur le vocabulaire (il est possible qu'un même mot désigne plusieurs choses au sein de contextes différents dans l'applicatif, c'est le moment de clarifier tout cela).

Il y a plusieurs concepts et méthodes pour structurer ce type d'atelier, quelques exemples:

- Atelier de sketching ;
- Atelier d'event storming ;
- …

Cela peut être fait sur post-it et pris en photos à la fin puis mis à disposition, ou virtuellement avec des outils comme FigJam, Miro ou autre. Demandez conseil autour de vous si besoin pour animer ce type d'atelier.



Résultat illustratif d'un atelier pour mieux cerner le produit (*source*)

Si votre produit semble très conséquent après l'atelier, il va être nécessaire de privilégier les fonctionnalités indispensables pour rendre cela digeste afin d'en développer une version minimale. Pour autant, gardez tout le contenu établi car il vous sera utile pour travailler ces potentielles futures fonctionnalités.

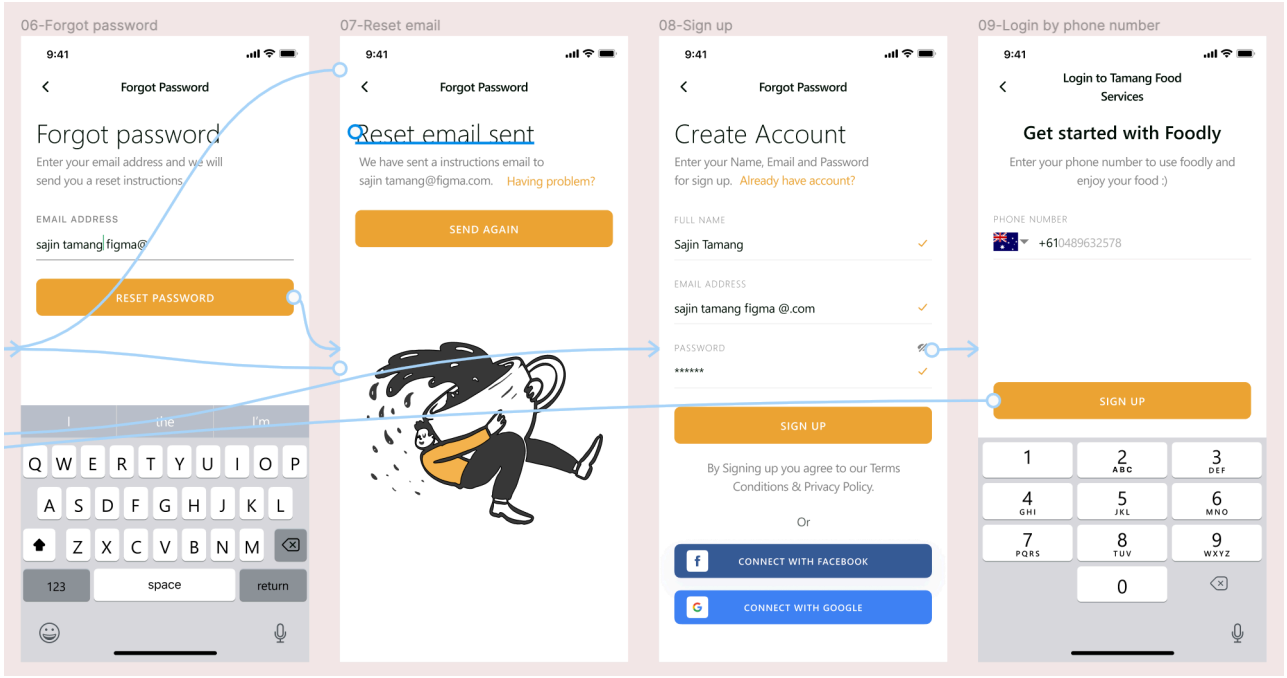
L'importance du maquettage

Vous arrivez à un point où beaucoup de choses abstraites ont été validées, cela semble prometteur. Il faut maintenant avoir une base de travail commune qui soit visuelle afin de faciliter le futur travail de réalisation.

Nous vous recommandons d'utiliser des outils comme Figma ou Penpot pour faciliter le travail et la collaboration. L'idée dans cet outil sera de :

1. Créer tous les écrans qui existent dans votre applicatif ;
2. Créer des liens entre chaque écran afin de comprendre les flows possibles (cela servira pour faire des "wireframes animés", c'est-à-dire des pseudo-démos) ;
3. Partager les maquettes aux beta-testeurs en leur demandant de refaire une passe sur tous les éléments afin de confirmer qu'ils sont d'accord avec ce qui devrait être développé.

C'est un moment crucial où il faut bien faire comprendre aux beta-testeurs que c'est LEUR moment. Et que s'ils ne prennent pas le temps d'exprimer leurs suggestions maintenant… Cela fera perdre pas mal de temps au projet.



Exemple d'un Figma d'application où on peut utiliser le wireframe animé (*source*)

> **RAPPEL**
>
> Toujours dans cette notion de validation en cycles, si vous aviez maquetté des écrans grossièrement pour valider la partie "UX design", il est important une fois le "UI design" mis en place de le repartager avec vos beta-testeurs.

Prototype technique ou pas ?

Nous aurions tendance à dire **non**.

Pourquoi ?

1. Le but du prototype technique est de rapidement mettre quelque chose dans les mains des beta-testeurs. On prend donc normalement énormément de raccourcis que ce soit sur la gestion de données, sa validation des données, la sécurité… afin de ne pas mettre sur pied un début d'usine à gaz qui ressemblerait au vrai produit. En faisant ça :
   i. On risque de ne pas représenter les vrais usages du produit, et donc il est peu probable que les gens aillent tester régulièrement un outil qui est partiellement stable ;
   ii. Il faudra dans tous les cas faire table rase de ce prototype car il a été construit sur des fondations précipitées ;
2. Nous considérons qu'il est plus efficace d'avoir bien fignolé ses maquettes et de disposer d'un "wireframe animé" afin de valider la cohérence du produit avec vos beta-testeurs. Cela ne demande aucune compétence technique, vous n'avez donc pas passé de temps à essayer de mettre en place des outils comme un faux frontend branché à une "base de données" tel que Grist ou Airtable…

Si les wireframes conviennent aux beta-testeurs vous pouvez passer à la réalisation d'un MVP (produit minimum viable). Vous vous lancez donc dans la réalisation du vrai produit, tout en limitant le développement initial qu'à certaines fonctionnalités.

⚠ CONTEXTE QUI SUIT

Nous considérons ici et dans la suite du guide que vous partez sur un produit numérique à construire. Les maquettes et MVP n'ont quasiment aucun sens si vous réutilisez un outil déjà fait sans le modifier (il devient automatiquement votre "produit").

Développement du produit

Éviter l'overengineering, et la hype

En parlant d'efficacité… l'overengineering c'est le fait de sur-complexifier le développement technique. Pourquoi faire simple quand on peut faire compliqué ? Parlez-en…

Produit minimum viable

Les spécifications du MVP découlent directement des phases de cadrage. L'idée est de se limiter en termes de fonctionnalités à réaliser dans un premier temps. Il faut e…

No code / low code… ou pas ?

L'idée du "no code" est de pouvoir réaliser son produit technique sans quasiment écrire de code et donc pouvoir se passer de développeurs.

Démarrer le développement technique

Pas de mystère il vous faut un développeur. Comme nous rentrons dans la phase la plus coûteuse en temps et en budget il est important que votre premier développeu…

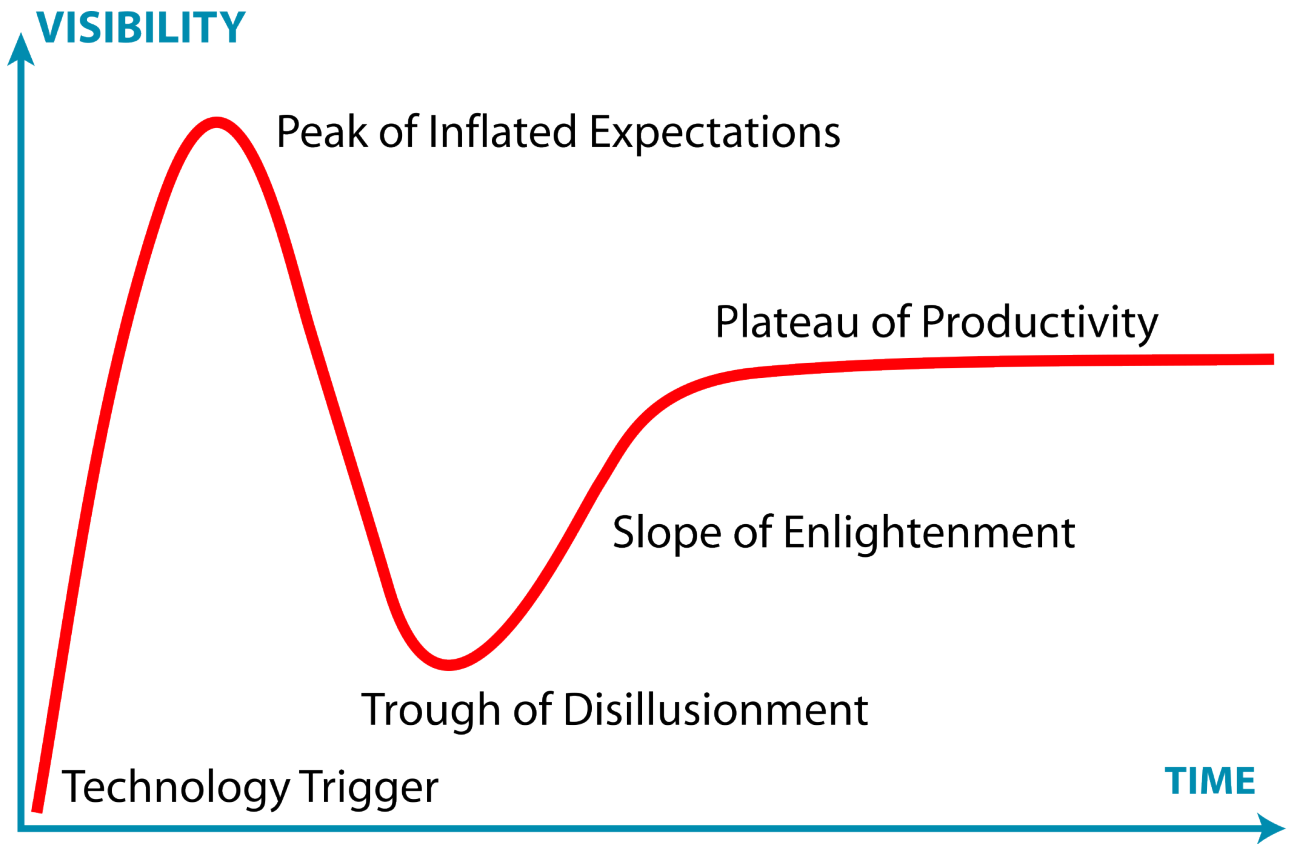
Ne pas mettre le développeur "à la cave"

C'est une erreur assez commune de seulement donner des directives à l'équipe technique et de ne pas en attendre plus. Pourtant ils ont du jus de cerveau à disposition…

Éviter l'overengineering, et la hype

En parlant d'efficacité... l'overengineering c'est le fait de sur-complexifier le développement technique. Pourquoi faire simple quand on peut faire compliqué ? Parlez-en aux développeurs, ils sauront de quoi il est question.

Loin de nous l'idée de critiquer l'innovation... Le problème c'est qu'il y a à chaque fois un effet de mode (la "hype") où tout le monde ne fait plus que parler de ça. Cela dure un temps, puis cela s'estompe. On peut parler du NoSQL, de la blockchain, des microservices... ça répond à des besoins très spécifiques mais on se retrouve submergé de conférences où on pourrait croire que c'est la voix à suivre par défaut.



(*source*)

Cette innovation peut très bien être "productive" si elle est adaptée à votre produit. Mais dans les faits, une majorité des produits numériques d'aujourd'hui ne demande pas beaucoup plus de complexité que ceux d'il y a 15 ans. Un monolithe (backend + frontend), une base de données, et hop ça fonctionne. *Bien évidemment on a beaucoup automatisé et standardisé le déploiement, les tests… mais l'essentiel des concepts est identique.*

La sur-complexité est encombrante sur le long terme :

- L'équipe technique n'a plus envie de mettre les mains dedans, au risque de faire bugguer quelque chose, ou tout simplement car le rapport `temps travaillé / résultat` n'est plus gratifiant ;
- Les développeurs pionniers de cette complexité dans le projet deviennent indispensables pour continuer le développement ou maintenir l'outil. C'est un vrai problème car votre produit va idéalement vivre des années, mais ces

personnes peuvent partir à tout instant… Bon courage aux nouvelles recrues !

FALLAIT Y PENSER !

En adoptant une stack technique simple et minimaliste, vous êtes déjà indirectement en train de répondre à certaines problématiques d'écoconception (oui oui 😏🌱).

Produit minimum viable

Les spécifications du MVP découlent directement des phases de cadrage. L'idée est de se limiter en termes de fonctionnalités à réaliser dans un premier temps. Il faut en garder assez pour répondre au principal besoin et pas trop non plus pour pouvoir rapidement sortir quelque chose.

Si nous partions directement sur le produit complet, il y aurait un risque de surcharge cognitive en voulant tout gérer, et implicitement rentrer dans un long tunnel où rien ne sort. Le problème dans ce cas est que si vous sortez votre produit 1 an et demi après :

- Est-ce que le besoin n'a pas changé ?
- Est-ce que vos beta-testeurs sont toujours motivés ?
- Est-ce que le besoin existe toujours ? *(il peut disparaître voire être comblé par une autre initiative de projet)*

Le fait d'itérer rapidement permet aussi de garder l'équipe motivée, les "quick wins" (victoires faciles) sont répartis tout au long du périple.

> **REMARQUE**
>
> On ne le détaillera pas mais l'enchaînement avant la "General Availability" (la mise à disposition réelle du produit au public) n'a dans ce guide pas de frontières franches entre : MVP, version alpha, version beta.

No code / low code… ou pas ?

L'idée du "no code" est de pouvoir réaliser son produit technique sans quasiment écrire de code et donc pouvoir se passer de développeurs.

Nous ne sommes pas assez expérimentés sur ce domaine pour vous dire que tel outil pourrait vous aider à réaliser tel ou tel type de produit (tout en respectant tous les aspects légaux nécessaires d'un projet public). Par contre selon nous, le *no code* a forcément à un moment une limite où la fonctionnalité n'a pas été prévue. C'est souvent à ce moment-là qu'il faut réfléchir à passer à du code (avec le risque de repartir de zéro sauf si l'outil *no code* dont vous êtes tributaire permet une assez grande flexibilité pour être "étendu" par du code).

Nous verrions mieux le nocode dans la réalisation de prototype, mais nous avons fait le choix dans ce guide de déconseiller cette partie et de seulement utiliser des "wireframes animés". Peut-être que le *no code* est la voie à adopter si :

- Votre projet est très très simple et doit le rester ;
- Votre projet n'a pas de budget pour un développeur ;
- Vous avez une deadline trop courte et vous avez repéré un bon outil nocode.

Démarrer le développement technique

Pas de mystère il vous faut un développeur. Comme nous rentrons dans la phase la plus coûteuse en temps et en budget il est important que votre premier développeur ait déjà réalisé des projets informatiques dans le passé car il va poser les bases du projet.

Idéalement, on recherche le mouton à 5 pattes afin de limiter la dette technique... il y a tellement de domaines à prendre en compte techniquement :

- Le frontend ;
- Le backend ;
- Le déploiement ;
- La sécurité ;
- L'accessibilité ;
- La maintenance ;
- ...

La complexité et l'évolutivité des technologies sont telles que même un passionné peut se perdre dans sa passion et ne pas aller droit au but : celui de délivrer un produit.

Nous vous recommandons :

1. Pour les entretiens de faire appel à 1 ou 2 autres personnes techniques expérimentées qui développent toujours afin de bien tester le profil du candidat ;

2. Au moment où le recruté proposera l'architecture technique globale sur laquelle il souhaite partir, de la faire valider par :
    - Les mêmes personnes techniques qui ont aidé à l'entretien ;
    - Votre responsable de la sécurité des systèmes d'information (RSSI) ;
    - Votre délégué à la protection des données (DPO).

Le but n'est pas d'instaurer une méfiance, mais la technique reste une nébuleuse pour beaucoup de personnes car c'est très abstrait et qu'il y a une pléthore de possibilités pour faire exactement la même chose (plus ou moins efficacement, et avec plus ou moins de risques).

> **INFO**
>
> La suite du guide recoupera assez souvent avec des concepts techniques. Cette disproportion dans le guide est sûrement un bais des contributeurs initiaux… Désolés 😉 ! Cela évoluera probablement avec le temps…

Ne pas mettre le développeur "à la cave"

C'est une erreur assez commune de seulement donner des directives à l'équipe technique et de ne pas en attendre plus. Pourtant ils ont du jus de cerveau à disposition et plein de bonnes idées, profitez-en !

Pourquoi disons-nous ça ? Idéalement tout est spécifié dans les maquettes et le développeur n'a plus qu'à faire une sorte de copier-coller durant l'implémentation pour rendre cela vivant. Dans les faits, cela demande énormément de travail de maquetter toutes les combinaisons possibles de l'applicatif (d'ailleurs on évite de le faire pour garder les maquettes compréhensibles). Le développeur va donc se retrouver à arbitrer certains choix afin d'avancer. Le fait qu'il ait été un minimum impliqué dans les échanges de spécifications avec le reste de l'équipe et les beta-testeurs lui permet de s'approprier les attendus.

- Il sera amené à faire des retours au UX/UI designer quant aux contraintes techniques qui l'empêchent de réaliser certains aspects de la maquette ou cas d'utilisation, donc autant le savoir dès le début ;
- On peut ainsi enlever quelques intermédiaires en mettant en relation les beta-testeurs et l'équipe technique. Comme ça si une erreur technique apparaît, pas besoin de faire le téléphone arabe, on parle directement à la personne concernée.

Documenter votre projet

Au même titre que les spécifications, les aspects techniques doivent être décrits et historisés. Cela facilite l'embarquement de nouvelles recrues, et cela vous sauvera quand dans 1 an vous vous direz "mince, c'est organisé comment ? Et pourquoi ?".

Le code

Éternel débat sur les commentaires de code, ni trop, ni pas assez. Normalement le nom de fonction suffit à comprendre ce que ça fait, mais pour comprendre certains ...

La macro-technique

Il est déconseillé de documenter votre projet technique en dehors de votre repository Git. Le simple fait de mettre une distance entre votre code et sa propre docume...

Le schéma de base de données

Nous privilégions l'utilisation d'un ORM pour représenter notre schéma de base de données, comme cela tout comme le code, le schéma est versionné.

Nommage des commits

Avoir le versioning Git est utile, mais c'est encore mieux si les commits ont une description explicite. Évitez les simples aaa ou fix, essayez d'être plus verbeux en restant...

Le code

Éternel débat sur les commentaires de code, ni trop, ni pas assez. Normalement le nom de fonction suffit à comprendre ce que ça fait, mais pour comprendre certains passages à l'intérieur il peut être bon de clarifier ce que ça fait, ou pourquoi vous l'avez fait (est-ce un workaround ? …).

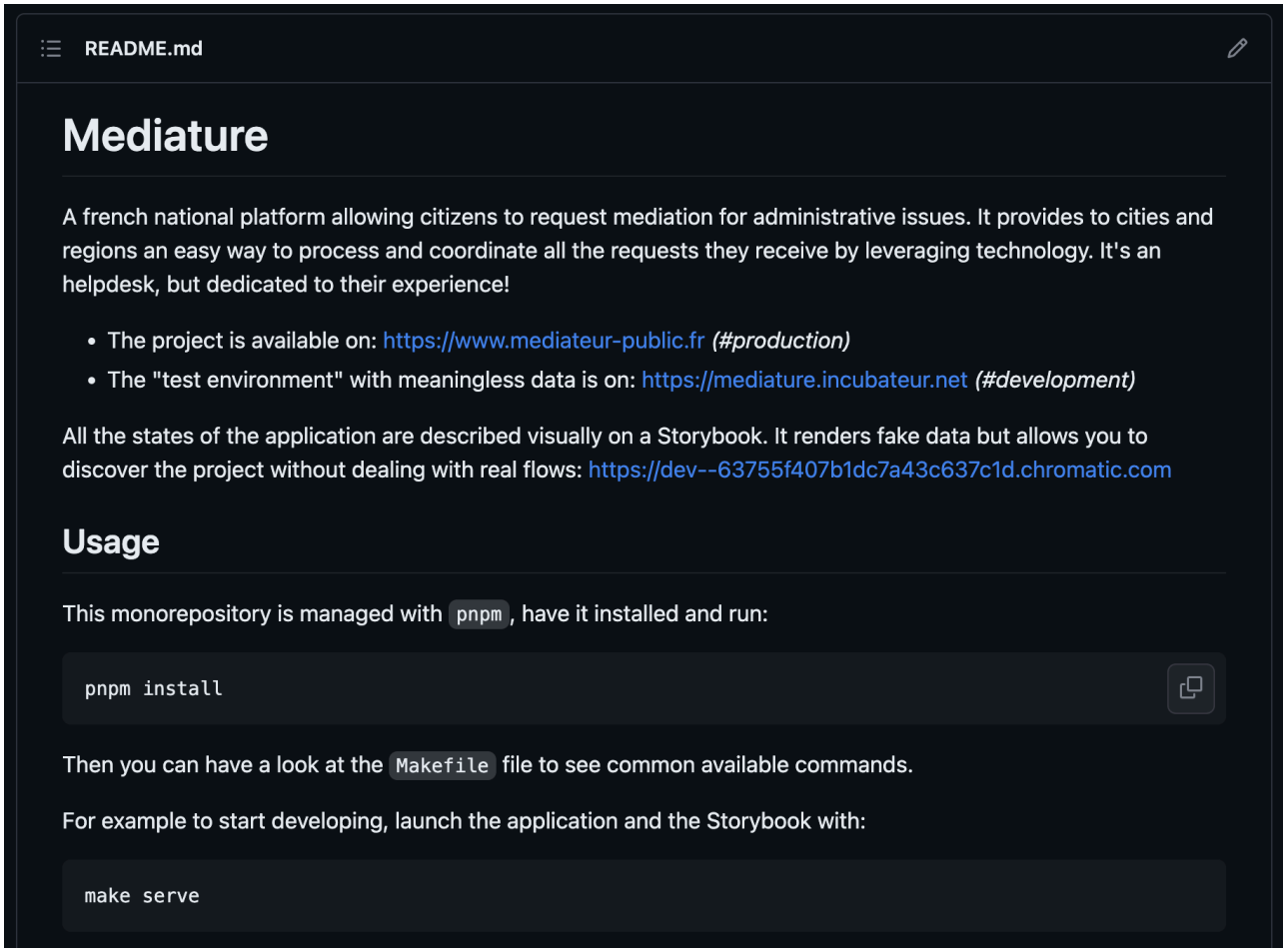
> **INFO**
>
> Une technique souvent vue en Java est de faire des mini-fonctions avec des noms explicites. Donc une fonction qui ferait normalement 50 instructions se retrouve avec juste des appels à 5 sous-fonctions, qui elles-mêmes appellent d'autres. On se retrouve à lire des phrases explicites, mais on retrouve surtout avec une pléthore de fonctions. À vous de juger votre idéal.

La macro-technique

Il est déconseillé de documenter votre projet technique en dehors de votre repository Git. Le simple fait de mettre une distance entre votre code et sa propre documentation fait qu'avec le temps ils seront désynchronisés. Utilisez plutôt des fichiers Markdown qui permettent un formatage assez poussé, tout en bénéficiant du versioning de Git.

Dans un `README.md` mettez :

1. Une courte description du projet ;
2. Les étapes pour lancer votre projet localement (utile pour l'autonomie des personnes) ;
3. Les étapes pour configurer tous les outils nécessaires à l'environnement de production (ceux qui ne peuvent pas être configurés via du code) ;
4. Quelques informations sur les bonnes pratiques à respecter pour contribuer au repository.



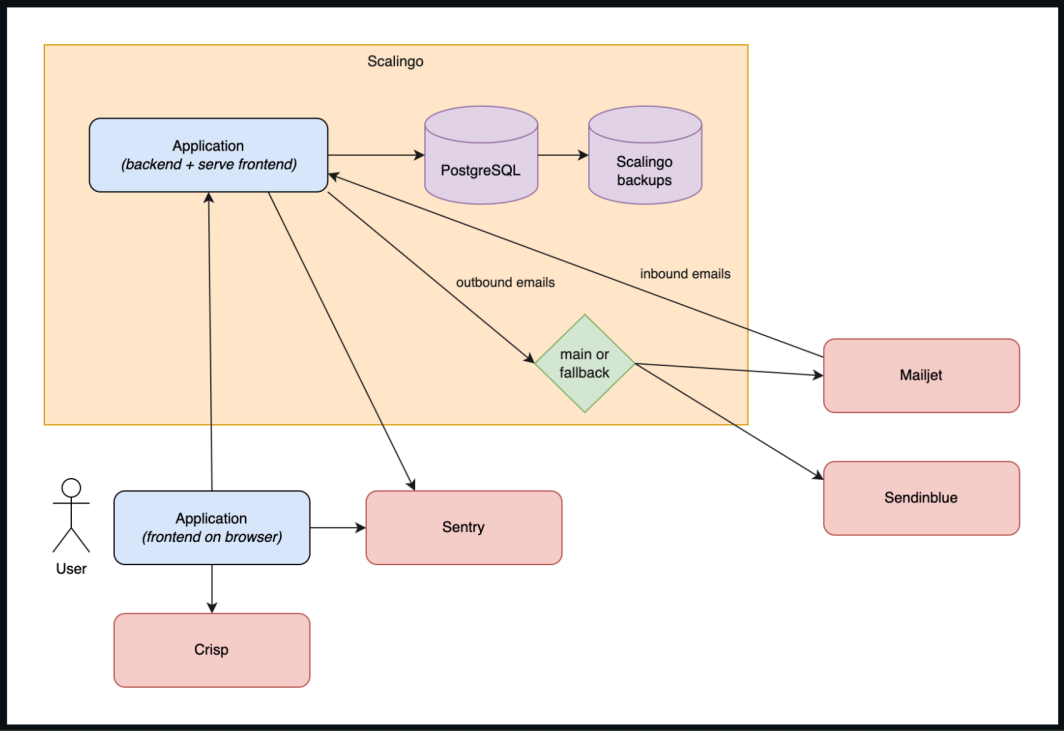
Exemple d'un README.md (*source*)

Sachez que si vous souhaitez faire des schémas de votre architecture (en UML ou pseudo-UML), l'idéal c'est aussi de les versionner. En plus vous pouvez utiliser l'outil draw.io directement dans l'éditeur de code Visual Studio Code, tout se fait au même endroit, c'est hyper pratique.

Dependencies

Target	Dependency	Version	Comment
Application	Librairies	-	Listed in `/package.json`, `/apps/*/package.json` and `/packages/*/package.json` (the entire dependency tree is available into `/pnpm-lock.yaml`)
Application	PostgreSQL	`v14.6+`	The version can differ due to provider upgrades

Services diagram



Exemple d'un schéma intégré à un README (*source*)

Si vous préférez définir vos schémas via du code plutôt qu'une UI, allez voir l'outil Mermaid.

Le schéma de base de données

Nous privilégions l'utilisation d'un ORM pour représenter notre schéma de base de données, comme cela tout comme le code, le schéma est versionné.

Dans le cas où vous utilisez une base relationnelle (mentionnée plus loin dans le chapitre "base de données"), sachez qu'en utilisant un outil comme DBeaver vous êtes en mesure de générer un diagramme représentant votre schéma de données.

Pour cela via l'outil, connectez-vous à votre base de données, puis, dans l'arborescence clic droit sur votre base et "*Voir le diagramme*".

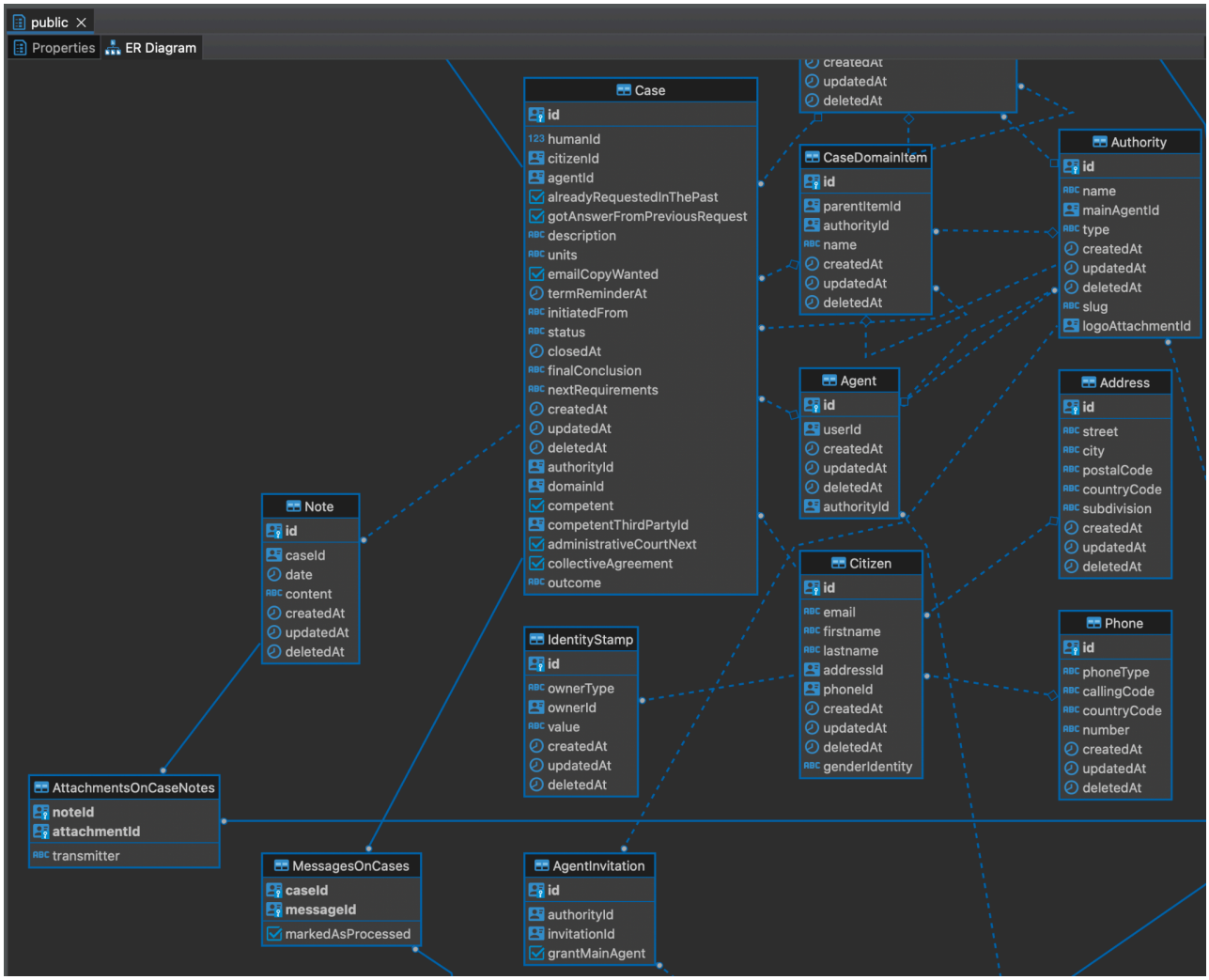


Diagramme d'un schéma de base de données dans DBeaver

> **REMARQUE**
>
> Comme celui-ci découle des relations déclarées par l'ORM dans votre code, il n'y a pas besoin de l'exporter pour le committer. Il reflète en permanence la réalité.

Nommage des commits

Avoir le versioning Git est utile, mais c'est encore mieux si les commits ont une description explicite. Évitez les simples `aaa` ou `fix`, essayez d'être plus verbeux en restant succinct comme `fix: header was not responsive`. Cela vous sauvera quand vous utiliserez la fonction "recherche" dans votre historique de commits, ou bien quand vous ferez un `git blame` afin de savoir qui a fait la modification et pourquoi (cela sert de documentation dans un sens).

Il existe certaines pratiques connues dont Conventional Commits pour correctement nommer et catégoriser les commits, cela apporte un certain standard à travers les projets.

Communauté & open source

📄 Tout internet repose là-dessus

Soyons honnêtes, tout organisme qui fait un produit numérique utilise directement ou indirectement des briques logicielles open source. C'est donc capital de compre...

📄 Puis-je ouvrir mon code ?

Comme cela dépend de votre contexte (quel ministère, secret défense ou non...), on vous laisse jeter un œil au guide de l'équipe Etalab qui y répond en détail 🙌.

📄 Quelle forge logicielle utiliser ?

Nous recommandons d'utiliser GitHub pour héberger votre repository. Bien que leur propre code source ne soit pas ouvert et que nous entendons l'argument "c'est un...

📄 Choix des librairies

Le choix des outils de manière générale est stratégique, il vaut mieux passer des fois quelques heures à la recherche d'un outil stratégique afin de référencer toutes les s...

📄 Fier de votre travail ?

C'est parfait si vous l'êtes 😎, c'est qu'il y a une raison. Dans la continuité de cet esprit de communauté, il peut être intéressant de faire une présentation (sous forme de...

📄 Écrire en anglais ou français ?

Tout comme pour le système international des unités de mesure, il faut bien un moment converger vers un langage commun puisque les outils techniques sont mondia...

📄 Peer-programming & LiveShare

Quand on parle de communauté, il faut aussi penser "local", ça peut très bien être des développeurs environnants autour du projet. Il peut être intéressant d'avoir des ...

Tout internet repose là-dessus

Soyons honnêtes, tout organisme qui fait un produit numérique utilise directement ou indirectement des briques logicielles open source. C'est donc capital de comprendre que sans la volonté de certaines personnes de rendre public leur code et gratuit d'utilisation, il y aurait beaucoup plus de choses à faire soi-même (ou à payer).

"Open source" ne veut pas non plus dire "open bar", il faut garder en tête la licence d'utilisation et le fait qu'une librairie (ou un outil) vit car quelque part des personnes ont fait preuve de bon sens pour ouvrir leur code. Quelques raisons qui ont pu les motiver :

- Philosophie du code libre afin d'apporter à chacun une certaine autonomie dans la réutilisation ;
- Transmettre des connaissances ;
- Bénéficier des remontées de bugs de la part des autres utilisateurs ;
- Bénéficier de la contribution au code de la part d'autres utilisateurs pour développer de nouvelles fonctionnalités ou corriger des erreurs ;
- Donner une visibilité à son travail (une entreprise peut très bien "open sourcer" son code mais en parallèle offrir un service SaaS pour faciliter la vie aux utilisateurs et financer le projet) ;
- Attirer des talents au sein de leur entité (l'open source fait partie des choses qui donnent du sens au travail des développeurs) ;
- Redonner de la valeur à leur travail. Si vous avez travaillé sur un projet qui se meurt et qui est voué à tomber aux oubliettes, l'open sourcer peut permettre

à des gens travaillant sur des problématiques similaires de gagner du temps voire même de reprendre en mains votre projet.

> **INFO**
>
> En complément vous pouvez lire ce guide sur la notion de "communs numériques" pour la comprendre et y participer en tant qu'acteur public.

Il faut voir cela comme du donnant-donnant. Certains salariés ont du temps dédié pour maintenir des outils, d'autres font ça bénévolement sur leur temps libre. Il est vital que vous puissiez contribuer un minimum à cette communauté à la hauteur de vos capacités.

- Vous rencontrez un problème sur un outil ? Regardez dans les "issues" du repository si quelqu'un a déjà rencontré le problème, sinon investiguez vous-même :
  - Si vous trouverez une solution et que le problème est propre à l'outil, partagez-la textuellement ou faites une contribution de code ;
  - Si vous êtes toujours bloqués, n'hésitez pas à publier votre problème (cela pourrait apporter du contexte pour retrouver l'erreur) ;
- Faites en sorte d'ouvrir votre code si votre entité le permet. **Vous travaillez pour le service public, alors autant que votre travail tombe dans le domaine public directement.** Un outil public en France a de grandes chances de pouvoir resservir dans d'autres pays. Tout comme la France bénéficie aussi d'outils que d'autres pays ont pu ouvrir.

> **OÙ ALLER ?**
>
> Les plateformes les plus connues où l'on trouve ces communautés et ces outils sont GitHub, GitLab et Stack Overflow.

Il se peut que les personnes non-techniques de votre entité voient la contribution

open source comme du temps perdu. Vous devrez peut-être faire un peu de pédagogie sur cette philosophie pour que les gens comprennent qu'ils bénéficient aussi de cela. Cela ne peut que donner une bonne image de votre entité si elle devient acteur du développement de tel outil ou telle librairie à succès.

Trouvez l'équilibre sain pour les finances de votre produit. Car effectivement, si vous passez votre temps à contribuer et à régler des problèmes d'autres outils alors que ça ne bénéficie pas directement votre projet… c'est super pour la communauté, mais vous mettez à mal votre propre équipe et projet 🤯.

> **OPEN DATA**
>
> Notez que l'open source va de pair avec l'open data. D'ailleurs l'État met à disposition des données publiques françaises, que ce soit des jeux de données, ou des API comme le site national des adresses.

Puis-je ouvrir mon code ?

Comme cela dépend de votre contexte (quel ministère, secret défense ou non…), on vous laisse jeter un œil au guide de l'équipe Etalab qui y répond en détail 🙌.

Quelle forge logicielle utiliser ?

Nous recommandons d'utiliser GitHub pour héberger votre repository. Bien que leur propre code source ne soit pas ouvert et que nous entendons l'argument "c'est une entreprise américaine" :

- Vous y trouverez la plus grande communauté ;
- C'est basé sur Git, donc vous pouvez migrer vers la concurrence quand vous voulez ;
- C'est gratuit pour les projets open source ;
- Les repositories publics sont archivés par plusieurs entités tierces.

Si cela ne vous convainc pas, rabattez-vous sur GitLab en version SaaS. **Mais ne cherchez en aucun cas à héberger votre propre forge, ça serait démesuré et puis… ce n'est pas votre métier.**

> **INFO**
>
> L'État recense tous les repositories publics déclarés sur la plateforme code.gouv.fr. Donc si vous avez un nouveau compte d'équipe sur GitHub, GitLab ou autre, n'hésitez pas à le référencer en contribuant à leur annuaire.

Choix des librairies

Le choix des outils de manière générale est stratégique, il vaut mieux passer des fois quelques heures à la recherche d'un outil stratégique afin de référencer toutes les solutions existantes pour que ce soit le plus adapté à votre situation et surtout le plus pérenne sur le long terme.

Pour chercher nous vous conseillons d'utiliser quelques mots clefs **en anglais** :

- En cherchant directement sur GitHub : `react modal`
- En cherchant sur votre moteur de recherche :
  - `"github" "react" "modal"`
  - `site:github.com "react" "modal"`
- En cherchant sur un outil comme ChatGPT : `javascript: liste-moi les librairies les plus connues pour afficher des modals en React, en précisant le lien vers leur repository`

Parmi les repositories que vous verrez, quelques indicateurs de "qualité" :

- Ceux qui ont beaucoup de "⭐" (+1), ils sont reconnus dans la communauté ;
- Ceux qui sont anciens (le plus simple est d'aller voir la page des contributeurs pour savoir en quelle année ça a commencé) ;
- Ceux où il y a encore des gens qui maintiennent la librairie en regardant :
  - La section "issues" pour voir le nombre d'ouvertes et surtout si certaines ont été fermées récemment) ;
  - S'il y a des commits récents ;
  - S'il y a des releases récentes ;
- Ceux qui correspondent le plus à vos spécificités techniques.

L'idéal est de combiner le maximum de ces indicateurs pour trouver le bon outil. Cela peut paraître exagéré, mais vous vous garantissez d'éviter d'essuyer les plâtres d'une librairie qui vient tout juste de sortir, ou d'une librairie qui est en train de mourir et où il y a peu de mutualisation du support.

> **IMPORTANT**
>
> Faites attention à toujours vérifier la licence déclarée sur une dépendance que vous voulez utiliser. Et ce, afin de ne pas mettre légalement votre entité dans une mauvaise situation.

Fier de votre travail ?

C'est parfait si vous l'êtes 😎, c'est qu'il y a une raison. Dans la continuité de cet esprit de communauté, il peut être intéressant de faire une présentation (sous forme de "retour d'expérience" par exemple) à vos collègues ou à d'autres entités afin de leur transmettre vos connaissances qui pourraient leur servir.

Si vous ne le sentez pas, une simple trace écrite sur internet peut suffire. Ne vous embêtez pas à héberger votre propre blog pour ça, vous pouvez poster sur des plateformes comme Medium, Reddit, DEV... Au mieux ça servira à d'autres, au pire, vous vous serez entraînés à vulgariser des concepts techniques 🙌.

Écrire en anglais ou français ?

Tout comme pour le système international des unités de mesure, il faut bien un moment converger vers un langage commun puisque les outils techniques sont mondiaux. Certaines personnes diraient qu'il faut continuer à tout faire en français mais :

- L'écosystème open source est quasi exclusivement en anglais. Si vous souhaitez obtenir de l'aide dans la communauté il vaut mieux présenter un contexte de projet où les gens peuvent comprendre votre code et votre documentation pour reproduire l'erreur de leur côté ;
- Sachant que vous utiliserez forcément des librairies, vous finiriez avec un mix français/anglais dans les noms des fonctions, des variables, et des concepts ;
- Ne pas être en anglais limite la réutilisabilité de votre outil par la communauté.

Deux exceptions :

- Si les membres de votre équipe n'ont pas les bases en anglais, il devient donc évident que vous perdriez du temps (mais il faut quand même envisager à ce que les personnes s'y forment car c'est justement indispensable pour tous ces aspects communautaires) ;
- Si vous utilisez des concepts très spécifiques à la France et que tenter de les traduire porterait à confusion.

En partant sur une rédaction en anglais on peut faire le choix d'ajouter un glossaire anglais/français dans la documentation afin d'identifier explicitement toutes les "entités métiers" (e.g. citizen/citoyen, admin/administrateur, case/dossier, message/

message...).

> **REMARQUE**
>
> Concernant les URLs, dans certains cas les mettre en français est un prérequis que ce soit politique ou pour apporter une certaine confiance aux utilisateurs de la plateforme. Si vous ne voulez pas interférer avec le fait d'avoir écrit votre outil en anglais, vous pouvez rajouter des règles d'URL rewriting.

Peer-programming & LiveShare

Quand on parle de communauté, il faut aussi penser "local", ça peut très bien être des développeurs environnants autour du projet. Il peut être intéressant d'avoir des moments de rencontre ou d'investigation afin de partager vos questionnements ou avancées.

Si jamais vous êtes à distance et que vous souhaitez vous plonger dans du code pour illustrer votre problème, sachez que Visual Studio Code a un module LiveShare afin que n'importe qui puisse ouvrir un éditeur virtuel pour interagir avec votre code local.

Souveraineté du projet

Enjeu national

Sans rentrer dans des débats passionnés, et comme votre métier n'est pas de tout faire, vous allez forcément à un moment dépendre d'une entreprise privée (héberge…

Portabilité du projet

Cela s'est déjà vu que des entreprises privées françaises passent sous giron étranger, ou plus simplement qu'une entreprise cesse son activité. Cela n'est malheureuseme…

Enjeu national

Sans rentrer dans des débats passionnés, et comme votre métier n'est pas de tout faire, vous allez forcément à un moment dépendre d'une entreprise privée (hébergement, stockage…). Il faut au maximum privilégier dans l'ordre : des entreprises françaises, européennes, puis dans le reste du monde.

- Il y a un enjeu d'un point de vue savoir-faire, et dévaloriser ces acteurs locaux est destructeur pour la filière technologique française ;
- Cela s'est déjà vu que des pays coupent des accès technologiques stratégiques pour pénaliser d'autres pays ;
- Cela aide à respecter les aspects légaux qui vous incombent (voir RGPD…).

TRANSFERT DES DONNÉES

En juillet 2023 la Commission européenne adopte une décision d'adéquation quant au transfert de données vers les États-Unis. Mais de notre côté nous préférons **toujours** privilégier les acteurs français. Cela fait plusieurs fois que de telles décisions européennes sont tantôt adoptées tantôt annulées… Il est difficile de garantir la souveraineté d'une donnée à partir du moment où elle est soumise à une autre juridiction.

Portabilité du projet

Cela s'est déjà vu que des entreprises privées françaises passent sous giron étranger, ou plus simplement qu'une entreprise cesse son activité. Cela n'est malheureusement pas de notre ressort mais il faut quand même limiter la casse.

Pour chaque entreprise privée que vous choisissez, il faut que la fonctionnalité dont vous avez besoin se base sur des standards *(nous excluons les projets de niches qui font de la R&D et où les standards peuvent ne pas encore exister)*.

Par exemple :

- Envoi d'email : utiliser le protocole SMTP plutôt que l'API interne du provider permet d'interchanger facilement les acteurs ;
- Hébergement applicatif : avoir le produit final disponible dans un "container" (type image Docker ou Buildpack) permet une compatibilité très large avec les hébergeurs ;
- …

Sans parler des problématiques de services tiers utilisés, il se peut aussi que votre produit soit transféré dans une autre entité interne, où l'hébergeur et les outils sont différents… C'est là qu'avoir une certaine portabilité aide. Après il faut rester lucide, toute migration demande un investissement. Il faut juste essayer de limiter les ajustements à ce qui est environnant et non à ce qui est relatif au code applicatif.

Un début de réponse est de simplifier la stack technique au maximum afin d'être le plus possible interopérable.

Utiliser la charte graphique de l'État

DSFR, quesako ?

Le SIG (Système d'Information du Gouvernement) développe et maintient le "Système de Design de l'État" (DSFR). Ce design system permet aux produits numériques ...

En fonction des technologies

Le DSFR est techniquement agnostique de toute librairie ou framework. L'idée est que peu importe le contexte où vous vous trouvez vous êtes en mesure de l'utiliser a...

En fonction de l'usage

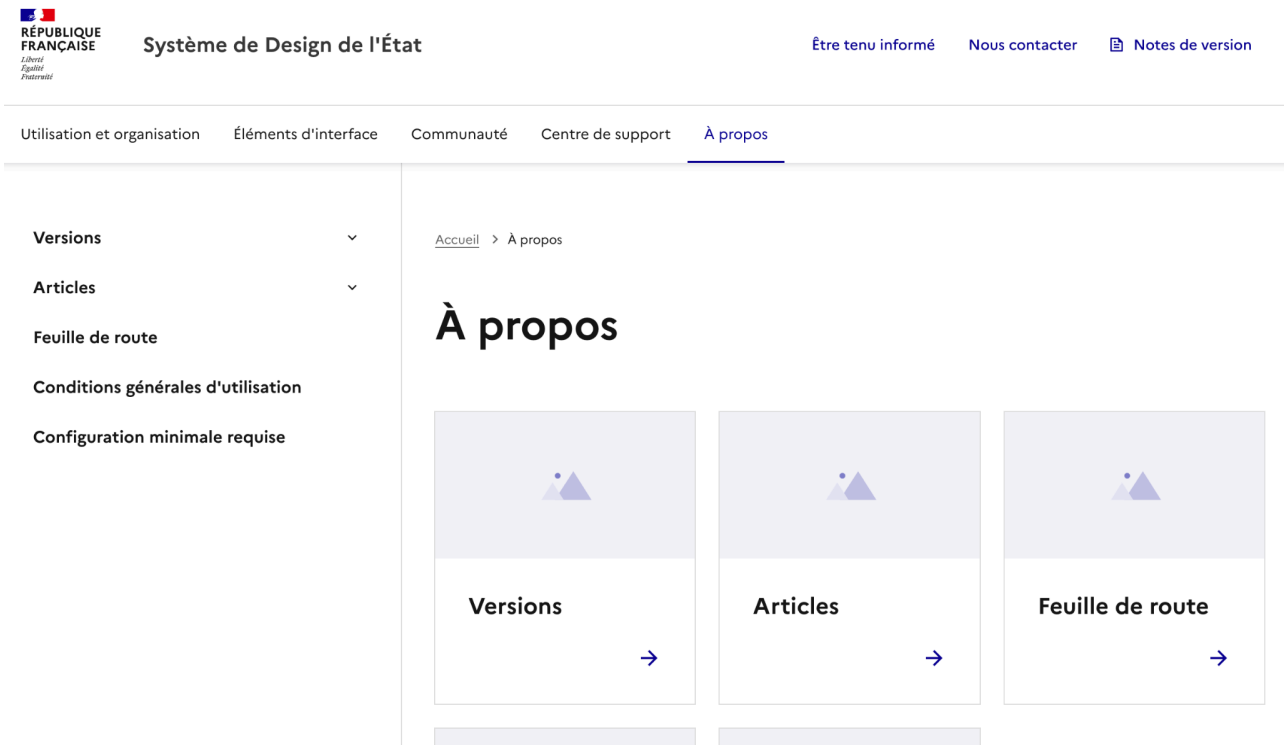
Il peut arriver que l'adoption du DSFR ne soit pas la "silver bullet" pour votre applicatif :

Un support à disposition

L'équipe du DSFR met à disposition un Slack pour échanger voire pour demander de l'aide. Vous pouvez demander l'accès ici.

DSFR, quesako ?

Le SIG (Système d'Information du Gouvernement) développe et maintient le "Système de Design de l'État" (DSFR). Ce design system permet aux produits numériques de l'État d'harmoniser leur apparence graphique tout en mutualisant l'effort en termes de développement graphique puisqu'il met à disposition un ensemble de composants prêts à l'emploi.



Aperçu d'une page utilisant le DSFR (*source*)

UTILISATION

Le DSFR est un standard **obligatoire** dans l'administration, et son utilisation est **exclusivement réservée aux produits numériques de l'État**.

En fonction des technologies

Le DSFR est techniquement agnostique de toute librairie ou framework. L'idée est que peu importe le contexte où vous vous trouvez vous êtes en mesure de l'utiliser avec du simple HTML, CSS, JS.

Cette version brute ne permet par contre pas de bénéficier de la puissance des frameworks web que vous utilisez. Pour combler cela certaines initiatives ont émergées pour implémenter le "DSFR brut" dans différents frameworks. Quelques références :

- Pour React : `react-dsfr` **(celle que nous préconisons)**
- Pour Vue : `vue-dsfr`
- Pour Angular : `ngx-dsfr`
- Pour Django : `django-dsfr`
- *Rappel de la version "brute" officielle :* `dsfr`

ASTUCE

A noter qu'il existe encore un flou sur le formatage des emails et des PDFs. Comme il n'y a rien d'officiel, vous pouvez lire ça pour les emails et ça pour les PDFs afin de découvrir une proposition de formattage.

En fonction de l'usage

Il peut arriver que l'adoption du DSFR ne soit pas la "silver bullet" pour votre applicatif :

- Si votre socle de projet utilise déjà un autre framework UI (Bootstrap, MUI, Vuetify…)
- Si vous estimez que les composants proposés par le DSFR ne suffisent pas (vous voulez utiliser un datepicker, combobox…)

Il est possible d'utiliser des "thèmes pseudo-DSFR" pour d'autres frameworks afin de combler les manques du DSFR. La librairie tierce `dsfr-connect` tente de répondre à ce besoin.

Un support à disposition

L'équipe du DSFR met à disposition un Slack pour échanger voire pour demander de l'aide. Vous pouvez demander l'accès ici.

REMARQUE

Il faut une adresse email terminant par `@*.gouv.fr` afin d'y être éligible. Si ce n'est pas votre cas et que vous travaillez sur le DSFR, vous pouvez toujours faire une demande. Dans le pire des cas et si vous êtes bloqués, vous pouvez toujours poster une issue sur leur repository.

Recommandations techniques argumentées

Les recommandations qui vont suivre permettent selon nous d'allier robustesse et efficacité lors du développement logiciel pour un produit numérique.

🗃️ Rapprocher le frontend du backend

3 éléments

🗃️ Un langage typé

3 éléments

🗃️ Une même stratégie : un framework web

3 éléments

🗃️ Toujours utiliser un module i18n

2 éléments

📄 … Et le framework backend ?

Comme décrit plus haut, on privilégie un monolithe (à l'ancienne 👴). Nous préconisons Next.js côté backend afin d'avoir une intégration native avec React. Essayez de…

🗃️ Communications client-serveur

6 éléments

🗃️ Une base de données qui fait tout (vraiment)

10 éléments

🗃️ Un hébergeur simple et souverain

4 éléments

📄 Authentifier les utilisateurs

Vous pouvez adopter plusieurs stratégies en fonction de votre produit (ce choix découle de l'investigation, ce n'est pas un choix technique) :

📄 L'intégration et le déploiement continu (CI/CD)

Nous préconisons d'utiliser une pipeline de CI/CD décrite dans votre repository. Si votre repository est par exemple sur GitHub nous ne voyons pas l'utilité d'utiliser un ...

📄 Avoir différents environnements

Combien nécessaires ?

📄 Les tests et la limite du mock

Nous ne sommes pas pour imposer des tests à tous les niveaux et sur tout votre code applicatif. Il est beaucoup plus important selon nous de :

🗃️ Backups

3 éléments

📄 Design patterns

Nous n'avons pas de design pattern à vous préconiser d'un point de vue logiciel. C'est aussi pour cela que l'on ne fournit pas de "stack toute faite à copier/coller". Ce af...

Rapprocher le frontend du backend

Choisir un même langage

Nous préconisons d'utiliser JavaScript (TypeScript surtout 😉) pour votre frontend et votre backend. Ce n'est pas le langage idéal techniquement de notre point de vue ...

Avoir le code dans un même repository

Sans parler de langage, nous vous conseillons d'avoir un monorepo (un seul repository) pour tout le code produit que vous pourriez avoir. Cela évitera les prises de tête ...

... et rapprocher le frontend du frontend

Certains produits numériques peuvent nécessiter "plusieurs frontend". L'exemple le plus simple est de souhaiter avoir un applicatif et une landing page (site vitrine). Il p...

Choisir un même langage

Nous préconisons d'utiliser JavaScript (TypeScript surtout ☺) pour votre frontend et votre backend. Ce n'est pas le langage idéal techniquement de notre point de vue pourtant la majorité des outils ont besoin d'une interface web, et même si vous deviez être présent sur mobile, à moins de faire un jeu vidéo vous seriez à même de développer une application native hybride qui utilise des technologies web.

Le fait d'aussi s'en servir côté backend va vous permettre de partager des communs. Imaginons que vous ayez une fonction pour filtrer ou trier une liste de départements, il y a de grandes chances que ce soit la même logique sur frontend, et sur backend. Avec cette stratégie, nous sommes aussi en mesure de définir des règles communes de validation de formulaire sur le frontend et sur le backend. On le verra plus loin, mais des choses qui étaient auparavant seulement générées côté serveur peuvent maintenant l'être côté frontend, cela permet de prévisualiser des emails, des PDFs…

COMMUNAUTÉ IMPORTANTE

De part l'omniprésence du JavaScript sur les frontends web, vous aurez de la facilité à trouver des personnes lors de vos recrutements.

Et pour être fair-play, un désavantage est que la communauté est très (trop ?) importante. Il y a peut-être trop d'initiatives différentes pour faire la même chose.

Avoir le code dans un même repository

Sans parler de langage, nous vous conseillons d'avoir un monorepo (un seul repository) pour tout le code produit que vous pourriez avoir. Cela évitera les prises de tête comme :

- Lorsque que vous travaillez sur une fonctionnalité vos commits devraient se suffire pour décrire tout le travail fait. Vous ne devriez pas avoir de commits dupliqués du type `feat: set up registration form` sur chacun de vos repositories de frontend, backend, et déploiement… c'est impossible à tracer, et si vous fonctionnez avec des reviews de pull requests c'est extrêmement compliqué pour la lisibilité mais aussi pour monter le setup localement ;
- Le frontend et backend restent couplés :
  - Si vous avez un repository frontend, et un pour le backend, il faut que vous soyez synchronisés pour qu'ils soient déployés en même temps afin de garantir que le frontend n'utilise pas des choses qui n'existent pas encore sur le backend, ou que le backend a besoin dorénavant d'informations spécifiques. *(on peut essayer de se dire que l'on ne produit pas de breaking change dans le contrat de communication entre le frontend et le backend, mais **ça complexifie quelque chose qui devrait être simple**)* ;
  - Si vous avez des communs en termes de code, il ferait sens d'avoir un repository "common" puisque ça ne fait pas plus partie du frontend que du backend. Mais à ce moment vous vous lancez dans la guerre de la gestion des dépendances pour être sûr qu'à chaque fois, le frontend ET le backend utilisent la même version du "common".

On a parlé de monorepo, mais en fait ce qu'il faut retenir : **utilisez un monolithe !** Cela vous permettra d'utiliser un framework unifié (frontend + backend) pour bénéficier sur étagères d'utilitaires testés et approuvés par une communauté : logique de communication, routing, connexion…

Et cela tombe bien, le fait d'utiliser du JavaScript pour le frontend et le backend simplifie la stack du développeur 😀.

> **REMARQUE**
>
> Si la notion de microservices vous semble indispensable, dites-vous qu'en organisant bien votre monolithe vous tirerez la plupart des bénéfices sans les désavantages (gestion de dépendances, déploiements différents…). Vous pouvez vous renseigner sur le pattern "modular monolith".

… et rapprocher le frontend du frontend

Certains produits numériques peuvent nécessiter "plusieurs frontend". L'exemple le plus simple est de souhaiter avoir un applicatif et une landing page (site vitrine). Il peut donc être tentant d'en faire 2 applicatifs distincts, dans les mêmes technologies (pour éviter de se disperser), et idéalement de les mettre dans le monorepository mentionné plus haut.

Mais vous devrez alors gérer plusieurs fois la quasi même configuration : même dépendances, même gestion d'URL, même configuration du framework, même gestion de tests, même client pour la base de données…

Il peut être intéressant de les gérer depuis le même applicatif, en jouant simplement sur le préfix du routage des URLs :

- `/landing/*`
- `/app/*`

… Comme cela un seul runtime suffit pour 2 frontends (voire même plus).

Bien entendu vous pourriez avoir des besoins spécifiques pour les URLs (SEO, esthétique…) pour par exemple avoir comme URLs :

- `https://www.example.com/` (landing)
- `https://app.example.com/` (app)

Plusieurs solutions sont envisageables…

Par le routage

Vous gardez 1 seule instance pour gérer tous les frontends, par contre il faut que votre hébergeur et/ou votre framework vous autorisent à gérer plusieurs domaines pour le trafic entrant afin de faire la bonne correspondance (tel domaine va utiliser tel préfixe dans le pathname… *Cela peut s'apparenter à de l'URL rewriting*).

Par le déploiement

Si la solution par routage n'est pas envisageable, vous pouvez vous tourner vers votre pipeline de déploiement. Au lieu d'avoir une seule étape de *build* après tous vos tests… vous en avez 2 où pour chacune vous pouvez choisir le routage voulu (`A` ou `B` plutôt que `A+B`). Et vous déployez chacun des builds à 2 endroits différents. Évitant ainsi une configuration de routage sur les domaines racines.

Un langage typé

Éviter les bugs bêtes au runtime

Nous vous conseillons quand c'est possible d'avoir tout votre code typé, et de privilégier des librairies qui ont été typées. Car définir des variables où vous admettez qu…

Éviter d'écrire une documentation du code

Les types apportent une compréhension sur comment une fonction doit être appelée et appréhender ce qu'elle retourne. C'est un gain de temps non négligeable lors …

TypeScript… typé mais pas compilé

En vous recommandant d'utiliser du JavaScript tout en prêchant le typage, nous vous préconisons d'utiliser le TypeScript.

Éviter les bugs bêtes au runtime

Nous vous conseillons quand c'est possible d'avoir tout votre code typé, et de privilégier des librairies qui ont été typées. Car définir des variables où vous admettez qu'il peut y avoir tout et n'importe quoi dedans complique la compréhension du code et est prompt à des erreurs au moment du runtime. Imaginons que dans un commit vous transformiez une variable qui était un *integer* en un *string*, sans typage vous n'avez aucun moyen de savoir où cela a pu créer des incohérences et incompatibilités dans le code.

Certains pourraient dire qu'on peut utiliser des tests unitaires... Dans les faits, rien que le fait d'avoir des types est un test en soi 😜.

> **REMARQUE**
>
> Faire un script sans type est "acceptable", mais dès que vous vous tournez vers la réalisation d'une librairie ou d'un produit, être sans typage est synonyme de souffrance 🥵.

Éviter d'écrire une documentation du code

Les types apportent une compréhension sur comment une fonction doit être appelée et appréhender ce qu'elle retourne. C'est un gain de temps non négligeable lors du développement.

Si on définit le contrat de notre API avec les types (voir le chapitre sur tRPC), on peut même utiliser des outils pour automatiquement générer des documentations d'API, voire générer des clients d'API dans plusieurs langages.

TypeScript… typé mais pas compilé

En vous recommandant d'utiliser du JavaScript tout en prêchant le typage, nous vous préconisons d'utiliser le TypeScript.

Sans rougir, nous allons énumérer les points négatifs de TypeScript en essayant d'y voir le positif. Pour bénéficier de la large prise en charge du JavaScript, le TypeScript est transpilé en JavaScript afin d'être interprété par les navigateurs et serveurs. Ce qui implique :

1. On se retrouve avec une communauté à ~2 facettes :
    i. D'un côté des librairies codées en JavaScript "brut" ;
    ii. De l'autre celles codées en TypeScript ;
    iii. Voire certaines qui font un peu des deux ;
2. Le typage devient une vue de l'esprit puisqu'au runtime tout est permis (vu que c'est transpilé en JavaScript) :
    i. Un code TypeScript peut très bien ne pas respecter les types (il "caste" les variables) ;
    ii. Même si les codes TypeScript ont bien été faits, notre confiance s'arrête au moment où le typage s'arrête (beaucoup de librairies historiques sont en JavaScript).

Pour vous rassurer :

- La communauté met à disposition des fichiers de typage pour les librairies en JavaScript (ça ne garantit pas le runtime mais ça aide) ;

- Si vous partez du principe que vous typez tout votre produit sans caster les variables, il y peu de souci à vous faire au niveau de votre code. Encore plus avec une mutualisation des types entre votre frontend/api/backend ;
- Dans les faits, il est extrêmement rare qu'une librairie connue retourne au runtime un type différent du typage indiqué. Cela peut arriver, mais cela reste largement moins fréquent qu'un développeur qui fera de toute façon ses propres erreurs dans son code. Il faut donc relativiser ;
- Ce n'est pas propre à l'écosystème JavaScript. Pour n'en citer que quelques-uns, PHP, Python, Ruby sont aussi historiquement des langages non-typés et qui essaient au fil du temps de combler ce manque ;
- Pour garantir du "totalement typé" il faut se tourner vers des langages compilés comme le C++, Golang, Rust... Ils sont d'une rigueur très appréciables mais apportent aussi leur lot de limitations (facilité d'adoption, flexibilité d'utilisation...). On pourrait imaginer s'en servir pour le backend, mais cela limiterait la réutilisabilité des communs avec le frontend (puisque les frontends web n'interprètent que le JavaScript).

Une même stratégie : un framework web

Les composants

L'intérêt principal d'utiliser un framework de composants est la réutilisabilité. En fonction de comment les composants sont structurés ils peuvent être autonomes et tr...

React

Comme framework web nous préconisons React car il reste au fil du temps une valeur sûre, pour :

Ce n'est pas un problème pour une application mobile

En choisissant un framework web votre frontend va être compatible avec quasiment toutes les situations. Par exemple, si vous souhaitez développer une application m...

Les composants

L'intérêt principal d'utiliser un framework de composants est la réutilisabilité. En fonction de comment les composants sont structurés ils peuvent être autonomes et transportables (embarquant leur logique, leur template, et leur style).

On se rapproche de la logique des "web components" nativement supportés dans les navigateurs depuis quelques années, sauf que les "framework components" (qui nécessitent une compilation) apportent quelques facilités au développement.

> **ASTUCE**
>
> À noter que c'est peu connu mais vous pouvez très bien utiliser des "framework components" dans des applicatifs sans compilation. Par exemple, sur un site Wordpress vous pourriez très bien charger le framework React avec une balise `<script>` et ensuite utiliser vos composants React qui seront compilés à la volée.
>
> En termes de performance c'est plus lourd, mais ça montre à quel point c'est applicable partout. Cela peut d'ailleurs vous évitez d'aller refaire du JavaScript brut… 😊

React

Comme framework web nous préconisons React car il reste au fil du temps une valeur sûre, pour :

- Sa constance au fil du temps ;
- Son adoption : c'est l'un des frameworks web les plus utilisés au monde ;
- Il y a une palanquée d'outils ;
- Le fait que depuis le début ils imposent l'utilisation du JSX pour le templating.

Dans notre continuité de vouloir "typer" de bout en bout notre code, il est important de faire le lien entre le code métier et le template HTML afin de savoir durant le développement si nous faisons une erreur de type ou de syntaxe. C'est exactement en ça que sert le JSX.

> **NOTE**
>
> On parle souvent de JSX quand on fait du JavaScript, et de TSX quand on fait du TypeScript.

MyComponent.tsx | Exemple de composant React avec un templating TSX

```tsx
import React from 'react';

interface MyComponentProps {
  name: string;
}

```

```tsx
function MyComponent({ name }: MyComponentProps) {
  return <h1>Hello, {name}!</h1>;
}

export default MyComponent;
```

> **QUID DES AUTRES ?**
>
> Vue.js aurait pu être une alternative mais son principal problème est de proposer son propre templating par défaut, ne faisant pas le lien avec le code métier. Il y a toujours la possibilité d'utiliser du JSX mais le fait que ce ne soit qu'optionnel ne nous plaît pas. De plus, avec la version "Vue 3" leur syntaxe a complètement changé et on se rapproche de plus en plus vers ce que fait depuis longtemps React.

Ce n'est pas un problème pour une application mobile

En choisissant un framework web votre frontend va être compatible avec quasiment toutes les situations. Par exemple, si vous souhaitez développer une application mobile vous pouvez très bien développer "une application web" qui sera encapsulée dans une application hybride native. En fait, l'application native va utiliser une "webview" (vue du navigateur natif) pour faire le rendu de l'application web, sauf que l'utilisateur ne s'en rend pas compte puisque l'application web apparaît en plein écran (sans barre d'URL…).

C'est une bonne stratégie pour gagner en coût de développement :

1. Si votre application web est "responsive" elle pourra alors s'afficher sur un desktop, une tablette, ou un téléphone. Et ce, que ce soit dans un navigateur ou dans une application hybride ;
2. Durant votre période de tests et pour éviter de gérer des contraintes relatives à Android ou iOS, il vaut mieux permettre aux personnes de tester l'application via une URL, quitte à activer le mode PWA (progressive web app) afin qu'ils puissent la mettre sur leur écran d'accueil comme une vraie application ;
3. Vous écrivez le même code JavaScript/TypeScript pour gérer une version web, Android, iOS, Electron… Alors que si vous partiez en plus du web sur du natif

Android (Java, Kotlin) et iOS (Objective-C, Swift), vous devriez trouver des librairies dans chaque langage, donc décliner votre fonctionnalité en 3 codes différents, et espérer avoir trouvé des librairies avec des rendus graphiques similaires. **Une perte de temps considérable !** *(à noter qu'avec une application hybride il est possible que vous touchiez un peu au code natif pour configurer la webview ou des plugins, mais ça reste rare et à la portée de tous car énormément de plugins tout faits existent)*

4. Une fois les bases natives pour l'hybridation mises en place, à partir du moment où vous ne touchez plus qu'à l'application web vous pouvez même vous permettre de ne développer que localement sur le navigateur de votre ordinateur. Vous êtes quasiment garantis que cela marchera sur mobile une fois compilé dans l'application hybride.

Nous recommandons pour cela l'utilisation de Capacitor.

> **REMARQUE**
>
> Apache Cordova a longtemps été une référence mais il masquait trop la couche native, ce qui limitait la personnalisation. Son petit frère Capacitor corrige tous ses défauts.

> **MISE EN GARDE**
>
> **Il est important de justifier votre besoin d'avoir une application mobile.** C'est un chemin semé d'embûches alors que certains produits pourraient démarrer avec une simple version "web" :
>
> - Pour compiler l'application hybride, vous avez besoin de faire pareil que pour une application "native", il faut utiliser des outils comme Android Studio ou Xcode (pour iOS) ;
> - Pour iOS il faut forcément posséder un Mac pour compiler votre application (ou utiliser un service en ligne payant mais ça complique le

débogage) ;

- Pour le Google Play le coût **par** compte développeur est de ~25 euros à vie, et pour l'App Store il est de ~100 euros **par an et par** compte développeur (si vous ne payez plus vous ne pourrez plus mettre à jour votre application) ;
- Les API natives de l'OS ont souvent des mises à jour qui apportent des breaking changes. Et vous ne pouvez pas vous permettre de les ignorer car les stores (Google Play et App Store) vous forcent à garder un certain rythme de croisière de mises à jour, au risque sinon de ne plus être listé chez eux ;
- Chaque déploiement d'une version d'application passe forcément par des validations de la part des stores. Cela prend majoritairement quelques heures pour un environnement "beta", voire quelques jours pour de la "production". Vous vous retrouvez donc à devoir essayer de synchroniser votre déploiement backend par rapport aux délais des stores. Notre conseil est de ne pas utiliser l'option "déployer automatiquement" après validation (puisque pour Google Play et App Store, ils ne donneront pas leur verdict en même temps). Au lieu de cela, vous attendez que tout soit au vert pour cliquer sur les boutons en même temps (juste après avoir déployé votre backend). **Le fix d'un bug n'est donc pas immédiat pour les utilisateurs ;**
- Pour compléter le point précédent, contrairement à un site internet où une nouvelle version est normalement récupérée directement à l'utilisation, avec le mécanisme des stores vos utilisateurs ne sont pas obligés de faire les mises à jour tout de suite, voire jamais. Vous vous retrouvez donc à devoir gérer une API sans aucun breaking change le temps que **tous** vos utilisateurs fassent la migration. *On peut utiliser des plugins pour dire à l'ouverture de l'application "hey, une nouvelle version est disponible, téléchargez-la" en rendant cela optionnel ou obligatoire, mais ça dépend de votre stratégie et de vos utilisateurs… veulent-ils voir*

ce message intempestif à chaque mise à jour ?

- Aussi, comme les validations des stores sont tantôt automatiques, tantôt manuelles, vous avez pu passer 10 fois le processus et malheureusement le jour où vous avez ABSOLUMENT besoin d'un déploiement, ils vous bloquent en vous reprochant quelque chose d'arbitraire. Et par malchance cela prendra 10 jours ou plus. Il faut alors avoir un contact privilégié dans son sac pour espérer accélérer le processus. Cela s'est déjà vu avec certaines applications de grandes banques françaises...

Le but n'est pas de vous faire fuir, mais plutôt de vous "préparer" à la dure réalité du déploiement mobile. Mais ce n'est en rien insurmontable 📱🚀.

Toujours utiliser un module i18n

Même si votre produit est monolingue

Le cas de figure le plus évident étant bien sûr si vous devez proposer votre site en plusieurs langues (exemple refugies.info). Mais même dans le cas contraire, il peut vou…

i18next

Nous utilisons le "standard" i18next car son écosystème est très important depuis des années. Vous ne devriez pas avoir de mal à faire ce que vous voulez en le choisiss…

Même si votre produit est monolingue

Le cas de figure le plus évident étant bien sûr si vous devez proposer votre site en plusieurs langues (exemple refugies.info). Mais même dans le cas contraire, il peut vous permettre de pallier à plusieurs problèmes.

Il vous permet de facilement gérer le singulier et le pluriel d'une phrase (cas très commun), voire d'afficher une phrase en fonction du genre (cas moins utilisé en général). Sans un tel module vous seriez en train de faire des `if / else` dans votre code métier et votre template.

Ensuite, il vous permet aussi de centraliser le formatage des dates. Plutôt que d'écrire à différents endroits `dd/MM/yyyy` ou encore `dd/MM/yyyy HH:mm:ss`, vous pouvez les définir dans des traductions nommées `date.default` et `date.defaultWithTime`. Ensuite utilisez l'interpolation de votre module i18n pour dire "*quand la variable en entrée est une date, alors j'applique le formatage de ma librairie de dates dessus*". Dans le cas où votre applicatif doit gérer d'autres langues, vous n'aurez qu'à créer les fichiers de traductions sans toucher au code qui s'occupe d'afficher des dates !

> **ASTUCE**
>
> Nous recommandons d'utiliser `date-fns` pour le formatage des dates et des intervalles. Il permet quelques raccourcis comme `PPPP` ou encore `PPPPpp` (voir leurs exemples).

Notez que si votre produit est monolingue, il ne faut pas perdre votre temps à

mettre tous vos textes dans le module de traduction. L'idée est surtout de se limiter aux phrases "dynamiques". Quand on souhaite traduire tout l'applicatif (hors contenu stocké en base de données), on se retrouve à un moment donné confronté à des imbrications multiples de textes traduits avec variables, où l'on voudrait mettre seulement une partie en gras… sauf que, comment passer du HTML sachant qu'il faut aussi en même temps l'encoder afin d'être sûr qu'il n'y ait pas d'injections… bref, gardez votre énergie pour autre chose !

i18next

Nous utilisons le "standard" i18next car son écosystème est très important depuis des années. Vous ne devriez pas avoir de mal à faire ce que vous voulez en le choisissant (par exemple en utilisant les mêmes phrases sur le frontend et le backend si besoin).

Pour le développement nous vous conseillons d'utiliser i18n Ally dans Visual Studio Code afin d'avoir un aperçu de la phrase réelle dans le code en lieu et place de l'identifiant i18n (cela aide à contextualiser).

Si seul le développeur s'occupe de la traduction il peut gérer cela simplement via les fichiers JSON de traductions. Si un non-technique entre en jeu, vous pouvez investiguer des outils plus amicaux comme BabelEdit ou Locize.

… Et le framework backend ?

Comme décrit plus haut, on privilégie un monolithe (à l'ancienne 👴). Nous préconisons Next.js côté backend afin d'avoir une intégration native avec React. Essayez de partir avec au minimum Next.js v13 en configurant le router avec "appDir".

Cette version utilise React v18 qui introduit les RSC (React Server Component), permettant le rendu HTML directement sur le serveur sans recalculer les composants depuis le frontend. Vous connaissez peut-être de longue date le SSR (server-side rendering) avec React, si vous voulez connaître la différence exacte avec les RSC, lisez ça.

> **HISTORIQUE**
>
> Avant, React était surtout connu pour être fourni en SPA (Single Page Application), mais cela rend le SEO pauvre et votre application n'était pas très bien référencée sur les moteurs de recherche. Là, les RSC apportent un nouveau souffle (bien mieux que faire du React SSR). Et si vous les codez bien, ils peuvent être compatibles pour un rendu sur serveur tout autant que sur frontend (ce qui facilite la réutilisation entre différents outils).

Les avantages avec les RSC :

- Meilleur SEO ;
- Écoconception (il n'y a plus d'hydratation des composants côté frontend) ;
- Rapidité du rendu de la page avec son chargement (valable aussi en

développement).

Communications client-serveur

Réduire au maximum les intermédiaires

Plus vous ajoutez de couches intermédiaires, plus vous faites de "mappers" (et donc du boilerplate code). Bien entendu, ne pas réutiliser les mêmes structures de donn…

tRPC

tRPC veut dire "TypeScript Remote Procedure Call". Vous l'aurez peut-être deviné mais cela va vous permettre de définir les contrats de vos endpoints d'API avec du Ty…

Permettre l'évolutivité

Notre parti pris est de faire du TypeScript de bout en bout, mais sachez qu'en utilisant tRPC vous ne vous bridez pas à simplement générer des clients TypeScript. Vous…

Validation des données entrantes

Que ce soit sur vos endpoints tRPC ou sur un endpoint à part dans Next.js pour recevoir des webhooks, la devise est de toujours valider les données entrantes.

Préférez l'API à la mise en cache frontend

Qu'est-ce que ça veut dire ? Dans React, un outil comme Redux permet de gérer des "stores" pour gérer l'état global de votre frontend au fil des pages. Comme entre le…

Avoir de la réactivité avec une API traditionnelle

Lorsque votre application autorise des interactions entre plusieurs utilisateurs (directement ou indirectement), il peut être envisagé d'avoir des mises à jour de la donné…

Réduire au maximum les intermédiaires

Plus vous ajoutez de couches intermédiaires, plus vous faites de "mappers" (et donc du boilerplate code). Bien entendu, ne pas réutiliser les mêmes structures de données entre votre base de données et votre couche métier fait sens, mais à quoi bon passer par des états intermédiaires au moment de passer sur le réseau ?

Si vous prenez l'exemple de GraphQL qui est très utilisé et permet d'avoir un contrat strict entre le serveur et le client, vous vous retrouvez à décrire un schéma dans un fichier `.gql` qui n'a rien à voir avec votre code (peu importe le langage). Vous devrez donc faire des mappers `métier ⇔ GQL`, et sur votre frontend, et sur votre backend. **Et vous perdez l'efficacité de "typer" de bout en bout votre application** (puisque GraphQL n'est pas un concept propre à JS/TS).

> **CONTEXTE**
>
> Nous recommandons d'utiliser un monolithe où les concepts sont très rapprochés, nous avons peu de raisons de nous ajouter une telle complexité.

Petite nuance nécessaire : GraphQL permet aussi de définir son schéma en code brut, ce qui à première vue règlerait le souci si c'est fait dans un même langage client-serveur. Mais l'architecture même de GraphQL apporte une (trop) grande complexité (resolvers imbriqués, et donc dataloaders, et donc query complexity…). C'est pour cela que le prochain chapitre mentionne une alternative.

tRPC

tRPC veut dire "TypeScript Remote Procedure Call". Vous l'aurez peut-être deviné mais cela va vous permettre de définir les contrats de vos endpoints d'API avec du TypeScript. Vous pouvez réutiliser vos structures métiers pour l'encodage et le décodage pour les communications API. Cela sera transparent pour vous et vous aurez de bout en bout typé votre backend jusqu'au frontend.

De plus, dès que vous modifierez ce que peut retourner votre serveur, votre linter sera en mesure de vous prévenir si certains appels clients sont dorénavant invalides (car ne respectant pas le contrat).

> **NOTE**
>
> Cela marchera que vous fassiez du tRPC depuis le frontend (suite à une action utilisateur), ou du tRPC via les RSC (composants rendus sur le serveur).

Permettre l'évolutivité

Notre parti pris est de faire du TypeScript de bout en bout, mais sachez qu'en utilisant tRPC vous ne vous bridez pas à simplement générer des clients TypeScript. Vous pouvez très bien faire de votre produit un "produit API" de qualité avec des clients API dans différents langages.

Comme vos endpoints sont typés avec tRPC, vous pouvez utiliser trpc-openapi afin d'exposer votre API sous le standard OpenAPI afin d'avoir une API REST. Ce standard vous permet même ensuite de générer des clients dans n'importe quel langage.

Validation des données entrantes

Que ce soit sur vos endpoints tRPC ou sur un endpoint à part dans Next.js pour recevoir des webhooks, la devise est de **toujours valider les données entrantes**.

Nous utilisons `zod` pour faire la validation au moment du runtime (quand les types ne peuvent être garantis sur les données entrantes). Cela nous permet une certaine flexibilité puisqu'il peut servir à la validation d'entrée d'un endpoint mais aussi pour formater une structure métier (dans le cas où vous voulez que ça émette une erreur si la structure récupérée depuis une base de données est mal formatée par exemple).

- Une pratique peut être de définir vos structures métiers avec `zod` puisque les types peuvent être "inférés" ;
- zod est nativement supporté par tRPC, donc vous n'avez qu'à mettre votre structure `input` au niveau du endpoint ;
- zod peut s'utiliser côté frontend avec `react-hook-form` afin de faire des validations sans même contacter le serveur. Ce qui est pratique dans notre cas c'est que la même structure de validation est utilisée dans les 2 mondes (avec ces textes d'erreurs qui peuvent être personnalisés).

entities/agent.tsx | Déclaration d'une structure d'entité

```
1  import z from 'zod';
2
```

```ts
import { UserSchema } from './user';

export const AgentSchema = z
  .object({
    id: z.string().uuid(),
    userId: z.string().uuid(),
    authorityId: z.string().uuid(),
    firstname: UserSchema.shape.firstname,
    lastname: UserSchema.shape.lastname,
    email: UserSchema.shape.email,
    profilePicture: UserSchema.shape.profilePicture,
    isMainAgent: z.boolean(),
    createdAt: z.date(),
    updatedAt: z.date(),
    deletedAt: z.date().nullable(),
  })
  .strict();
export type AgentSchemaType = z.infer<typeof AgentSchema>;
```

actions/agent.tsx | Validation d'une mutation avec les propriétés d'entités existantes

```ts
import { AgentSchema } from '@mediature/main/src/models/entities/agent';
import { AuthoritySchema } from '@mediature/main/src/models/entities/authority';
import { UserSchema } from '@mediature/main/src/models/entities/user';
import z from 'zod';

export const AddAgentSchema = z
  .object({
    userId: UserSchema.shape.id,
    authorityId: AgentSchema.shape.authorityId,
    grantMainAgent: z.boolean(),
  })
  .strict();
export type AddAgentSchemaType = z.infer<typeof AddAgentSchema>;
```

Préférez l'API à la mise en cache frontend

Qu'est-ce que ça veut dire ? Dans React, un outil comme Redux permet de gérer des "stores" pour gérer l'état global de votre frontend au fil des pages. Comme entre les pages vous allez probablement récupérer certaines mêmes données, il peut vite être tentant de chercher à économiser les requêtes réseaux en réutilisant ce qui a déjà été récupéré.

Sauf qu'en faisant cela vous vous lancez d'office dans plusieurs complications :

1. Vous ne garantissez plus du tout que la donnée que vous affichez est à jour au moment de l'affichage de la nouvelle page (puisque vous l'avez mise en cache pour une certaine durée). Faisant courir le risque que l'utilisateur fasse des actions sur des valeurs périmées ;
2. Vous répliquez la notion de cycle de vie de la donnée dans le frontend. Ce qui apporte une complexité dans l'implémentation.

Comme expliqué pour la mise en cache backend des données, il ne sert à rien d'essayer d'optimiser avant de rencontrer certaines limites. Et ce qui est sûr, c'est qu'un simple backend avec une base de données simple devrait largement supporter vos quelques premiers milliers d'utilisateurs sans essayer d'optimiser le frontend.

> **REMARQUE**
>
> Vous verrez dans le chapitre sur le framework backend que nous n'utilisons

même pas besoin de librairie tierce pour gérer l'état de nos composants frontends.

Avoir de la réactivité avec une API traditionnelle

Lorsque votre application autorise des interactions entre plusieurs utilisateurs (directement ou indirectement), il peut être envisagé d'avoir des mises à jour de la donnée en temps réel.

Cas avancé et plus complexe

J'ai une application bancaire et j'aimerais voir en temps réel dans mon application mon solde se mettre à jour si une transaction se produit. Il faut pour cela que votre backend notifie votre frontend des données à mettre à jour, ou tout simplement en envoyant un ordre similaire à "rejoue tes requêtes".

2 solutions s'offrent à vous :

1. Qu'à son lancement le frontend se manifeste auprès du serveur pour maintenir un canal "de souscriptions en temps réel" (que ce soit une WebSocket ou du long-polling). Cela peut se faire directement avec la librairie tRPC que nous conseillons ;
2. Il est aussi envisageable d'utiliser les Push notifications qui sont maintenant un standard mobile et web. À noter qu'elles font surtout sens pour permettre de notifier un utilisateur même lorsque l'application numérique n'est pas ouverte. Par exemple, si vous recevez un message dans votre applicatif, vous aurez une notification de votre navigateur web même si vous n'avez pas l'application ouverte. *Pour cette solution précisément, il est recommandé d'utiliser un service*

connu pour la distribution des Push notifications afin de vous simplifier la gestion des envois (bien que cela rajoute un acteur tiers).

Cas standard et simple

Vous pouvez aussi apporter de la réactivité frontend sans avoir de notification de la part du backend. Techniquement cela paraît très "sommaire" mais c'est terriblement ingénieux. Quelques exemples :

- Je soumets un formulaire, les requêtes *(queries)* sont automatiquement rejouées et la page met à jour ses données sans rechargement visible. Si vous utilisez une base de données atomique, les données soumises apparaissent directement sans devoir mettre en place des *patchs* manuels de l'interface. Cela est possible car c'est le même client qui gère les *queries* et les *mutations* ;
- Je vais sur un autre onglet pendant X minutes, je reviens, et tous les *getters* sont rejoués ;
- L'application peut aussi explicitement forcer le rafraîchissement des données sans rechargement.

Cela est possible suivant le client HTTP que vous utilisez. Nous avons fait le choix de `react-query` qui est directement compatible avec tRPC. Sans configuration explicite on bénéficie des exemples cités.

> **ASTUCE**
>
> Si vous n'avez pas d'API tRPC mais un format standard type OpenAPI, vous pouvez générer un client `react-query` grâce à des librairies comme `openapi-codegen` ou encore `orval`.

Une base de données qui fait tout (vraiment)

RDBMS… simplement du relationnel

Il faut utiliser du relationnel. Les bases relationnelles garantissent un schéma de données ainsi que l'affiliation entre plusieurs items de cette base.

ORM

Nous faisons le choix d'utiliser Prisma pour interagir avec notre base de données RDBMS. Comme nous souhaitons versionner le plus de choses, avoir un ORM permet d…

PostgreSQL plutôt que MySQL

Nous préconisons PostgreSQL pour du RDBMS. Cette base de données est orientée objets, ce qui permet de pousser un peu plus loin les contraintes sur les formats de …

Un seul espace de stockage, ça serait le rêve ?

Ce qui est le plus compliqué quand on gère de l'opérationnel ce n'est pas le stateless (normalement équivalent à votre code JS qui tourne), mais le stateful (partout où …

Stockage de fichiers (au revoir buckets S3)

Bien que le bucket S3 soit devenu une référence voire même le choix par défaut quand on commence un projet, on aimerait vous mettre en garde en redétaillant l'inté…

Job queues / message broker

Une "job queue" est utile :

Cache

Il ne sert à rien de chercher à optimiser son applicatif avant même d'avoir atteint les limites de la stack actuelle. Si vraiment vous avez des problèmes de performances…

Moteur de recherche

Certains produits auront peut-être besoin d'un outil lourd comme ElasticSearch pour stocker et optimiser les recherches à effectuer, mais encore une fois, dans la gran…

📄 Faire la maintenance

Le premier avantage est quand vous développez, plutôt que de lancer plusieurs services pour faire tourner un produit très simple, vous avez votre PostgreSQL et c'est t...

📄 Suis-je toujours un produit portable avec mon stateful ?

Nous préconisons toujours d'utiliser les produits de votre hébergeur lorsqu'il s'agit du stateful. Car l'hébergeur se chargera à votre place des mises à jour opérationnelle...

RDBMS… simplement du relationnel

Il faut utiliser du relationnel. Les bases relationnelles garantissent un schéma de données ainsi que l'affiliation entre plusieurs items de cette base.

Les plus connues ont d'énormes avantages :

- Elles sont disponibles "out-of-the-box" chez tous les hébergeurs qui proposent du stockage (utile pour la portabilité du produit) ;
- Elles peuvent être requêtées par un même langage (SQL) ;
- Transactions possibles à travers plusieurs tables ;
- Atomicité des opérations (lock) ;
- Table qui peuvent faire jusqu'à 32 TB ;
- Scalabilité ;
- …

Plusieurs articles sur internet font un retour d'expérience sur des produits web supportant **des millions d'utilisateurs en étant restés sur une technologie RDBMS**. Nous pensons que le jour où vous serez sur le point de gérer des millions d'utilisateurs, vous aurez une équipe et un budget dimensionnés pour que votre produit tienne la charge. En attendant, si vous avez des pics de charge, augmentez les caractéristiques de votre base de données (CPU, RAM, stockage). Et si votre produit est critique et ne doit jamais avoir de downtime, votre hébergeur propose sûrement de faire des "replicas" de votre base de données afin de former un cluster.

Plus récemment il y a eu une mode d'adoption sur les bases de données NoSQL. C'est des bases de données où la structure des données n'est pas garantie, et où souvent les opérations transactionnelles sont impossibles. **Le NoSQL est adapté à des fonctionnalités de niche (même pas à des produits).** Si vraiment vous pensez devoir utiliser du NoSQL car une RDBMS ne suffit pas, parlez-en à plusieurs personnes autour de vous en exposant le cas d'utilisation qui selon vous nécessite cette technologie.

Pour aller plus loin dans la réflexion, comme votre base de données (NoSQL) n'est pas capable de garantir un schéma de données, cela veut dire qu'à tout instant il peut y avoir n'importe quoi à l'endroit où vous allez chercher votre donnée. Peut-être que le `userB` possède les champs `prénom + nom + âge` alors qu'historiquement `userA` possède juste `âge` car il était là bien avant la prise en charge des noms. Vous vous retrouvez du coup à marcher sur des œufs **et finalement votre "schéma de base de données" est implicitement caché dans votre propre code métier puisque vous devez faire des vérifications sur la structuration de la donnée**.

ORM

Nous faisons le choix d'utiliser Prisma pour interagir avec notre base de données RDBMS. Comme nous souhaitons versionner le plus de choses, avoir un ORM permet de garder cette rigueur.

Sachez que Prisma a été développé avec TypeScript en tête. Le type des données retourné par les requêtes est dynamique en fonction de la requête. Par exemple si vous faites l'équivalent d'un `SELECT * FROM users` toutes les propriétés seront proposées par le linter TypeScript, alors que si vous faites un `SELECT age FROM users` seul l'âge sera proposé comme propriété pour chaque utilisateur.

> **RAPPEL**
>
> Le fait d'avoir une base de données avec des types garantis, qui se retrouvent dans le TypeScript, est dans la continuité de notre logique de typage "de bout en bout".

Un ORM a toujours ses limites si jamais vous voulez optimiser les requêtes... L'optimisation peut se faire en utilisant du SQL en direct mais vous perdez cette notion de types. Une alternative est de créer une `VIEW` en base de données, et avec son équivalence typée dans Prisma afin de garder la notion de typage strict.

PostgreSQL plutôt que MySQL

Nous préconisons PostgreSQL pour du RDBMS. Cette base de données est orientée objets, ce qui permet de pousser un peu plus loin les contraintes sur les formats de données contrairement à une MySQL/MariaDB.

À titre d'exemples comme MySQL n'est pas nativement orienté objets :

- Il a fallu attendre MySQL v8 en 2018 pour enfin supporter les `ENUM` (sachant que MySQL est sorti en 1995) ;
- La gestion d'une colonne en `uuid` n'est pas native :
  - Si vous voulez que ce soit lisible n'importe quand pour un humain il faut forcer le type à `string`… mais du coup ce n'est pas optimisé en stockage et pour les jointures ;
  - Si vous voulez optimiser la taille vous devez définir le type `BINARY(16)` mais au moment de faire vos `SELECT` vous ne verrez rien de lisible. Pour ça il faut préciser le helper `BIN_TO_UUID()` à chaque endroit où il y a vos UUIDs, et `UUID_TO_BIN()` au moment de vos `INSERT`. **Économisez votre temps… utilisez PostgreSQL** ;
- Vous ne pouvez pas créer de "custom type" où vous définiriez vous-même la structure de données attendue.

Cela peut vous paraître exagéré mais quand vous faites des migrations de données (car oui cela arrive), avoir une base qui sait un minimum ce qui peut ou ne peut pas être fait aide grandement. Surtout quand vous travaillez sur des Go de données.

Un seul espace de stockage, ça serait le rêve ?

Ce qui est le plus compliqué quand on gère de l'opérationnel ce n'est pas le stateless (normalement équivalent à votre code JS qui tourne), mais le stateful (partout où vous avez de la donnée stockée).

- Si votre stateless est bien conçu (gestion du SIGTERM, transactions de base de données…), qu'il y ait 1, 2, 3 instances voire même 0 cela ne perturbe en rien ce qui est vital : les données utilisateurs ;
- Si votre stateless est supprimé par mégarde, vous pouvez le repeupler et il remplira toujours sa fonction primaire ;

Par contre… si votre stateful est instable il y a un risque qu'il y ait une corruption de données (risque limité avec l'usage des transactions, mais ça peut arriver dans un fil d'exécution qui appelle différent outils stateful). Et si par mégarde vous le supprimez, il faut rentrer dans une longue investigation.

Par chance vous aurez mis en place des backups mais entre le dernier backup et la perte des données il va être très dur de reconstituer ce qui manque (voire impossible).

> **TOUT ÇA POUR DIRE QUOI ?**
>
> C'est déjà assez compliqué et stressant de gérer un outil stateful, alors il va de soi que si vous les multipliez vous ne faites pas un cadeau à la partie opérationnelle de l'équipe.

Dans les parties suivantes nous allons essayer de vous convaincre que se servir de votre PostgreSQL comme d'un couteau-suisse peut vous simplifier la vie.

Stockage de fichiers (au revoir buckets S3)

Bien que le bucket S3 soit devenu une référence voire même le choix par défaut quand on commence un projet, **on aimerait vous mettre en garde** en redétaillant l'intégration d'un bucket S3 au sein d'un applicatif.

Si vous utilisez du S3 vous allez normalement stocker vos fichiers dans un espace distinct de votre base de données. Chaque fichier aura une URL unique, et dans votre base de données vous garderez en référence cette URL. Vous vous retrouvez dans la situation où les deux peuvent ne plus être synchronisés, avoir l'URL dans votre base de données ne veut pas dire que le fichier existe sur votre bucket S3.

Sachez qu'une PostgreSQL est capable de gérer au sein d'une colonne des fichiers faisant maximum 1 Go chacun, il faut pour cela utiliser le type `BYTEA` (qui utilise le TOAST).

> **INFORMATION**
>
> Il existe aussi la possibilité de stocker des "large objects" qui peuvent faire jusqu'à 4 To chacun, il faut une étape supplémentaire pour les ajouter avant de les référencer dans une table traditionnelle. Et cela pose des questions de volumétrie qui seront abordés plus loin.

Vous devez spécifier et quantifier ce que vous allez héberger pour dire oui ou non aux buckets S3.

Un élément important qui est mentionné dans le chapitre légal et qui doit être pris en compte dans votre calcul : **la CNIL recommande de ne pas conserver les données utilisateurs plus de 2 ans**.

Cas concrets :

1. Si je sais que mon produit ne va héberger que des photos de profil des agents publics et les logos de toutes les collectivités de France. Je sais d'avance que la totalité des fichiers ne dépassera jamais une certaine limite. **Ma PostgreSQL suffit** ;
2. Si mon outil reçoit des demandes de médiations et que chaque citoyen et agent public peut ajouter à chaque demande des pièces justificatives (limitées à 5 Mo chacune), a priori il n'y a pas de limite (cela peut être exponentiel). Sauf que l'on sait historiquement qu'il y a en moyenne 5 fichiers par dossier, et 10 000 dossiers créés par an. Comme l'outil supprime les dossiers après 2 ans pour respecter la recommandation CNIL, le stockage reste acceptable et une **PostgreSQL devrait suffire** ;
3. Si mon outil archive les médias réalisés durant des événements publiques, j'accepte ainsi des photos en haute qualité (10 Mo ou plus) et des vidéos qui peuvent faire plusieurs heures (plusieurs Go). De plus, du fait de la volonté d'archiver la recommandation CNIL n'est pas appliquée. Je me retrouve avec un enjeu stratégique sur le stockage, **ma PostgreSQL n'est pas adaptée, je choisis du S3**.

Points importants (hors problématiques de stockage) :

- Nous devons rester sur des tailles de fichiers acceptables quand on passe via PostgreSQL car c'est notre propre API qui sera sollicitée pour servir le flux ;
- En utilisant une base PostgreSQL pour le stockage **il ne faut jamais essayer de produire vous-même la logique d'upload d'un fichier**. Cela implique beaucoup de technique : headers HTTP, le "resumable upload" (fait de reprendre une upload si la connexion internet est perdue durant quelques secondes)… **Nous**

préconisons plutôt d'utiliser `tus` (client et serveur) sachant qu'ils proposent une intégration dans l'API Next.js ;

- **Si vous utilisez des buckets S3, ne faites pas transiter les fichiers par votre API.** Vous avez sûrement choisi S3 car il est question de volumétrie importante. Le fait d'utiliser votre API en intermédiaire met un point de congestion et fait reposer sur vous le fait de bien dimensionner votre API alors qu'une upload/lecture peut être anecdotique (**sachant que ça peut en plus perturber vos opérations sensibles sur l'API**). Nous recommandons dans ce cas de choisir un fournisseur S3 qui permet de "signer des URLs" que ce soit en lecture ou en upload. Avec cette architecture aucun fichier ne transite via votre API :
  i. Votre frontend demande à l'API "je veux uploader un fichier" ;
  ii. L'API renvoie une nouvelle URL signée du provider S3 ;
  iii. Le frontend peut uploader son fichier directement sur l'URL signée (en utilisant une librairie cliente compatible comme `tus`) et obtient l'URL finale du fichier uploadé ;
  iv. Le frontend peut transmettre l'URL du fichier uploadé à l'API ;
  v. L'API est en mesure de valider le fichier de façon synchrone (ou non), et d'enregistrer l'URL en base de données.

Notez que **les technologies comme `tus` ou S3 ne gèrent pas les règles de sécurité propres à votre métier** (qui peut uploader ? qui peut lire ce fichier ?). C'est à vous de gérer cette partie. Nous abordons cela plus en détail dans le chapitre sur la sécurité.

> **INFO**
>
> Écrire et lire des fichiers dans votre base de données va forcément utiliser un peu plus de ressources (notamment la bande passante). N'hésitez pas à "augmenter" les performances de votre base si besoin, et à dimensionner le tuyau d'entrée pour faire passer les fichiers :

Next.js **Nginx**

Il faudrait augmenter la configuration dans le fichier du endpoint d'upload en exportant par exemple :

my-endpoint.tsx

```
export const config = {
  api: {
    bodyParser: {
      sizeLimit: '10mb',
    },
  },
};
```

Il faudrait augmenter la configuration :

/etc/nginx/http.d/default.conf

```
server {
  listen 80;
  client_max_body_size 10M;
}
```

Job queues / message broker

Une "job queue" est utile :

- Pour faire un travail en asynchrone (et ne pas bloquer la requête courante qui est synchrone) ;
- Répartir la charge : les tâches en attente peuvent être exécutées 1 par 1 ;
- En cas d'échec on peut automatiquement rejouer le "job" jusqu'à temps que ça passe ;

> **OMISSION**
>
> Dans cette partie nous allons omettre le pattern d'event sourcing et de CQRS car nous estimons que les appliquer à tout son applicatif apporte une certaine complexité dans le développement. Étant dans un monolithe avec un seul stockage, et pouvant bénéficier de transactions SQL, vous devriez pouvoir passer outre (et donc nous ne parlerons dans la suite de "job queue", pas de "message broker").

Il ne faut pas essayer de tout garantir avec asynchronisme, vous risqueriez de complexifier votre outil (et donc sa phase de débogage). Voici quelques cas concrets :

- Quand l'utilisateur envoie un formulaire, aucune raison de mettre cette action dans une file d'attente pour modifier la base de données, si quelque chose

échoue il est acceptable que l'utilisateur reçoive une erreur serveur afin de retenter ;

- Si vous devez faire une action métier sensible comme créer une transaction financière chez un partenaire, en faire un "job" permettra de vous garantir que si votre partenaire est hors service, la transaction sera quand même créée une fois que le partenaire sera de nouveau sur pieds ;
- Si mes partenaires peuvent me notifier d'événements métiers indispensables à mon produit, il vaut mieux mettre le payload HTTP reçu (souvent du JSON) directement dans une "job queue" même si vous le traitez immédiatement. Cela permettra :
  i. Si vous avez une erreur temporaire, de pouvoir rejouer le webhook (très souvent les partenaires n'ont pas de logique de "retry" poussée donc il serait risqué de se reposer sur eux) ;
  ii. Si jamais votre partenaire change de format de données dans son payload, on sait d'avance que cela ne passerait pas de votre côté car vous validez la donnée en entrée, il vaut mieux avoir le payload complet utilisé pour investiguer et savoir ce qu'il faut ajuster.

Nous préconisons d'utiliser la librairie `pg-boss` en lieu de place d'outils dédiés comme "Redis Pub/Sub", Kafka (...). Elle va savoir reproduire toute la logique des "job queues" en ayant quelques tables dans votre base de données. De plus :

- Elle fait très bien le travail pour un usage normal ;
- Cela vous évitera de devoir le maintenir un outil stateful supplémentaire ;
- Cela vous évitera de devoir déboguer via plusieurs outils à la fois ;
- Quand vous ferez un backup ou un "restore" de votre base de données, les événements seront synchronisés avec l'état global des données de votre produit.

> **INFORMATIONS ANNEXES**
>
> - `pg-boss` sait gérer les cron jobs, donc plutôt que d'utiliser une

fonctionnalité propre à votre provider (qui ne garantit souvent pas l'unicité de l'opération), vous pouvez vous en servir ;

- Next.js ne permet *pour l'instant* pas de lancer directement des processus asynchrones (il faudrait modifier le serveur web mais on perdrait l'optimisation qu'ils ont fait). Notre approche est d'utiliser un endpoint `/init` qui sera appelé au lancement du serveur ;
- À vous de choisir votre stratégie de rejeu en fonction de la criticité du job :
  - N'oubliez pas de spécifier un maximum de rejeux pour un job car certains ne passeront jamais et il serait dommage de flooder votre système ou vos partenaires (sans jamais traîter les jobs suivants, qui eux, peuvent peut-être passer) ;
  - Vous pouvez vous reposer sur Sentry pour être notifié d'un job qui est abandonné afin que vous puissiez investiguer et tenter de le rejouer manuellement.

Cache

Il ne sert à rien de chercher à optimiser son applicatif avant même d'avoir atteint les limites de la stack actuelle. Si vraiment vous avez des problèmes de performances essayez d'abord de cibler l'origine du problème :

1. Si c'est l'applicatif JS qui est trop juste en CPU/RAM, essayez d'augmenter sa gamme chez votre provider ;
2. Si c'est de la latence sur le réseau, assurez-vous que l'applicatif et la base de données soient dans la même zone géographique (même datacenter) ;
3. Si c'est la base de données, ajustez aussi sa gamme (CPU/RAM) voire passez sur un stockage SSD plutôt que HDD.

Souvent on met en cache en utilisant des bases `key / value` comme Redis afin de stocker en clef l'URL `GET` et en valeur le payload HTTP. Le problème est que cela apporte énormément de complexité logique.

- Quand mettre en cache ?
- Quand invalider le cache ?
- Quand je débogage, est-ce une donnée à jour ?

Il est bien plus facile dans un premier temps d'augmenter les performances de nos outils car ça se fait en un clic. Vous vous économiserez plutôt que de gérer un outil externe.

Moteur de recherche

Certains produits auront peut-être besoin d'un outil lourd comme ElasticSearch pour stocker et optimiser les recherches à effectuer, mais encore une fois, dans la grande majorité des cas votre PostgreSQL suffira.

Nous ne vous proposons quand même pas de retourner à l'opérateur SQL `LIKE %ma recherche%` 👴, mais plutôt d'utiliser les types de données `tsquery` et `tsvector`. Si comme conseillé vous utilisez l'ORM Prisma, l'utilisation est assez facile et quasi transparente sur des colonnes distinctes. Par contre cela ne marchera pas si vous cherchez "John Doe" et que "John" est dans la colonne `firstname` et "Doe" dans la colonne `lastname`.

Pour faire ce type de recherche sur plusieurs colonnes (exemple : le prénom, le nom, l'adresse, la biographie), la stratégie est de concaténer chaque colonne voulue dans une colonne générée et indexée, représentant un `tsvector`. Cette colonne est donc optimisée pour de la recherche.

Prisma ne gère pas encore nativement les colonnes générées, mais rien ne vous empêche de la créer à la main. Différents articles existent sur internet mais pour vous donner une idée cela donnerait quelque chose comme :

```
ALTER TABLE MyTable
ADD search TSVECTOR
GENERATED ALWAYS AS (
  to_tsvector('english',
    firstname || ' ' || lastname || ' ' || biography || ' ' ||
full_address
  )
```

```
7  ) STORED;
```

Plutôt que de juste concaténer bêtement les colonnes, on transforme le tout en `tsvector` afin qu'il soit déjà optimisé pour la recherche.

> **À SAVOIR**
>
> Un `tsvector` est optimisé pour une langue en particulier afin de trier les mots. Par défaut, c'est la configuration `default_text_search_config` qui est utilisée. Sachez que même si votre contenu est en français, faites des tests avec `english` avant d'utiliser la valeur `french`. Cette dernière différencie les mots avec ou sans accent, ce qui peut être problématique… N'hésitez pas à jeter un œil à toutes les options de langues possibles.

Faire la maintenance

Le premier avantage est quand vous développez, plutôt que de lancer plusieurs services pour faire tourner un produit très simple, vous avez votre PostgreSQL et c'est tout.

L'autre avantage d'avoir juste votre PostgreSQL pour gérer les données métiers, les fichiers et le queueing, c'est que quand vous avez une erreur, vous pouvez déboguer la situation depuis un seul et même endroit. Pareil pour restaurer des backups, tout se fait durant la même opération.

Comme mentionné pour la documentation de la base de données, nous utilisons le client DBeaver au quotidien.

Suis-je toujours un produit portable avec mon stateful ?

Nous préconisons toujours d'utiliser les produits de votre hébergeur lorsqu'il s'agit du stateful. Car l'hébergeur se chargera à votre place des mises à jour opérationnelles et vous garantira un minimum de disponibilité de service. Pour rappel, ce n'est pas votre métier… Alors que votre hébergeur, lui, doit faire une palanquée de tests pour être sûr que quand il lancera une migration, cela ne mettra pas en défaut sa dizaine de milliers de clients.

Dans le cas où vous complexifiez votre stack stateful, et que chez un provider vous utilisez des buckets S3 et un message broker managé, on se retrouverait dans la situation où :

- Leur message broker pour le queueing est des fois une solution "homemade" et n'est pas compatible si vous migrez sur un autre hébergeur ;
- Si jamais ils proposent une solution grand public de message broker, vous vous retrouveriez sûrement avec du "Redis Pub/Sub" ou du Kafka, et ça implique que vous gériez vous-même la complexité de ces outils ;
- Tous les hébergeurs ne proposent pas de buckets S3. Pire encore, même ceux qui les proposent n'ont pas les mêmes services à proposer (signature d'URL pour la lecture ou l'upload, planifier l'auto-suppression de fichiers…).

C'est pour cela que nous conseillons de tout faire tenir dans votre PostgreSQL

quand c'est adapté à votre produit.

Un hébergeur simple et souverain

📄 Simple ? Explications

Les principaux enjeux ont déjà été mentionnés dans les précédents chapitres (dont celui sur la portabilité d'un produit).

📄 Données sensibles

En tant que produit vous allez normalement héberger un minimum de données personnelles. Vous ne pouvez donc pas prendre un hébergeur lambda qui ne répond pa…

📄 Présentation de quelques acteurs

Rappelons que vous devriez choisir des hébergeurs français 😇.

📄 Se décharger des outils non sensibles

Les hébergeurs applicatifs vont vous faciliter la vie pour voir les logs de votre application, gérer les ressources… Pourtant il y a certains outils annexes qui sont très dema…

Simple ? Explications

Les principaux enjeux ont déjà été mentionnés dans les précédents chapitres (dont celui sur la portabilité d'un produit).

Nous préconisons de choisir chez un hébergeur ce qui pourrait se référer au "serverless" pour votre applicatif, ce afin de juste avoir à jouer sur le nombre d'instances ou les ressources disponibles pour répondre à votre besoin. Nous parlons ici de serverless sur des "images finales" dont la technologie est open source (comme Docker ou Buildpack), et non pas de FaaS (functions as a service).

> **REMARQUE**
>
> Les FaaS répondent à un besoin sommaire (comme un script), mais rarement au besoin d'un produit complet (sachant qu'un ensemble de "functions" indépendantes poserait un problème de synchronisation des versions du runtime comme mentionné ici).

Il nous semble par exemple démesuré de gérer la complexité de ressources sur un cluster Kubernetes pour gérer un produit. Bien évidemment certaines entités ont déjà de tels clusters en place et on vous redirige vers la validité de ce guide pour ne pas être en porte-à-faux vis-à-vis de votre entité. Gardez juste en tête que le coût opérationnel (temps / argent) pour maintenir de tels outils est bien supérieur au fait de payer un hébergeur tiers. D'ailleurs ceux qui sont déjà passés par-là ont peut-être remarqué que finalement on refait le travail d'un hébergeur traditionnel, sauf que l'on a pas le budget pour (ni la légitimité pour 😊), et on arrive difficilement à un résultat qualitatif (autonomie des équipes, documentation, outils de monitoring...).

Données sensibles

En tant que produit vous allez normalement héberger un minimum de données personnelles. Vous ne pouvez donc pas prendre un hébergeur lambda qui ne répond pas à un certain nombre de critères de confiance. Que ce soit légalement ou d'un point de vue sécurisation des données.

Pour faire simple, si l'hébergeur affiche les standards suivants vous partez dans une bonne voie :

1. SecNumCloud ;
2. HDS ;
3. ISO 27001 (grand minimum).

À savoir qu'en avril 2023 l'État annonçait qu'il faut que tout hébergeur se conforme prochainement au SecNumCloud pour continuer à héberger des services publics. Cela ne sera pas immédiat puisque certains hébergeurs ont un panel de produits assez large à adapter... Mais si votre hébergeur est HDS c'est déjà un très bon gage de sécurité.

ANNUAIRE

L'ANSSI référence les hébergeurs SecNumCloud, mais cela ne liste pas les petits hébergeurs qui sont basés sur de plus gros (ils ne gèrent pas l'accès physique aux bâtiments...).

Présentation de quelques acteurs

Rappelons que vous devriez choisir des hébergeurs français 😎.

Voici quelques exemples d'hébergeurs qui proposent du "serverless" pour les applicatifs :

Nom	Quelle offre ?	Container final à configurer
Scalingo	Offre standard	Buildpack
CleverCloud	Offre standard	Image Docker **ou** directement du JavaScript
Scaleway	"Serverless > Containers"	Image Docker
OVH	"Web Cloud > Web PaaS (partenariat Platform.sh)"	Directement du JavaScript

Liste non exhaustive et sans favoritisme puisque Sillon est décrit par une équipe et n'est pas une vision globale de l'État.

Se décharger des outils non sensibles

Les hébergeurs applicatifs vont vous faciliter la vie pour voir les logs de votre application, gérer les ressources… Pourtant il y a certains outils annexes qui sont très demandés comme Sentry pour du monitoring d'erreurs. Cet outil propose une offre SaaS mais celle-ci dépend d'un hébergeur étranger. Vous pourriez tenter de l'héberger vous-même mais ça revient à complexifier votre stack (surtout qu'il faut que votre hébergeur accepte une image Docker/Buildpack sinon cela risque d'être compliqué).

Comme c'est un outil "non vital" pour votre produit nous recommandons de mutualiser la gestion d'un tel outil :

1. Soit dans votre entité quelqu'un l'a déjà mis en place, demandez à ce qu'ils créent un sous-projet pour que vous ayez un identifiant ;
2. Soit vous pouvez vous adresser à une entreprise qui fait du pseudo-SaaS de cet outil comme Empreinte Digitale (mais il existe sûrement d'autres acteurs) ;
3. … Ou vous jugez que vous pouvez complètement vous en passer.

Dans la situation (1) et (2), comme vous propagez de la donnée à l'extérieur d'outils que vous gérez, il est important de la nettoyer au maximum. À vous de faire en sorte qu'aucune donnée utilisateur ne soit remontée, ni prénom/nom, ni email…

Authentifier les utilisateurs

Vous pouvez adopter plusieurs stratégies en fonction de votre produit (ce choix découle de l'investigation, ce n'est pas un choix technique) :

1. Avoir votre propre système d'enregistrement et connexion ;
2. Utiliser un système de connexion connu comme FranceConnect ;
3. Utiliser les 2.

Nous préconisons d'utiliser `next-auth` au sein de Next.js pour répondre à n'importe lequel de ces cas de figure.

La (1) vous permet :

- D'aller vite ;
- D'avoir un comportement similaire en local, développement et production ;
- De ne pas reposer sur un système tiers qui peut être indisponible.

La (2) :

- Faciliter la connexion à vos utilisateurs ;
- Reposer sur un standard sécurisé comme OpenID Connect (OIDC), ou SAML pour des systèmes plus historiques.

Pour la (1) si vous ne voulez pas alourdir le développement et votre expérience utilisateur, utilisez la plus basique des connexions directement depuis votre applicatif (cela marche depuis des années et ça n'en est pas "moins sécurisé"). Assurez-vous juste d'avoir dans votre base de données des mots de passes hashés avec l'algorithme `bcrypt` (10 itérations minimum). *Les raisons de déployer vous-même un outil lourd compatible OIDC comme Keycloak ou Hydra peut être*

approprié si vous gérez une multitude de frontends distincts et que vous voulez avoir un seul compte commun pour les gouverner tous…

Pour la (2) c'est directement des initiatives du service public, pour détailler les principaux :

- FranceConnect ;
- AgentConnect ;
- ProSantéConnect ;
- MonComptePro ;
- Aidants Connect (pas listé dans la comparatif, mais à titre informatif).

Vos utilisateurs	MonComptePro	FranceConnect	AgentConnect	ProSantéConnect
Particuliers	❌	✅	❌	❌
Professionnels	✅	❌	❌	❌
Professionnels de santé	❌	❌	❌	✅
Entreprises	✅	❌	❌	❌
Agents de l'administration centrale	✅	❌	✅	❌
Agents des collectivités territoriales	✅	❌	❌	❌

Comparatif fait par MonComptePro (*source*)

L'intégration et le déploiement continu (CI/CD)

Nous préconisons d'utiliser une pipeline de CI/CD décrite dans votre repository. Si votre repository est par exemple sur GitHub nous ne voyons pas l'utilité d'utiliser un outil tiers comme CircleCI plutôt que GitHub Actions (comme ça au moins vous avez tout dans un même endroit).

Sauf exception, **l'important est de ne surtout pas essayer de gérer vos propres workers (serveurs) de CI/CD**. Pourquoi ? Votre pipeline procède normalement ainsi :

1. Je récupère le code source ;
2. Je télécharge les dépendances ;
3. Je "lint" le projet (pour être sûr qu'il est statiquement valide) ;
4. Je fais mes tests unitaires avec Jest ;
5. Je "build" mon projet ;
6. Je fais mes tests d'intégration end-to-end avec Cypress *(on ne les recommande pas, ils sont très lourds à maintenir pour un gain limité comparé à utiliser Storybook pour décrire son applicatif)* ;
7. Je déploie mon applicatif sur l'hébergeur ;

Toutes ces phases sont très consommatrices en ressources (CPU, mémoire, stockage, réseau). Sachant que si vous faites plusieurs `push` à la suite vous lancez

plusieurs fois la même pipeline, et sachant aussi que vous avez peut-être plusieurs projets sur vos workers :

- Il faut gérer proprement un cache pour les dépendances en fonction de chaque technologie de projet *(dans notre cas ça se limiterait à JavaScript)* ;
- Si les CPU saturent tout va lentement, exceptionnellement si vous êtes sur Kubernetes et que vous avez un opérateur, vous allez avoir un nouveau worker qui va être mis sur pied (mais ça va prendre quelques minutes) ;
- Si la RAM sature, là c'est le worker qui crash car… "Out of memory", il faut relancer la pipeline ;
- Et le pire dans tout ça, c'est que vous pouvez essayer de dimensionner les workers en fonction d'une utilisation intensive, mais finalement vous vous retrouvez avec des machines qui servent réellement que 5% du temps ou moins.

> **ASTUCE**
>
> Si vous disposez d'une messagerie interne comme Mattermost ou Slack nous vous conseillons de brancher la pipeline de votre projet sur un channel type `monprojet-monitoring` afin d'être sûr de ne rien manquer de la vie technique du projet.

Avoir différents environnements

Combien nécessaires ?

Production

Cela va dépendre de votre produit et de vos préférences. Par exemple, quand vous rédigez un article sur un blog, il est fort probable que vous fassiez vos modifications directement sur ce qu'on pourrait appeler "environnement de **production**". Et cela s'y porte très bien, car l'outil reste peu complexe et la prise de risque est donc faible.

Development

Quand vous faites du développement informatique (avec une équipe de plus d'1 personne), il y a un moment où vous voulez normalement montrer ce qui a été développé à des personnes associées au projet, aussi pour qu'elles testent avec des vérifications manuelles. Il est peu probable que des personnes non-techniques téléchargent le code technique, installent ce qu'il faut (base de données…), pour enfin lancer le produit localement et pouvoir le tester. Le plus simple est donc de leur mettre à disposition un environnement de **développement** (ou de **test**, suivant votre nommage).

Il peut être intéressant de le différencier de la production via un bandeau dans le header pour éviter de malencontreuses actions. Et aussi de sensibiliser les testeurs

sur le fait d'utiliser des données "fake" non-sensibles car cet environnement peut être manipulé par plus de monde que la production.

Aussi, si le code est open source et que personne n'utilise ses données personnelles, il n'est pas nécessaire de vous embêter à le mettre derrière un proxy type `oauth2-proxy`. Pour la sécurité :

- Il est surtout important de ne pas utiliser les mêmes instances (applicatif et stockage) que la production en cas d'infiltration ;
- Pour mitiger le risque de données sensibles oubliées dedans vous pourriez tous les X mois faire un reset avec un jeu de données de test.

> **REMARQUE**
>
> L'environnement de développement peut cohabiter avec des environnements éphémères (par fonctionnalité développée, souvent appelés "review apps"), voire même être remplacé par ceux-ci.

Staging

Certains projets ont un environnement de **recette** intermédiaire entre `development` et `production`, qui est censé représenter la production sans pourtant l'être, avec des jeux de données de production anonymisés par exemple. Ce qui veut dire qu'une modification doit passer `development → staging → production`, rajoutant une étape de test qui n'est pas forcément soutenable pour une équipe.

L'utilité du `staging` va surtout dépendre de la criticité de votre projet. Si vous pouvez vous permettre de parfois pousser des erreurs en ayant l'agilité de les corriger rapidement, n'avoir que `development + production` permet de rester simple.

Comment ça s'organise ?

Les environnements sont souvent différenciés par des branches Git différentes, qui, à chaque commit, seront déployées.

Il y a pas mal de standards pour gérer à la fois les environnements "runtime", ainsi que les branches correctives ou de fonctionnalités (Gitflow, GitHub flow, Gitlab flow…). Nous choisissons Gitflow comme base en l'adaptant en fonction des phases d'avancement du projet (peut-être pas besoin des branches `release`, peut-être pas besoin de branches de `feature` si qu'un seul développeur…). **L'important est d'être à l'aise pour éviter de perdre du temps, conciliez-vous au début du projet.**

Les tests et la limite du mock

Nous ne sommes pas pour imposer des tests à tous les niveaux et sur tout votre code applicatif. Il est beaucoup plus important selon nous de :

- Démontrer que votre produit sert réellement en répondant aux besoins utilisateurs (et donc en développant des fonctionnalités) ;
- D'avoir un maximum de flexibilité dans la maintenance de votre outil, car même en testant, il vous arrivera des pépins auxquels vous n'auriez jamais pensés.

Essayez de focaliser vos tests unitaires sur des briques critiques qui gèrent de la donnée métier (validation d'un numéro de téléphone, génération d'une URL signée pour un fichier…). Après si vous êtes à l'aise et efficace avec le TDD ou le BDD, bien entendu il faut continuer les pratiques qui marchent ☺.

De toute façon avec le temps votre pourcentage de couverture en tests va progresser, il est probable qu'à chaque bug remonté il soit plus facile de le tester de manière isolée de l'applicatif (impliquant de faire des tests unitaires pour facilement débugger).

Par contre, quand on veut tester à un niveau plus abstrait on se retrouve souvent à faire des mocks qui ne veulent plus rien dire puisqu'on les a fait spécifiquement pour que le test concerné passe. Par exemple, si l'on veut tester un code qui a des interactions avec la base de données, on peut essayer de mocker la base, mais ça ne nous dira jamais si finalement la requête de jointure ou les `SELECT` imbriqués auraient marché dans une vraie situation. Il existe pourtant bien quelques librairies

qui essaient de faire "database in memory" mais ça ne reste qu'une pseudo-copie où vous n'aurez pas les vraies problématiques de l'outil.

> **ASTUCE**
>
> Si vous souhaitez faire des tests d'intégration, nous préconisons de faire des tests en restant local à votre code, avec toute la logique dans votre test (comme pour des tests unitaires). Pour cela nous utilisons `test-containers` qui va communiquer avec votre Docker local pour peupler de façon éphémère par exemple une PostgreSQL, un `mailcatcher`… afin de faire des tests qui reproduisent un minimum l'environnement de production. Comme l'image d'un container est en cache après le premier test, peupler un container prend à peine quelques secondes et cela devient totalement transparent.
>
> Le processus marche aussi dans les CI/CD (en tout cas sur GitHub Actions) puisque le runner Docker est disponible via son API HTTP, donc vos tests sont en mesure de dire à la pipeline "peuple un container, puis éteins-le". Plus besoin de spécifier en dur dans la pipeline les potentielles dépendances de services 🙌.

Backups

Le code source

Dans le chapitre sur les forges logicielles nous mettions en avant code.gouv.fr afin d'y indexer le compte parent du repository (si celui-ci est public). Sachez que l'INRIA ...

3-2-1 : stateful !

La mise en place de backups est primordiale pour assurer la continuité d'un service en cas de dégât technique, inattention ou piratage.

Plan à suivre en cas d'incident

Le DRP (disaster recovery plan) est un document qui vous permet d'être un minimum organisé le jour où vous essuierez un plâtre. Le format est standardisé mais il faut ...

Le code source

Dans le chapitre sur les forges logicielles nous mettions en avant code.gouv.fr afin d'y indexer le compte parent du repository (si celui-ci est public). Sachez que l'INRIA a développé le projet Software Heritage afin d'archiver les codes sources, et ils ont comme référence code.gouv.fr. En y étant référencé vous n'avez donc pas à vous en préoccuper.

Bien entendu ça ne peut pas archiver les configurations privées du repository (celles pour la CI/CD…). Mais si vous avez documenté chaque étape de configuration dans votre `README.md`, vous saurez ainsi remettre sur pied rapidement votre repository dans les rares éventualités où :

- Vous souhaitez passer sur une autre forge ;
- La forge que vous utilisez a perdu vos données et les backups de vos données 🔥.

3-2-1 : stateful !

La mise en place de backups est primordiale pour assurer la continuité d'un service en cas de dégât technique, inattention ou piratage.

Nous préconisons le pattern 3-2-1 :

- 3 copies de la même donnée (l'original et 2 copies) ;
- 2 types de stockage différents ;
- 1 copie "hors site" ;

> **REMARQUE**
>
> Dans le cas où votre "stateful" est seulement dans PostgreSQL cela va vous simplifier la vie car vous n'aurez besoin que de faire un dump de la base de données. Si par contre vous avez un outil comme S3, vous pouvez utiliser des outils comme rclone pour synchroniser votre S3 original avec un S3 de backup (en désactivant les synchronisations immédiates de suppression).

Si l'on transpose ce pattern à notre cas :

- La donnée originale est dans PostgreSQL ;
- L'hébergeur fait automatiquement des backups de la base PostgreSQL et les garde quelques jours ou quelques semaines (✅ pour les 2 types de stockage) ;
- On fait régulièrement tourner un dump via un "cron job" pour faire un backup de la donnée originale et la mettre en sûreté dans un S3 chez un autre hébergeur sur une région différente (✅ pour les 3 copies distinctes et ✅ pour avoir 1 copie sur une localisation différente).

ASTUCE

Idéalement pour la copie n°3 on fait une copie du backup de l'hébergeur afin de ne pas surcharger la production, mais certains hébergeurs font des backups "point-in-time recovery" qui ne sont pas récupérables.

SÉCURITÉ

Pour la copie hors site il est préférable que le "job" tourne à l'extérieur de l'hébergeur de la production. Pour illustrer :

- Le job a accès en lecture sur la production ;
- Le job a accès en lecture et écriture sur le S3 "hors site" ;
- Si l'environnement de production est compromis par un piratage, on est sûrs qu'ils n'ont pas accès à l'environnement "hors site". Et s'ils ont accès au job, ils n'ont pas les droits pour détruire la production.

Plan à suivre en cas d'incident

Le DRP (disaster recovery plan) est un document qui vous permet d'être un minimum organisé le jour où vous essuierez un plâtre. Le format est standardisé mais il faut prendre en compte votre contexte : si vous êtes un produit isolé du reste de l'entité, avec des outils simples, gardez votre document simple. Il ne faudrait pas rentrer dans "la rédaction d'un process compliqué juste pour faire du process".

Si on prend l'exemple des backups (car le stateful reste le plus critique), il est quand même utile d'avoir noté quelque part que vous avez fait le choix d'avoir la copie n°3 sur l'hébergeur "X". Car avec les années peut-être vous ne serez plus dans l'équipe, et si personne ne sait où chercher… c'est là où ça risque d'être compliqué.

Design patterns

Nous n'avons pas de design pattern à vous préconiser d'un point de vue logiciel. C'est aussi pour cela que l'on ne fournit pas de "stack toute faite à copier/coller". Ce afin de vous laisser de la flexibilité quant à l'organisation de votre propre projet.

Notre guide essaie d'apporter des réponses quand vous ne les avez pas, mais nous ne nous substituons en rien à l'expertise de votre équipe.

On vous a précédemment mis en garde concernant des patterns comme l'event sourcing, le CQRS... C'est principalement pour que vous essayez de garder les choses les plus simples et les plus accessibles pour vous et de nouvelles recrues. Nous ne remettons pas en doute que ces patterns peuvent peut-être être utiles une fois que votre produit aura pris des proportions à faire rougir (mais ce jour-là, vous ressentirez vous-même le besoin d'adapter l'architecture).

Sécurité primaire

Sensibiliser

Peu importe votre rôle dans l'équipe, si vous avez un minimum de connaissances en cybersécurité, n'hésitez pas à vous assurer que chacun fasse le b.a.-ba.

Chiffrement SSL par défaut

Si vous avez choisi notre stratégie d'hébergement votre hébergeur vous permet en un clic d'activer la génération d'un certificat SSL afin de disposer d'une utilisation H...

Robustesse des mots de passe

Les pirates ont des bases de données de mots de passe déjà utilisés. Donc s'ils vous ont dans le collimateur, plutôt que de tester toutes les combinaisons imaginables p...

Gestionnaire de mots de passe

Il faut utiliser des mots de passe différents sur chaque service que vous utilisez. Pourquoi ? Car si des pirates trouvent vos identifiants pour un site, ils auraient juste à te...

2FA / Security keys

Le problème d'une authentification avec un simple mot de passe, c'est que si quelqu'un vous voit l'écrire dans le train, ou vous le vole en étant à l'autre bout du monde...

Accès restreints et alertes

Les accès à donner pour chaque service, outil ou fichier doivent être préalablement réfléchis. Pour aller droit au but : dites-vous qu'au niveau d'un produit, l'environne...

RSSI

Peu importe à quelle entité vous êtes rattaché, vous avez sûrement dans votre hiérarchie un RSSI (responsable de la sécurité des systèmes d'information). Il est judicieu...

Surveiller les CVE

Les CVE (~ failles de sécurité) sont disponibles via plusieurs flux RSS comme ceux de l'ANSSI (exemple de celui qui rassemble tout). Comme beaucoup d'informations p...

📄 En cas de fuite de données

Si vous constatez une fuite de données, le plus important est :

📄 Audits de sécurité

L'ANSSI a lancé le service MonServiceSécurisé afin de permettre une autoévaluation sur les principes de sécurité mis en place dans votre produit. C'est sous forme de ...

📄 Configurer les headers HTTP

Les headers HTTP venant de votre serveur vont dicter aux navigateurs internet comment ils peuvent agir (ou non). Certains doivent être réglés pour limiter les risques li...

📄 Sécuriser les URLs des fichiers hébergés

Que nous utilisions un hébergement de fichiers en base ou via des buckets S3, il faut garantir que les URLs de nos fichiers sensibles soient éphémères. Même si vous ave...

📄 Se prémunir des fichiers infectés

C'est malheureux mais il faut toujours se méfier des données qui entrent dans votre applicatif, encore plus quand il s'agit de fichiers. Quelques exemples :

Sensibiliser

Peu importe votre rôle dans l'équipe, si vous avez un minimum de connaissances en cybersécurité, n'hésitez pas à vous assurer que chacun fasse le b.a.-ba.

Nous allons détailler dans ce chapitre quelques points en particulier mais n'oubliez pas le plus rudimentaire :

- Mettre à jour son ordinateur et ses logiciels ;
- Faire attention au phishing ;
- Verrouiller son ordinateur dès que je m'en éloigne ;
- Ne pas écrire ses mots de passe en clair (sur papier, et même dans un fichier) ;
- Si un collègue vous demande des accès sensibles par écrit, n'hésitez pas à lui passer un petit coup de téléphone pour vous assurer que la demande vient bien de lui ;
- Ne partagez les mots de passe qu'avec les collègues qui en ont une nécessité ;
- Avoir un disque dur chiffré ;
- Faire un formatage long avant de jeter ou revendre du matériel informatique ;
- …

Le but n'est pas de vous infantiliser, c'est juste que d'après des études statistiques (le résultat varie très peu), la majorité des failles de sécurité viennent d'une erreur humaine.

Chiffrement SSL par défaut

Si vous avez choisi notre stratégie d'hébergement votre hébergeur vous permet en un clic d'activer la génération d'un certificat SSL afin de disposer d'une utilisation HTTPS sur votre nom de domaine. Cela permet au navigateur de vos utilisateurs de chiffrer leurs communications jusqu'à l'hébergeur, évitant ainsi qu'ils subissent des attaques de type man-in-the-middle.

Selon l'hébergeur vous pouvez aussi désactiver l'accès via du simple HTTP en demandant une redirection systématique vers le protocole HTTPS. Si l'hébergeur ne le propose pas, vous pouvez le faire dans votre frontend directement en vérifiant la variable `window.location.scheme`, et aussi sur votre backend en regardant les headers HTTP (pour savoir quelle a été l'adresse appelée).

Robustesse des mots de passe

Les pirates ont des bases de données de mots de passe déjà utilisés. Donc s'ils vous ont dans le collimateur, plutôt que de tester toutes les combinaisons imaginables pour accéder à votre compte, ils vont tester les mots de passe les plus évidents trouvés, puis ils peuvent aussi essayer des suites logiques de mots en analysant vos profils sur internet (nom d'un chien, nom d'un parent, ville… on appelle cela du reverse engineering).

Il ne faut donc pas se contenter d'un simple `123456` (d'ailleurs si un site public vous a déjà autorisé à rentrer cela, signalez-le leur ?), mais plutôt essayer de rendre la tâche difficile aux hackers comme :

- au moins 12 caractères ;
- au moins une majuscule et une minuscule ;
- au moins un chiffre ;
- au moins un caractère spécial (`@`, `!`, `?`…).

Plus le mot de passe est complexe et peu évident et plus il sera long et énergivore de le "brute forcer". Si vous voulez en savoir plus l'ANSSI fournit un petit calculateur pour estimer la robustesse de vos mots de passe.

L'ASTUCE DE LA PHRASE DE PASSE ("PASSPHRASE" EN ANGLAIS)

Il est parfois difficile de se souvenir de mots de passe long, une technique consiste à litteralement faire de votre mot de passe **une phrase de passe**

comme `Sillon a été débuté en 2023`.

Si le site ne vous permet pas de mettre certains caractères comme les espaces, adaptez le format : `SillonAEtéDébutéEn2023`

IL FAUT RELATIVISER

Qu'ils aient récupéré une base de données où vous êtes inscrit (avec votre mot de passe hashé), ou qu'ils tentent en direct sur un service en ligne… ils ne vont normalement pas passer des mois à chercher VOTRE mot de passe.

Gestionnaire de mots de passe

Il faut utiliser des mots de passe différents sur chaque service que vous utilisez. Pourquoi ? Car si des pirates trouvent vos identifiants pour un site, ils auraient juste à tenter la même combinaison `email + mot de passe` sur d'autres sites pour y accéder.

Pire encore si le service piraté est celui de votre boite email car ils seraient en mesure de faire des "réinitialisations de mot de passe" sur chaque site... Voir le chapitre sur le 2FA pour s'en prémunir.

Le problème avec cette stratégie c'est que plus vous avez d'outils et plus votre mémoire aura du mal à retenir vos mots de passe. S'en suit un biais naturel où pour définir vos mots de passe vous allez y intégrer des patterns communs. De manière schématique cela donnerait :

- Mot de passe du site "Pomme" : `pomme_c0llectiviteparis75`
- Mot de passe du site "Orange" : `orange_c0llectiviteparis75`

Ici c'est flagrant que le mot de passe sur le site "Poire" sera probablement `poire_c0llectiviteparis75`. Même si vous essayez de rendre cela un peu plus complexe, il existe des bases référençant les biais les plus utilisés lors de la création de mot de passe.

Pour palier à cela, nous préconisons l'utilisation d'un gestionnaire de mots de passe afin de tout le temps utiliser des mots de passe complètement aléatoires ayant minimum 32 caractères. Le gestionnaire aura le rôle de trousseau de clés virtuel :

vous pouvez y accéder via une application mobile, un site internet, ou via une extension de votre navigateur.

Les gestionnaires que nous recommandons :

- Dashlane (français) ;
- Bitwarden (open source) : bien que leur offre SaaS soit hébergée à l'étranger, Bitwarden a le mérite d'être open source. Chaque modification que vous apportez à votre coffre est chiffrée localement et reste chiffrée jusque dans leur base de données. Donc en SaaS leur équipe n'a pas la possibilité de lire vos mots de passe. *(Notez qu'héberger soi-même un serveur Bitwarden est déconseillé car c'est vraiment le genre de service que vous voulez opérationnel 24/7)*
- Passbolt (open source) : contrairement à Bitwarden son défaut est qu'il ne supporte pas le standard WebAuthn (détaillé plus bas), par contre son avantage est de permettre une granularité très précise dans les droits de partage (par mot de passe, et non par collection de mots de passe).

> **REMARQUE**
>
> Ces trois outils ont la gestion des équipes afin de partager des mots de passe liés au produit (API keys...). Bien entendu si vous créez des équipes essayez de faire des petits groupes. Si c'est lié à de la technique n'allez pas partager cela avec ceux du marketing, et vice-versa.

Le gestionnaire de mot de passe (coffre) est protégé par un seul et même "master password" au minimum. Il faut qu'il soit compliqué mais que vous puissiez vous en souvenir. Vous vous en doutez, **en l'état ce n'est pas suffisant,** il est risqué d'avoir une simple stratégie de "un pour les gouverner tous", il faut mettre le paquet pour sécuriser son gestionnaire (**voir absolument le chapitre suivant sur le 2FA**).

2FA / Security keys

Le problème d'une authentification avec un simple mot de passe, c'est que si quelqu'un vous voit l'écrire dans le train, ou vous le vole en étant à l'autre bout du monde avec un virus, c'est alors "open bar", vous n'avez aucun moyen de limiter la casse.

C'est là où **le deuxième facteur d'authentification (2FA) est primordial**, c'est normalement une sécurité additionnelle proche de vous physiquement pour valider des opérations (de connexion dans notre contexte).

Ainsi, si :

- On vous pirate à distance votre mot de passe, il manque au pirate le 2FA ;
- On vous vole votre 2FA, il manque au pirate le mot de passe.

Si jamais la personne arrive à avoir votre mot de passe et votre 2FA… Soit vous avez utilisé un 2FA basique, soit vous êtes victime d'un coup organisé.

Il existe plusieurs types de 2FA :

1. Validation par SMS (la moins sécurisée en cas de vol de téléphone puisque souvent aucun besoin de le déverrouiller pour lire les SMS) ;
2. Validation par l'application mobile du produit (les banques appellent souvent dans ce cas votre téléphone un "terminal de confiance") ;
3. Validation par TOTP (suite de chiffres valide 30 secondes) (souvent géré par une application nommée `??? Authenticator`) ;
4. Validation par une clé de sécurité (WebAuthn étant le dernier standard).

On va surtout s'intéresser aux (3) et (4) qui sont les standards les plus utilisés pour

l'authentification. Tout est basé sur une clé privée unique stockée dans l'application `TOTP Authenticator`, ou dans votre clé de sécurité. Ce qui permet de chiffrer une séquence d'autorisation durant une authentification.

Si vous utilisez un gestionnaire de mots de passe, le 2FA est primordial 🔑.

Utiliser la (3) pour protéger vos comptes **(en plus du gestionnaire)** est déjà très bien, mais assurez-vous juste :

- Que votre TOTP Authenticator soit sauvegardé automatiquement sur des serveurs et que vous ayez récupéré les "clés de backup". Histoire que si vous perdez votre téléphone vous puissiez réinstaller l'application et retrouver tous les codes (sinon vous serez bloqués sur tous vos comptes, et il est très rare que les supports répondent pour débloquer ce genre de situation) ;
- Que votre téléphone (si l'Authenticator est dessus) ainsi que l'Authenticator aient un code de verrouillage pour être débloqué (ou par empreinte digitale).

Sinon pour le (4), **nous recommandons à ce que vous ayez quelqu'un autour de vous qui puisse vous montrer un peu comment ça marche au quotidien. C'est un processus plus étoffé que la (3) et qui demande un minimum de rigueur, il vaut donc mieux être bien sensibilisé.** Une clé de sécurité ressemble à une clé USB, sauf que son contenu ne peut pas être lu (celui qui tente de la démonter grille obligatoirement le circuit et la clé privée de chiffrement reste… privée puisque détruite). Il est très risqué d'en avoir qu'une car si vous la perdez (qui n'a jamais perdu de clé ?), vous perdez d'office votre seul moyen de 2FA.

Il est recommandé d'utiliser 3 clés de sécurité minimum (elles ont donc chacune une clé privée différente en leur sein) que vous configurerez sur chacun de vos comptes les plus sensibles (gestionnaire de mots de passe et boîtes emails). Vous vous doutez bien, il ne faut pas garder vos 3 clés ensemble, donc dans l'idéal :

- Une sur vous en permanence quand des vérifications 2FA vous sont demandées (de préférence une clé qui fait USB+NFC afin de pouvoir être

utilisé sur téléphone) ;
- Une cachée chez vous (sous le plancher… comme il vous sied tant que vous puissiez la retrouver) ;
- Une cachée chez le fils de la cousine de la tante de votre père 😎.

Si jamais vous perdez l'une des clefs, vous vous connectez avec l'une des clés de secours sur tous vos comptes ayant été configuré avec les clés de sécurité, puis vous supprimez l'ancienne et pour ajouter une nouvelle que vous venez d'acheter en remplacement.

Vous l'avez peut-être décelé mais si vous souhaitiez utiliser la méthode "security key" sur chacun de vos comptes il faudrait à chaque inscription avoir accès à toutes vos clés (compliqué quand vous les avez cachées à 50 kilomètres de chez vous). C'est pour cela que nous proposions de surtout configurer le gestionnaire et vos boîtes emails, car ces comptes ne bougeront quasi pas dans le temps.

La technique ensuite pour les autres services "plus basiques" est d'utiliser le 2FA TOTP (méthode (3)). C'est votre gestionnaire de mots de passe qui va servir d'Authenticator pour générer les codes qui durent 30 secondes (normalement les gestionnaires ont cette fonctionnalité).

> **PAS TOUJOURS LE CHOIX**
>
> Vous verrez que de toute façon vous n'aurez des fois pas le choix que d'adopter le (3) en 2FA. Hormis les services spécialisés dans la sécurité, l'option 2FA en clé de sécurité est assez rare comparé au TOTP.

Si on résume dans le cas d'un redémarrage de mon ordinateur :

1. Je souhaite me connecter sur un site `ABC` qui est protégé avec un 2FA TOTP ;
2. Je lance mon gestionnaire, entre mon master password, et utilise ma clé de sécurité pour passer le 2FA du gestionnaire ;

3. Je copie/colle le mot de passe du site `ABC` ;
4. Le site me demande le 2FA TOTP que je vais chercher dans mon gestionnaire (pour simplifier, souvent le gestionnaire va automatiquement le mettre dans le presse-papier pour que vous n'ayez plus qu'à le coller après un auto-remplissage des identifiants).

> **QUOI ACHETER ?**
>
> La marque la plus connue et très appréciée historiquement est Yubico (suédois) avec ses Yubikeys, mais dorénavant il existe plein d'alternatives (que nous n'avons pas testés).

Activer le 2FA est bénéfique tout autant dans le cadre privé que dans le cadre professionnel. Nous recommandons que votre équipe utilise le 2FA dans tous vos outils. Et quand c'est possible, de forcer dans l'outil le 2FA à tous les membres.

Notez qu'il est difficile de résumer tous les cas d'utilisation de ces méthodes par écrit, nous essayons ici de poser des bases afin de vous aider à vous lancer.

Accès restreints et alertes

Les accès à donner pour chaque service, outil ou fichier doivent être préalablement réfléchis. Pour aller droit au but : dites-vous qu'au niveau d'un produit, l'environnement de production ne doit être idéalement accessible que par 1 personne (voire 2 pour un peu de redondance).

Si votre hébergeur le permet, activez les alertes :

- Par email pour toute modification sur les configurations de l'environnement de production ;
- Par un système de chat (Mattermost, Slack…) pour des modifications sensibles sur les configurations de l'environnement de développement.

Vous pouvez aussi utiliser un outil comme GitGuardian pour vous prémunir de secrets qui seraient committés sans faire exprès sur votre repository (promis, ça arrive 😬).

RSSI

Peu importe à quelle entité vous êtes rattaché, vous avez sûrement dans votre hiérarchie un RSSI (responsable de la sécurité des systèmes d'information). Il est judicieux d'aller voir ce qui a été mis en place par cette personne et son équipe, afin d'être aligné avec leurs bonnes pratiques.

Par exemple, ils demandent peut-être à chaque produit de faire un DAT (document d'architecture technique), ou plus sommairement une matrice de flux pour limiter les risques (elle aide à comprendre d'où entrent les données et où elles partent).

Surveiller les CVE

Les CVE (~ failles de sécurité) sont disponibles via plusieurs flux RSS comme ceux de l'ANSSI (exemple de celui qui rassemble tout). Comme beaucoup d'informations passent dedans il peut être difficile d'être concentré dessus en tant que simple technicien d'un produit.

Vous pouvez transmettre votre stack au RSSI et lui demander s'ils ont une politique de surveillance (et s'ils vous remonteront les alertes qui vous concernent). Sinon, des forges comme GitHub peuvent vous notifier des CVE en fonction de l'arbre de dépendances de votre projet.

> **AVOIR UNE RÉACTION PROPORTIONNÉE**
>
> Gardez en tête qu'une CVE critique sur l'une de vos dépendances ne veut pas forcément dire que c'est critique pour vous. Cela peut dépendre du contexte d'utilisation de cette dépendance, peut-être que vous n'utilisez pas du tout la fonctionnalité concernée.

En cas de fuite de données

Si vous constatez une fuite de données, le plus important est :

1. Agir en limitant cette fuite et en changeant au plus vite les accès potentiellement corrompus ;
2. Communiquer aux personnes les plus à même de vous aider sur le court terme :
   i. À votre RSSI pour qu'il puisse vous accompagner ;
   ii. À la CNIL s'il y a un risque pour la vie privée des personnes concernées (et optionnellement aux personnes concernées par la fuite) ;
3. Comprendre en détails comment cela a pu se produire et ce qui a été touché ;
4. Documenter cette mauvaise expérience.

Quelques documents de l'ANSSI pour mieux se préparer et gérer une crise :

- Anticiper et gérer sa communication *(2021)* ;
- Les clés d'une gestion opérationnelle et stratégique *(2021)* ;
- Agilité & sécurité numériques *(2018)*.

> **ATTENTION**
>
> Si vous avez laissé en accès public des données privées (brièvement ou non), il faut réagir comme si c'était une fuite de données. Présupposer que personne n'ait accédé aux données exposées est une erreur.

Audits de sécurité

L'ANSSI a lancé le service MonServiceSécurisé afin de permettre une autoévaluation sur les principes de sécurité mis en place dans votre produit. C'est sous forme de QCM donc même sans connaissance poussée en cybersécurité vous allez pouvoir faire un tour d'horizon, cela vous donnera des idées sur ce qu'il reste de judicieux à faire. *À noter que les questions sont génériques, certaines peuvent ne pas être adaptées à votre produit*.

Une fois que vous vous sentez prêt, demandez à votre RSSI de jeter un œil à votre QCM afin qu'il puisse apporter son expertise. Si c'est concluant vous pourrez ensemble passer à l'étape suivante sur MonServiceSécurisé afin de demander une homologation. Cela permet d'avoir une trace datée de tout ce qui a été diagnostiqué, et ça permet d'avoir une note sur 5 pour situer où l'on est exactement. *La note reste relative à votre contexte puisque toutes les questions n'ont peut-être pas été répondues : vous pourriez avoir 3/5 que ça serait très bien.*

Vous l'aurez compris, MonServiceSécurisé est juste déclaratif, personne ne va venir inspecter votre code source ou tenter de faire du pentesting (tests d'intrusion). **Mais dans le cas où vous hébergez (rien qu'un petit peu) : des données sensibles comme des données de santé, financières, judiciaires… Il faut passer un vrai audit de sécurité.** L'ANSSI en elle-même ne pratique pas les audits mais a des partenaires privés certifiés pour cela (vous pouvez demander des conseils au support MonServiceSécurisé si vous ne savez pas à qui vous adresser).

Cela va dépendre de votre produit et du prestataire mais comptez dans les 10 000 euros pour un audit. Ils vont ensuite vous fournir un rapport avec toutes les failles détectées. À vous de corriger celles qui sont pertinentes (le contre-audit n'est pas une obligation).

DO IT YOURSELF

Les prestataires pour les audits font en partie tourner des logiciels automatisés pour tenter des intrusions. **Si un vrai audit n'est pas requis par votre projet**, vous pouvez toujours faire tourner ce type de logiciels vous-même pour détecter de potentielles brèches. Si cela vous intéresse, jetez un œil du côté de SAST, DAST et IAST.

Configurer les headers HTTP

Les headers HTTP venant de votre serveur vont dicter aux navigateurs internet comment ils peuvent agir (ou non). Certains doivent être réglés pour limiter les risques liés à la sécurité. Le plus simple est d'utiliser un outil comme Helmet.js pour définir par défaut la meilleure configuration à appliquer.

Comme avec Next.js vous n'interagissez pas directement avec le serveur Express afin de garder l'optimisation native du serveur web, et qu'Helmet.js ne propose pas de plugin Next.js, notre solution est pour l'instant de configurer dans votre `next.config.js` la propriété `headers` afin de renseigner toutes les valeurs par défaut de Helmet.js.

> **INFO**
>
> Pour chaque nouvelle ressource utilisée et hébergée sur un autre domaine vous devrez ajuster les propriétés du header `Content-Security-Policy` (type `default-src`, `img-src` `script-src`…). La console de votre navigateur vous préviendra que le header du serveur n'est pas assez permissif.

Concernant les headers entrants vous ne devriez pas être à risque puisqu'en suivant nos recommandations d'hébergement vous ne gérez pas vous-même la partie routage (qui se fait habituellement avec du Nginx, HAProxy, Istio…). Portez toutefois une attention particulière si vous vous basez sur l'IP du client pour authentifier une action (cela est parfois vrai pour les webhooks qui n'utilisent pas encore la signature du payload). Le formatage du header transmettant la chaîne d'IPs n'est pas la même en fonction des contextes (à cause des outils). Nous

conseillons la lecture de cet article (l'auteur a réalisé une librairie en Golang mais nous ne sommes pas au fait d'un équivalent en JavaScript).

Sécuriser les URLs des fichiers hébergés

Que nous utilisions un hébergement de fichiers en base ou via des buckets S3, il faut garantir que les URLs de nos fichiers sensibles soient éphémères. Même si vous avez un identifiant de fichier aléatoire (UUID) et que vous estimez qu'on ne peut pas deviner l'URL :

- L'URL peut être gardée dans l'historique de navigation ;
- L'utilisateur légitime peut l'avoir stockée dans un autre endroit visible par d'autres ;
- L'utilisateur peut ne plus être légitime pour voir ce fichier.

En ajoutant une durée de validité à l'URL du fichier vous garantissez que le fichier ne sera vu que dans le contexte où l'utilisateur aura communiqué avec l'API. *Bien entendu, pour des fichiers publics qui peuvent être indexés (logos…), il n'y a aucune raison d'y ajouter une sécurité.*

L'idée est de donner une validité à notre URL en y ajoutant en paramètre une signature (générée par chiffrement). Imaginons que nous sommes dans une SPA (les appels API se font en asynchrone du chargement de page) :

1. L'utilisateur va sur `monsite.com/compte` ;
2. Le frontend demande à l'API de récupérer l'objet utilisateur ;
3. L'API :
   i. Récupère l'utilisateur en base de données et voit que la propriété `private_photo` est une pièce jointe qui a l'identifiant 123 ;
   ii. Passe l'identifiant dans un petit helper qui va d'abord générer l'url finale `monsite.com/fichier?id=123` ;
   iii. Puis générer un JWT avec comme payload `urn:claim:file_id = 123` et une validité de 10 minutes pour ensuite l'ajouter en paramètre pour donner `monsite.com/fichier?id=123?token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1cm46Y2xhaW06ZmlsZV9pZCI6IjEyMyIsImlhdCI6MTUxNjIzOTAyMn0.xpP88WITmYZ-F0l1iL_VbAcc-PBU53xeE4Dnm-M9Pj4` ;
   iv. Renvoie l'objet utilisateur avec pour `private_photo` l'URL finale.

C'est très schématisé mais c'est l'idée. Ainsi le endpoint `/fichier` a juste à vérifier la validité du JWT avec n'importe quelle librairie JWT classique.

- Si vous utilisez les buckets S3 : certains providers de buckets S3 ont cette fonctionnalité de signature d'URL, il faut donc que votre backend se charge de signer l'URL avec la librairie cliente du provider ;
- Sinon si vous êtes en base de données il faut créer vous-même ce petit helper.

Le bénéfice d'avoir la génération de l'URL signée quand on récupère la donnée métier est que vous passez forcément par les étapes de vérification des droits d'accès. Vous évitez ainsi de centraliser les droits d'accès au niveau de la table de fichiers (qui ont des risques d'être désynchronisés avec les vrais entités métiers).

Quelques points importants (que ce soit en base ou via S3 directement) :

- La validité du JWT n'a pas besoin d'être très longue puisque aussitôt le frontend récupère les données, aussitôt il va charger le fichier (le fichier peut être gardé en cache) ;
- En l'état actuel des choses, la mise en cache ne va pas marcher car le JWT est différent à chaque affichage de page. La technique est de rendre fixe l'URL sur un intervalle donné (ce n'est pas parfait, mais ça fait le travail) :
  - Au moment de générer le JWT on spécifie la date d'expiration en utilisant un modulo (de 10 minutes par exemple) ;
  - Les expirations se font donc à des "heures fixes" (1h10, 1h20, 1h30…) ;
- Assurez-vous quand même sur votre endpoint de renvoyer le header HTTP `Cache-Control` afin d'adapter le contexte et de ne pas garder localement des images dont la validité a expiré :
  - Pour un fichier public avec la valeur : `public, immutable, no-transform, max-age=0`
  - Pour un fichier privé avec la valeur : `private, max-age=${minutesToSeconds(15)}`.

Désolés si l'ensemble paraît être un peu de travail, mais c'est le nécessaire pour sécuriser proprement les fichiers (l'utilisation d'un S3 prend en charge quelques étapes mais laisse reposer sur vous l'architecture de sécurisation ainsi que sa mise en place).

Se prémunir des fichiers infectés

C'est malheureux mais il faut toujours se méfier des données qui entrent dans votre applicatif, encore plus quand il s'agit de fichiers. Quelques exemples :

- On peut utiliser une extension d'image sur un fichier exécutable pour qu'il passe des vérifications trop sommaires ;
- Même si vous utilisez un hébergeur de fichiers S3, celui-ci ne s'occupe en général pas de vous prémunir des virus ;
- On peut injecter du code malicieux au sein de fichiers comme des PDF, alors qu'à première vue un fichier PDF semble inoffensif. Cela va dépendre du logiciel qui ouvre ce fichier ;
- Un fichier infecté qui se retrouve dans votre applicatif sera peut-être récupéré par un agent public, et peut contaminer son ordinateur voire tout le réseau de son travail. Et dans ce cas… on ne peut pas le blâmer, **c'est l'applicatif qui est censé apporter de la sécurité à ses utilisateurs** !

Voici quelques pistes pour essayer de contrer ce type d'attaque :

1. Comme mentionné dans le chapitre sur le stockage de fichiers, l'upload de fichiers ne se fait pas en 1 temps. Il vaut mieux faire d'abord l'upload sur un endpoint dédié pour récupérer l'identifiant du fichier, et dans un second temps transmettre cet identifiant dans le formulaire métier. Le "temps 1" est une bonne occasion pour effectuer les vérifications listées plus bas. Si jamais vos fichiers sont trop gros et que vous ne voulez pas bloquer de manière synchrone l'utilisateur le temps de l'analyse, vous pouvez les effectuer en asynchrone, en choisissant que tant qu'elles ne sont pas terminées l'API

refusera de servir l'URL du fichier. À vous de déterminer ce qui est le plus approprié ;

2. Ne pas faire confiance aux metadatas que le frontend vous joint en même temps que le fichier. Une fois le fichier dans votre zone tampon vous pouvez vérifier son vrai type en analysant le marqueur au sein du fichier (cela peut se faire avec `magic-bytes`). L'idée est que certains types de fichier (hors simple fichier texte par exemple) ont une suite de premiers bytes connue, donc sans analyser la totalité du fichier on peut savoir déjà s'il correspond au type que l'on attendait. **C'est une première vérification "rapide"** ;
3. Et comme n'importe quel fichier peut potentiellement contenir du code malicieux, il est utile d'utiliser un antivirus pour scanner l'intégralité du fichier. **C'est une vérification "lente" mais complète**. Nous préconisons d'utiliser l'outil ClamAV via la librairie JavaScript clamscan. Comme c'est un exécutable, il faut soit :
    i. L'avoir en local dans le même container que votre applicatif :
        a. Pour des buildpacks : https://github.com/Scalingo/clamav-buildpack ;
        b. Pour des environnements où vous avez la main (via Dockerfile par exemple), l'installer avec un gestionnaire de packages ;
    ii. L'avoir accessible sur le réseau et interagir avec par TCP (par exemple en déployant l'image Docker officielle). Ce cas est à privilégier si :
        a. Vous n'avez pas la possibilité de l'embarquer avec votre applicatif ;
        b. Vous êtes limités en ressources pour votre applicatif, et ne voulez pas le pénaliser ;
        c. Vous voulez mutualiser l'instance pour plusieurs applicatifs.

Il est important de garder en tête qu'**un antivirus aura toujours un temps de retard sur les failles**. Vos fichiers hébergés peuvent être détectés "inoffensifs" à l'instant T, mais corrompus en les scannant de nouveau 3 semaines plus tard (car vous aurez mis à jour la base de connaissance de votre antivirus). La récurrence des scans dépend de votre contexte :

- Si vous n'hébergez que des images (type `.jpg`, `.png`…), il est probable que

seul le scan initial soit suffisant puisque les utilisateurs sont censés ouvrir ce fichier dans un logiciel approprié (qui ne va pas exécuter le fichier). Vous pourriez vous passer de la complexité et du coût de tout rescanner ;

- Si vous hébergez des documents plus complexes (tableurs, fichiers exécutables…), il faut malheureusement être très vigilant, et scanner périodiquement fait sens (avec le prérequis d'avoir mis à jour votre antivirus entre chaque occurrence).

Faire la vigie en étant outillé

Pour maintenir un service de qualité dans la bonne direction il faut avoir des capteurs qui vous signalent quand quelque chose se passe mal. Mais rappelez-vous de ne pas fournir d'informations sensibles à ces outils.

Pour les besoins techniques

Techniquement nous recommandons d'utiliser Sentry (mentionné dans le chapitre sur les outils tiers) pour centraliser vos erreurs frontend et vos erreurs backend. Vous…

Pour les besoins métiers

Support utilisateur

Pour les besoins techniques

Techniquement nous recommandons d'utiliser Sentry (mentionné dans le chapitre sur les outils tiers) pour centraliser vos erreurs frontend et vos erreurs backend. Vous pouvez par défaut activer le "catch all" afin que le client Sentry remonte toute erreur qui n'est pas "catchée". Cela vous donnera la "stack trace" pour investiguer plus en détails. Mais vous pouvez aussi explicitement formater les erreurs à remonter afin de mettre des règles d'alertes depuis l'interface.

Par exemple durant la consommation d'une "job queue", au lieu de laisser partir des erreurs lambda vous pourriez formater des `JobQueueError` et dans l'interface dire que si vous avez plus de 3 `JobQueueError` sous 1 minute, alors vous envoyez un email à l'équipe pour investigation.

Pour les besoins métiers

Support utilisateur

Si un utilisateur a une idée ou un problème, il faut que l'utilisateur puisse le partager facilement avec le support. Dans une phase d'investigation nous préférons éviter le fait d'aller chercher l'email du support, écrire la demande, et quelques temps après avoir une réponse… et continuer dans l'asynchronisme car il est possible que l'utilisateur ne se donne même pas la motivation de faire la première étape.

Nous recommandons d'utiliser une messagerie "livechat" comme Crisp qui apporte un peu de dynamisme car elle est à portée de clic. Derrière, plusieurs membres de l'équipe peuvent être reliés au compte et choisir quelles discussions ils s'assignent. Sachant que l'application est disponible sur toutes les plateformes (dont mobile), c'est vraiment d'une praticité et expérience utilisateur supérieures.

Livechat sur ce site

> **ASTUCE**
>
> Pour des questions RGPD et afin d'éviter de mettre une demande de consentement de cookies (car l'action utilisateur sera volontaire), nous recommandons d'avoir un bouton "Support / Contactez-nous" qui, **seulement au moment du clic**, va charger le script Crisp et ouvrir le livechat.

L'idée lors de la phase d'investigation est d'avoir le bouton "Support" bien visible pour inciter à faire des retours. Une fois que votre produit est plus stable avec beaucoup d'utilisateurs, vous pouvez le dissimuler un peu plus voire même revenir

à un système d'emailing (si vous n'avez pas la bande passante pour prétendre gérer un livechat).

Mesure d'impact

Bien que l'on puisse, lors des phases initiales d'investigation, quantifier l'appétence de nos beta-testeurs. Une fois le produit lancé, il faut être en mesure de poursuivre cette analyse pour continuer à remettre en question l'utilité du produit et ajuster la roadmap.

On pourrait très bien tout surveiller, mais il faut surtout adapter la mesure aux problématiques métiers qui font que l'on répond au besoin. Exemples à quantifier :

- Pour un produit pour des demandes de médiation locales :
  i. Nombre de dossiers ouverts / résolus / en échec ;
  ii. Localisation des demandeurs (pour peut-être avoir plus d'effectif dans les collectivités) ;
- Pour un produit qui aide à la sécurisation d'un service numérique :
  i. Combien de projets ont répondu aux questions ;
  ii. Combien de projets sont allés jusqu'à l'homologation ;
  iii. Quelle est la note moyenne estimée de tous les projets.

Pour mieux comprendre et contextualiser ces statistiques métiers vous pouvez les corréler par exemple à : nombre de visites (uniques ou non), quelles pages sont visitées, pendant combien de temps, depuis telle page est-ce que l'utilisateur arrête sa visite… L'outil le plus connu et le plus étoffé est sans aucun doute Google Analytics, mais **il pose un certain nombre de problèmes concernant la vie privée de vos utilisateurs**. Comme alternative nous préconisons l'utilisation de Matomo qui répond aux principales fonctionnalités mentionnées.

À vous ensuite d'analyser ces données sur un intervalle de temps, comparer avec les périodes précédentes, construire des graphiques… Plusieurs possibilités :

1. La plus complexe et complète : utiliser un outil de business intelligence (BI) comme Metabase pour relier vos différents outils (base de données, Matomo…) ;
2. La plus simple et basique : périodiquement exécuter une requête SQL sur votre base de données pour en extraire les données intéressantes (sous format `.csv` par exemple) et les importer dans un tableur.

Dans les deux cas, vous pouvez utiliser une ou plusieurs `VIEW` dans votre base de données afin de préformater les données voulues pour qu'elles soient facilement accessibles (notamment par un outil de BI). Mais aussi pour restreindre les droits en lecture à ces seules vues afin d'éviter que quelqu'un puisse se balader jusqu'aux mots de passe…

> **RECOMMANDATIONS**
>
> - **Faites toujours vos statistiques sur des données anonymisées.** Cela limitera le risque que des données sensibles se propagent dans les outils. Il est d'ailleurs peu probable que comparer des noms ou des numéros de téléphone aient un sens métier pour vous ;
> - Datez les actions métiers sur vos entrées en base. Avoir un `createdAt`, `closedAt` ou autre vous laissera utiliser des comparaisons temporelles. Si cela n'a pas été fait ou n'est pas approprié, vous pouvez toujours définir explicitement un intervalle de temps à laquelle faire snapshot des vues statistiques, pour les stocker dans une table dédiée. On aurait par exemple en *ligne 1* `janvier 2023` avec toutes les statistiques compilées, en *ligne 2* `février 2023`… c'est juste que vous ne pourrez pas modifier cette intervalle sur les données du passé ;
> - N'hésitez pas à partager ces données avec toute votre équipe. Si les statistiques sont positives, c'est toujours gratifiant de voir que l'outil

que l'on développe est utilisé. Et si elles ne le sont pas, discutez-en.

L'accessibilité est obligatoire

L'accessibilité numérique est le fait de permettre aux personnes en situation de handicap de pleinement utiliser un service numérique. Et depuis 2012 **elle est obligatoire sur tous les sites publics**. Malheureusement que ce soit les chefs de projet, développeurs, designers… le constat est que peu s'en soucient dans les prémisses d'un projet alors que rendre son service accessible :

- Peut influer sur votre UI, et donc sur vos maquettes ;
- Peut influer sur votre code technique (on donne du contexte au code HTML avec ARIA) ;
- Peut influer sur votre panel de beta-testeurs.

Comme c'est une obligation légale, un projet public peut se retrouver 2 ans après à devoir passer des semaines à adapter son site alors qu'y réfléchir dès le début aurait eu un impact moindre sur la vélocité (sans parler que pendant 2 ans le projet a écarté une obligation légale…).

Rien qu'en s'y intéressant quelques heures, cela permet d'avoir quelques automatismes qui réduit la fraction de l'accessibilité. Le DSFR a été pensé avec ces problématiques, et permet d'orienter le designer et le développeur afin de ne pas faire les erreurs les plus basiques (contraste entre les couleurs, espacement…). Aussi, si vous devez choisir un framework UI tiers (ou une simple librairie pour un composant), jetez d'abord un œil s'ils ont une approche accessible, car adapter des composants "packagés" non prévus pour l'accessibilité ("a11y") peut se révéler impossible.

Nous considérons 3 statuts dans l'accessibilité d'un site public :

- **Non conforme** : le site n'a pas été audité ;
- **Partiellement conforme** : le site a été audité mais le score n'est pas assez élevé ;
- **Conforme** : le site a été audité et a obtenu un score exemplaire ;

Pour savoir si votre site est accessible, il faut d'abord avoir conscience que seulement ~25% de l'accessibilité peut être testée automatiquement. Cela donne de vraies bases puisque sans elles il est impossible que les ~75% restant soient valides. Pour se faire vous pouvez utiliser :

- Lighthouse (disponible en extension dans votre navigateur) ;
- `axe-core` si vous utilisez un "bundler" pour votre applicatif (type Webpack..) ;
- Storybook avec l'extension `storybook-addon-a11y` (mettre Storybook au centre de votre développement est une pratique qui peut se révéler efficace).

Ensuite vous devez passer par un audit d'accessibilité où des vérifications "manuelles" et "en situation" vont être faites sur votre produit. Cet audit complet peut être fait par un organisme tiers et son coût est d'environ 4500 euros.

> **CONSEIL**
>
> Il est préférable de faire cet audit quand votre produit est assez avancé techniquement (mais pas non plus des années après son ouverture 🙂).

> **REMARQUE**
>
> Il est aussi possible que dans votre entité vous ayez une équipe accessibilité qui s'occupe de faire des "mini-audits". Vous resterez légalement "non conforme" sans l'audit complet, mais cela vous apportera un premier retour d'experts.

Parmi les obligations légales en accessibilité, il faut afficher le statut sur la page d'accueil (prévu par le DSFR pour être dans le footer), et avoir publié votre déclaration d'accessibilité (vous pouvez vous aider d'un générateur).

L'écriture inclusive

📄 FALC. Tout simplement.

Le facile à lire et à comprendre (FALC) est une méthode d'écriture claire et limpide. Bien que le ministère de la Culture propose d'envoyer des textes à "traduire", il faut …

📄 Règles de féminisation

Ce point est encore à approfondir de notre côté, n'ayant pas trouvé de texte officiel clé indiquant les bonnes pratiques "dans tous les domaines". Nous avons seulemen…

FALC. Tout simplement.

Le facile à lire et à comprendre (FALC) est une méthode d'écriture claire et limpide. Bien que le ministère de la Culture propose d'envoyer des textes à "traduire", il faut tout d'abord rappeler que nous devons nous-mêmes nous mettre dans la situation de l'utilisateur au moment de la rédaction, sans a priori. Cela rentre clairement dans le travail d'expérience utilisateur (en plus de faire des parcours UX fluides).

Quelques exemples :

- Ne pas justifier le texte (concernent principalement les personnes dyslexiques) ;
- Évitez de mettre des mots trop techniques quand cela n'est pas nécessaire (et si besoin, pointer vers des liens).

Si vous voulez en savoir plus, n'hésitez pas à lire les règles européennes pour une information facile à lire et à comprendre ou encore à consulter les diapositives du Pôle Numérique Inclusif de beta.gouv (qui nous a aussi conseillé l'outil Hemingway pour identifier les structures de phrase compliquées).

La rédaction de Sillon joint au maximum les sources ou la documentation à chaque concept, mais n'est pas encore passé au crible du "FALC" pour tous les autres aspects de l'écriture. Désolés 🙏.

Une rédaction limpide participe aussi aux critères d'accessibilité vus dans le chapitre précédent.

Règles de féminisation

> **REMARQUE**
>
> Ce point est encore à approfondir de notre côté, n'ayant pas trouvé de texte officiel clé indiquant les bonnes pratiques "dans tous les domaines". Nous avons seulement trouvé une circulaire pour le ministère de l'Éducation qui nous semble appropriée.

L'égalité entre les usages des genres est importante pour respecter l'inclusion. Voici quelques pistes :

1. Des mots épicènes (mots qui désigne indifféremment le genre) ;
2. Formules englobantes : `les responsables des achats` plutôt que `les acheteurs et les acheteuses` ;
3. Formulation décomposée : `le candidat ou la candidate` ;
4. Point médian : `certain·e·s`.

Petite attention sur le fait que la (4) s'éloigne du "facile à lire et à comprendre". Et que la (3) allonge relativement la longueur de phrase.

Là aussi tout n'a pas encore été passé au crible parmi la centaine de pages de Sillon 🙏.

> **ASTUCE**
>
> Si jamais vous souhaitez pousser l'engagement de votre site plus loin en proposant à l'utilisateur de choisir explicitement l'identité de genre à utiliser pour interagir, nous vous conseillons d'utiliser un module `i18n` afin

de centraliser l'écriture qui devient "dynamique" au même titre qu'avec du pluriel.

Le légal

La licence de votre produit

Que le code technique de votre produit soit en libre accès ou caché dans un coffre, il est important de choisir une licence adéquate pour que l'utilisation du code de v...

Vos dépendances ont aussi des licences

Que l'on parle d'outils tiers ou de librairies techniques, il est important de connaître leurs restrictions liées à l'usage afin de ne pas mettre votre produit dans une situati...

La protection des données

Le règlement général sur le protection des données (RGPD) encadre le traitement des données personnelles. En suivant nos chapitres vous vous confortez en partie à la...

Cycle de vie de la donnée

Nous mentionnions la possibilité de supprimer les données d'un utilisateur, sachez que dans tous les cas la CNIL recommande de garder les données seulement 2 ans a...

Documents requis

Certains documents peuvent être requis et être facilement accessibles depuis votre produit numérique :

Sécuriser la marque de son produit

Le nom de domaine

La licence de votre produit

Que le code technique de votre produit soit en libre accès ou caché dans un coffre, il est important de choisir une licence adéquate pour que l'utilisation du code de votre produit reste dans un périmètre que vous acceptez. Nous vous conseillons de lire ce document qui introduit l'usage des licences, et d'utiliser le tableau de correspondances de data.gouv.fr pour choisir la licence la plus adaptée.

Nous aimerions juste apporter une clarification sur la licence `MIT` qui est très permissive :

- Cela fait sens pour une simple librairie technique ;
- Mais dans le cadre d'un produit complet, cela autorise une entreprise privée à tout reprendre, mettre son propre logo et le revendre à une entité publique qui n'aurait pas connaissance de l'outil public initial. *Vous pouvez limiter ce genre d'abus en adoptant la licence* `AGPL-3.0`.

À noter que le contenu hors technique devrait être mis sous la licence de l'État `etalab-2.0`.

Vos dépendances ont aussi des licences

Que l'on parle d'outils tiers ou de librairies techniques, il est important de connaître leurs restrictions liées à l'usage afin de ne pas mettre votre produit dans une situation irrégulière.

Il faut par exemple, **à chaque fois que vous choisissez une librairie, vérifier sa licence** qui se situe normalement dans le repository du code source (ou adossée à la page de présentation de la forge).

La protection des données

Le règlement général sur le protection des données (RGPD) encadre le traitement des données personnelles. En suivant nos chapitres vous vous confortez en partie à la plupart des critères RGPD quant à la localisation des données… Pourtant cela va beaucoup plus loin, par exemple :

- Si un utilisateur vous demande quelles informations vous possédez de lui vous devez être en mesure de lui fournir un "export" complet ;
- Si un utilisateur vous demande un "export" pour transférer les données vers un autre service (portabilité), vous n'y êtes pas contraint car vous menez une mission d'intérêt public ;
- Si un utilisateur demande la suppression de son compte et de ses données, là encore du fait de votre mission d'intérêt public vous n'y êtes pas obligé. Si vous décidez de répondre à sa demande, quelques nuances sur la notion "d'effacement" :
  - Vous pouvez de façon ciblée anonymiser les éléments d'un utilisateur quand la suppression impliquerait le dysfonctionnement du produit (par exemple sur un historique d'échanges) ;
  - Certains domaines (judiciaire, bancaire…) requièrent une traçabilité sur plusieurs années, vous ne pouvez pas perdre l'identité d'un utilisateur.

Nous vous conseillons dans tous les cas de vous rapprocher d'une personne compétente dans ce domaine pour mieux cerner ce que vous pouvez faire ou ne pas faire. Car bien que le RGPD existe depuis quelques années, il y a parfois des incohérences où même les experts ont des doutes :

- l'adresse IP permet d'identifier quelqu'un, je la supprime ou pas ? Si je dois la supprimer de mes logs, déjà c'est compliqué, et en plus je perds la traçabilité

en cas d'action malveillante… que faire 🤯…
- Vous avez des backups, donc vous stockez encore les données utilisateurs de la personne. *(et si vous restaurez le backup en production, il faut aussi ne pas oublier de renettoyer la donnée…)*

Cet interlocuteur peut être le délégué à la protection de données de votre entité (DPO), la direction des affaires juridiques (DAJ), ou encore la CNIL.

En tout cas vous l'aurez sûrement compris, les décisions techniques que vous prenez peuvent vous rendre la vie infernale :

- Si vous utilisez un pattern type "event sourcing" et que vous devez garder la pile d'événements pour recréer l'état du système… Dans la logique vous aurez un `createUser(123, "Jean", …)` puis plus tard vous aurez un `deleteUser(123)`. Le problème c'est qu'à jamais vous aurez stocké les informations personnelles de Jean (*il est possible de faire des "snapshots" d'une pile d'événements pour supprimer une partie de l'historique, mais ça complexifie encore plus le pattern)* ;
- Si vous envoyez un email avec des données sensibles via un provider d'emailing, il faut vous assurer que celui-ci ne garde pas des copies du contenu des emails. Et si c'est le cas, il faut qu'il ait une vraie raison de le faire et que vous puissiez les supprimer. *(Sur ce point et si vous avez un applicatif avec un espace "mon compte", nous recommandons de se limiter aux prénom et nom dans le contenu de l'email et de mettre des liens vers les pages contenant la donnée sensible)*

Veillez à bien vous renseigner sur les outils tiers anodins que vous embarquez dans votre produit. Rien que d'utiliser des polices *(fonts)* directement depuis des serveurs tiers donnent certaines informations quant à vos utilisateurs. C'est la même chose quand vous utilisez des outils d'analytics en SaaS.

> **REMARQUE**

> Nous essaierons de clarifier les points les plus complexes quand nous aurons des certitudes légales sur ceux-ci...

Cycle de vie de la donnée

Nous mentionnions la possibilité de supprimer les données d'un utilisateur, sachez que dans tous les cas la CNIL recommande de garder les données seulement 2 ans après la dernière utilisation (sauf exceptions qu'ils ont mentionnées).

Cas concret :

1. Un citoyen fait une demande de médiation ;
2. Le litige est réglé le 31/01/2019 ;
3. Comme le service n'a pas besoin du dossier et des pièces jointes (ils ont déjà fait un export statistique pour l'année 2019), le service doit supprimer tous les éléments de cette demande dès le 01/02/2021.

En cas de doute, rapprochez-vous de votre délégué à la protection des données (DPO).

> **EXEMPLE D'EXCEPTION**
>
> Un établissement d'enseignement supérieur devra par exemple respecter la durée d'utilité administrative (DUA) de 50 ans pour conserver les dossiers étudiants.

Documents requis

Certains documents peuvent être requis et être facilement accessibles depuis votre produit numérique :

1. Mentions légales ("legal notice") ;
2. Politique de confidentialité ("privacy policy")
3. Conditions générales d'utilisation ("terms of use") ;
4. Analyse d'impact relative à la protection des données (AIPD) **(TODO: à clarifier)**.

Quelques cas concrets :

- Site vitrine uniquement informatif, sans cookie, sans compte utilisateur, sans fonctionnalité particulière : *Mentions légales* ;
- Site vitrine mais qui traite des données ou dépose des cookies : *Mentions légales + Politique de confidentialité* ;
- Site proposant un service avec compte utilisateur, dépose de cookies, et fonctionnalités : *Mentions légales, Politique de confidentialité, Conditions générales d'utilisation*.

Ce sont des documents publics légaux, il faut donc que **leur rédaction soit supervisée par une équipe légale** au sein de votre entité afin d'éviter tout risque de sanction pénale.

INFORMATIONS SUR LES COOKIES

- Les cookies déposés doivent être listés dans la "Politique de confidentialité" ;

- Ceux liés à l'utilisation explicite du produit n'ont pas besoin d'un consentement, de même si c'est les cookies d'un service tiers et que l'utilisateur doit dûment cliquer sur une validation pour interagir avec ce service.
- Le DSFR propose une "modal" de consentement pour les lister et ainsi demander l'autorisation de l'utilisateur **avant de charger les scripts de services.**

Sécuriser la marque de son produit

Le nom de domaine

Après concertation et quand vous pensez avoir le nom "définitif" de votre produit **il est judicieux de réserver au plus tôt les noms de domaine** avec les extensions `.fr` et `.com`.

L'idée en faisant cela **avant de communiquer sur votre futur produit** est d'éviter le cybersquattage et la spéculation sur les actifs numériques. Il faut savoir que beaucoup de noms de domaines sont achetés dans le but exclusif d'être revendu à prix d'or...

Si vous vous retrouvez dans la situation où le nom de domaine voulu est pris et que vous le voulez absolument :

1. Solution la plus rapide : vous pouvez tenter de faire une offre d'achat, mais passez par un intermédiaire pour que le vendeur n'ait pas notion que c'est pour un projet public. Car malheureusement certaines personnes n'auraient aucun scrupule à gonfler le prix pensant que vous avez un budget illimité...
2. Si votre produit ou "marque" a un passif vous êtes **peut-être** légitime à récupérer la propriété sur ce nom de domaine :
    i. S'il y a selon vous spoliation du patrimoine immatériel de l'État vous pouvez vous rapprocher de l'APIE pour qu'ils vous accompagnent *(condition : il ne faut pas être dans une entité publique juridiquement autonome pour être éligible à leur aide)* ;

ii. Vous pouvez tenter de récupérer un domaine avec une extension française (type `.fr`) en démarrant une procédure SYRELI auprès de l'AFNIC, le coût est de 250€ HT. Il faut fournir un document "un peu légal" qui amène des preuves datées sur le bien fondé de votre demande : mettre en cause le cybersquattage, votre légitimité par ancienneté… Vous pouvez passer par un prestataire pour préparer le dossier mais sachez qu'en étant très rigoureux cela peut aussi se faire seul. *Il existe aussi la procédure PARL de l'AFNIC (plus chère) mais nous ne l'avons jamais expérimentée* ;
iii. Pour toute extension non gérée par l'AFNIC, il faut passer par l'ICANN et l'UDRP (solution la plus complexe puisque la procédure est internationale et que chaque pays a sa propre législation. Donc vos arguments ne sont peut-être pas valables… On vous déconseille d'y perdre du temps).

> **REMARQUE**
>
> L'État n'est pas privilégié pour les demandes AFNIC, et n'a donc pas de passe-droit. Vous devez monter un dossier "béton" comme le ferait une entreprise pour remporter la procédure.

Si votre produit doit avoir une notoriété incontournable, vous pouvez réserver d'autres extensions connues comme `.info`, `.org`, `.eu`… Vous éviterez ainsi les sites qui veulent abuser de votre notoriété pour mettre en avant leur propre contenu. Il faut juste garder en tête que cela a un coût récurrent, et ne pas chercher à toutes les avoir car il en existe plus de 1500…

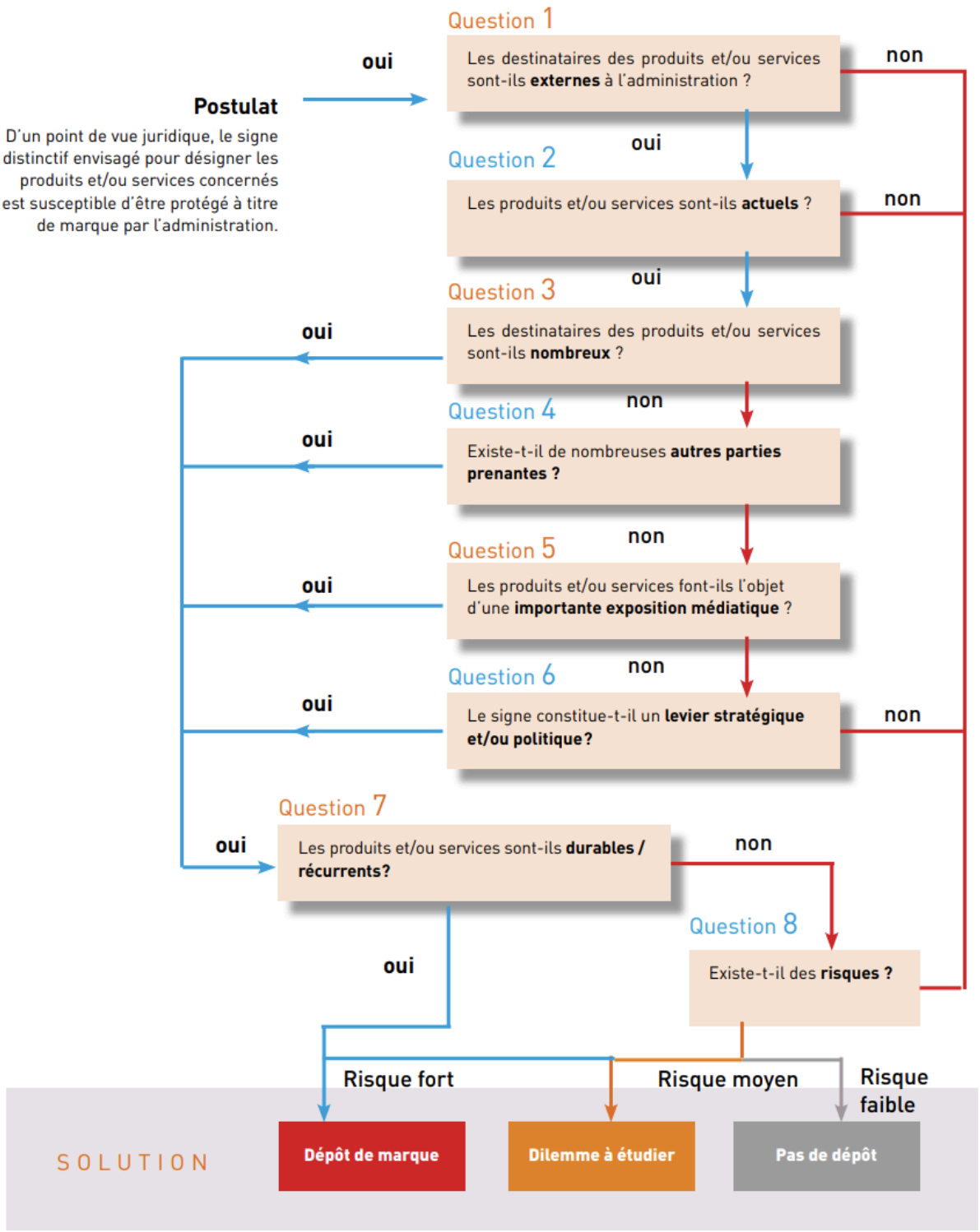
Sachez aussi qu'en tant que service public établi et reconnu vous pouvez prétendre à un nom de domaine terminant par `.gouv.fr`. C'est une extension à part entière (surprenant hein ?), vous pouvez donc le réserver chez n'importe quel registrar de noms de domaine. Il faudra par contre avoir l'autorisation explicite du SIG (service d'information du gouvernement).

La marque

Dans la même veine que le fait de sécuriser ses noms de domaine, il peut être judicieux de déposer la marque du produit pour se protéger de toute usurpation. Nous allons étoffer notre propos avec les éléments que nous avons pu recueillir auprès de l'APIE (voir leur documentation).

> **REMARQUE**
>
> Comme pour les noms de domaine, l'APIE ne gère pas les dépôts pour les agences comme l'ANCT. Par contre elle a quand même un rôle sans distinction d'accompagnement, de publication et de formation. Leur recommandation si vous n'êtes pas dans un ministère est d'avoir un juriste en interne capable de le faire.

Arbre de décision



Savoir si un dépôt de marque est justifié (*source*)

Avant toute sortie d'un produit numérique il est important de vérifier auprès de l'INPI que vous ne portez pas atteinte à une autre marque avec le nom de votre produit. Vous êtes d'ailleurs tenu de faire la vérification avant une quelconque demande de dépôt. Le dépôt de marque est à prendre avec sérieux : le produit peut être contraint à changer de nom même une fois lancé s'il y a litige.

L'enveloppe Soleau (ou e-Soleau) peut faire office de "première étape". Elle coûte ~15 euros et peut être un outil de datation en cas de litige, par contre ce n'est en aucun cas un droit. Donc pour tout ce qui touche à des produits de communication (comme "France Relance") c'est une bonne approche. Sa datation permet juridiquement de se défendre contre du "parasitisme", mais c'est très long et coûteux. Alors qu'une marque déposé a des recours extra-judiciaires ;

Pour le cas d'un dépôt de marque, il y a 4 critères importants :

- La pérennité ;
- Est-ce qu'il identifie un service par un public ;
- Usage externe à l'émetteur (hors État dans notre cas), prévoir l'évolutivité du produit si c'est d'abord en interne puis ouvert aux citoyens ;
- Enjeu médiatique ou usurpation.

Le coût varie entre 200 et 800 euros pour une durée de 10 ans (même coût au renouvellement), ce coût est supporté par l'administration demandeuse de la marque. Et l'APIE peut faire une analyse si vous détectez une utilisation litigieuse de votre marque. *À noter que l'ANSSI surveille le cyber-squattage pour les produits des ministères.*

Le dépôt de marqie doit avoir une couverture appropriée :

- Pour une marque purement française, que pour la France ;
- Pour une marque comme HVE, au niveau européen ;

- Pour quelque chose de mondial c'est très compliqué et très cher car il faut surveiller l'usage des marques localement, se défendre localement… l'intérêt est discutable.

> **UTILE**
>
> Il faut joindre le logo à la demande de dépôt de marque car il peut aider à différencier des marques déjà déposées afin que la demande passe quand même.

Si avec tous ces éléments vous estimez qu'un dépôt pour votre produit est utile, utilisez ce formulaire. Si vous avez un doute, n'hésitez pas à contacter l'APIE, ils sont là pour vous guider.

Aller plus loin

Stratégie pour développer plusieurs produits numériques

La base de connaissances présentée essaie de mettre en avant des concepts sans vous forcer la main concernant leur adoption.

Contribuer à cette base de connaissances

Pour être transparents, il n'est pas encore déterminé comment les contributions pourront être acceptées puisqu'il faut que l'on garde un lien entre chaque partie, mais ...

Décliner cette base de connaissances

Il est possible que vous soyez en accord avec une majorité de notre contenu au point de vouloir reprendre la base de connaissances et d'y apporter vos spécificités po...

Chapitres bientôt à venir

Quelques idées de chapitres qui devraient voir le jour prochainement :

Stratégie pour développer plusieurs produits numériques

La base de connaissances présentée essaie de mettre en avant des concepts sans vous forcer la main concernant leur adoption.

Une fois que vous avez des idées bien à vous sur le langage (ou les) à utiliser, quel hébergeur… Il peut être intéressant de mutualiser et répliquer vos choix sur vos futurs projets :

- Il n'y a plus à remettre en question 90% des bases techniques à chaque fois, vu que ces bases n'ont quasi pas évolué dans la communauté depuis des années (framework, router, base de données… majoritairement rien de nouveau sous le soleil ☀) ;
- Si chaque équipe avance en autonomie, vous garantissez ne pas vous retrouver avec des produits exotiques que personne n'a envie de reprendre ;
- Vous vous facilitez le recrutement ! Chaque personne recrutée sera autant compétente sur le projet A que sur le projet B. Les personnes ont une certaine transversalité ;
- Les équipes peuvent se comprendre entre elles, et donc s'entre-aider ;
- Vous facilitez le travail des métiers transverses qui vont venir chapeauter la sécurité, l'opérationnel…

Bien entendu il faut rester lucide. Pour une organisation très importante (donc avec un historique conséquent), il y aura forcément de l'hétérogénéité dû à

l'organigramme et aux évolutions technologiques au fil du temps.

… Par contre, pour une plus petite organisation, ayant des projets lancés sur une fenêtre de 3-4 ans, la mutualisation est tout à fait envisageable pour éviter de surcomplexifier la structure interne.

> **QUE L'ON S'ENTENDE**
>
> Si la stack "passe-partout" que vous aviez adoptée semble ne pas convenir à la majorité des projets, il peut être judicieux de la remettre en cause plutôt que forcer d'autres projets à la subir. Voir si les gens en sont satisfaits ou non est un bon indicateur 🚦.

Contribuer à cette base de connaissances

Pour être transparents, il n'est pas encore déterminé comment les contributions pourront être acceptées puisqu'il faut que l'on garde un lien entre chaque partie, mais aussi que les modifications fassent sens pour nous et pour le document.

La subjectivité, qui est forcément présente dans notre rédaction, peut malheureusement être un frein aux contributions. Il est préférable dans un premier temps de partager les thématiques de modifications envisagées via la messagerie ou en ouvrant une issue GitHub afin d'en "débattre".

Fort heureusement, si votre avis et vos modifications ne sont pas pris en compte et que vous estimez que le projet Sillon se fourvoie, il vous est toujours possible, dans une logique *open source*, d'en décliner son contenu.

En ce qui concerne la matérialisation d'une contribution, tout se passe sur le repository Git. *(initialement c'était sur un Word collaboratif, mais la contrainte du format de publication nous a orienté vers un projet technique, qui reste normalement facilement modifiable avec l'interface de GitHub)*

Décliner cette base de connaissances

Il est possible que vous soyez en accord avec une majorité de notre contenu au point de vouloir reprendre la base de connaissances et d'y apporter vos spécificités pour l'exposer en interne.

Nous n'avons pas de suggestions parfaites à vous proposer, mais nous sommes curieux d'avoir vos retours si vous en avez d'autres.

Faire un fork complet du code

🟢 Avantages	🔴 Inconvénients
Flexibilité totale pour le modifier	Ne va pas bénéficier des corrections et améliorations facilement (il est toujours possible d'essayer de rebase depuis le code original mais cela peut se complexifier si jamais on change la structure des chapitres)

L'utiliser comme une documentation à tiroirs

🟢 Avantages	🔴 Inconvénients
En utilisant un outil de documentation comme Docusarus (ou autre) vous pourriez importer seulement les sections qui vous intéressent (actuellement dans des fichiers Markdown séparés) et écrire les vôtres à côté	Vous êtes cantonnés à des imports bruts des sections, sans granularité en leur sein

Chapitres bientôt à venir

Quelques idées de chapitres qui devraient voir le jour prochainement :

- Synchroniser son environnement de développement avec ses collègues (`prettier`, outillage, `asdf`...)
- Présentation des capacités de Storybook pour aller plus loin que le design system : y intégrer tous vos états applicatifs
- Synchroniser ses documents légaux avec des pages DSFR
- Rich text et WYSIWYG, où en est-on ?
- Quid de l'`infra as a code` ?
- Le bug bounty
- Proposer du 2FA dans son produit (quand on est hors OIDC)
- Préparer son départ du projet
- Arrêter un projet (enjeux sur les données...)
- Gestion des erreurs : différencier celles techniques de celles métiers (et les remonter dans Sentry)
- Quels outils pour faire une landing page DSFR simplement