

5XUI-1

PROJET WEB UI/UX

Thibault Six

25 chapitres

SUPPORTS DE COURS

https://cours.tsix.be/cours/projet-web-uiux

Version 12 — Chapitre Objets et JSON (avant Ref et V-model) · 11/10/2026

PRÉSENTATION

5XUI — Projet Web UI/UX

Slides de présentation du cours

deck1presentationcours PDF

Pourquoi ce cours ?

Vous savez coder une page. Ce cours vous apprend à **concevoir un site** : partir d'un humain et d'un besoin, le formaliser dans un cahier des charges, lui donner une identité visuelle, le réaliser avec un framework moderne, puis le mettre en ligne et le faire trouver.

C'est aussi la **répétition générale de votre travail de fin d'études** : le cahier des charges que vous rédigez ici est celui de votre TFE, et le portfolio que vous construisez est votre premier vrai produit, dont vous êtes le client.

Acquis d'apprentissage visés

Au départ d'un projet individuel validé par le chargé de cours, en prenant en compte les principes généraux d'accessibilité, vous serez capables :

1. de **produire un cahier des charges et son dossier technique** répondant à la proposition ;
2. de **produire les documents UI, UX et d'inclusive design** (accessibilité) permettant d'y répondre ;
3. de **produire et déployer l'application web** répondant aux consignes ;
4. de **justifier vos choix** dans la présentation de la démarche UX réalisée.

Pour le degré de maîtrise, on tient compte de la précision et de la complétude du cahier des charges, de la qualité des documents UI/UX, de la pertinence des choix techniques et de leur justification, et de votre autonomie face à la documentation.

Grille d'évaluation

- Fiche UE 2026-2027 (PDF) PDF
- Grille d'évaluation (PDF) PDF

Deux volets, un fil rouge

Le cours alterne deux volets qui se répondent :

Volet A — Design & développement	Volet B — Concevoir et piloter un projet
Révisions **p5.js** (dessin, couleurs, variables, conditions, boucles, art génératif), intégration avec **Vite**	Qu'est-ce qu'un cahier des charges, et pourquoi c'est un **brief**
Design : tendances web 2026, logo, charte graphique, maquettes	Client, objet et objectifs du site, public cible et **persona**
Vue.js : Node/NPM, structure d'un projet, **TailwindCSS**, composants, `ref`, `v-model`, `v-if`, `v-for`, events, Vue Router	Concurrence, contenu, arborescence et navigation
Déploiement : Vercel, hébergement OVH	Charte graphique, accessibilité
Référencement : SEO, balises OG, Google Analytics	Planning, budget, contraintes techniques, justification des choix

Le **fil rouge** est votre **portfolio personnel** : vous en dessinez le logo, la direction artistique et les maquettes pendant le volet design, puis vous le codez en Vue + Tailwind, le déployez sur Vercel et le référencez. À la fin de l'année, il est en ligne et il vous ressemble. C'est ce que vous présentez à l'examen, avec le cahier des charges de votre TFE.

L'IA

Les assistants, les générateurs d'images et les agents sont **autorisés et encouragés**. Mais l'IA n'a ni intention, ni goût, ni client : c'est vous qui les apportez. Vous êtes le **directeur artistique** de vos outils. On vous évalue sur la direction, pas sur l'exécution. Le chapitre d'introduction y est consacré. Et pour diriger, il faut savoir de quoi on parle : c'est pour ça qu'on apprend aussi les bases de Vue.js à la main.

Ce cours et ses documents ont été écrits en collaboration avec Claude Fable 5 et GPT-5.6.

Sommaire

- 5XUI — Projet Web UI/UX

1 Introduction
- 1.1 Le carnet d'inspiration
- 1.2 Le test du slop

2 P5.js
- 2.1 Outils de dessin
- 2.2 Couleurs
- 2.3 Premières formes
- 2.4 Variables
- 2.5 Conditions
- 2.6 Boucles
- 2.7 Art Génératif

3 Vue.js
- 3.1 CLI
- 3.2 NodeJS et NPM
- 3.3 Installation
- 3.4 Extensions VS Code
- 3.5 Structure d'un projet
- 3.6 TailwindCSS
- 3.7 Premiers pas avec Tailwind
- 3.8 Components
- 3.9 Objets et JSON
- 3.10 Ref
- 3.11 V-model
- 3.12 Vercel
- 3.13 Hébergement OVH

CHAPITRE 1

Introduction — Avec le nouveau GPT, c'est fini le design ?

> *Chapitre d'ouverture du cours 5XUI · Projet Web UI/UX · 2026-2027*

Slides du chapitre

deck2introdesignia PDF

1. Exercice : le prompt générique

Commençons par ce que vous avez tous en tête. On ouvre un générateur, on lui demande un site, et trente secondes plus tard on a une page. Propre, responsive, en ligne en deux clics. Alors, à quoi sert un webdesigner en 2026 ?

Plutôt que d'en parler, faisons-le.

Consigne. Choisissez un sujet de petit site vitrine : un food truck, un salon de coiffure, un club de tennis de table, un photographe, un gîte. Écrivez un prompt **générique**, comme le ferait quelqu'un qui n'a jamais réfléchi au design : « Fais-moi un site moderne et épuré pour un food truck. » Lancez-le dans l'outil de votre choix. Quinze minutes.

Puis regardez le résultat, et posez-vous une seule question : **qui a décidé quoi ?**

Personne n'a décidé que ce food truck parlait aux étudiants du campus plutôt qu'aux familles du dimanche. Personne n'a décidé que la couleur devait ouvrir l'appétit plutôt que rassurer. Personne n'a décidé de ce que le visiteur doit faire en arrivant, ni pourquoi il reviendrait. La page ressemble à toutes les autres parce que **personne n'a rien demandé de précis**. Comparez vos résultats entre vous : ils se ressemblent tous.

2. Le mot clé, c'est « demande »

L'IA fait tout ce qu'on lui demande. C'est vrai, et c'est précisément le point : elle fait ce qu'on lui *demande*. Elle ne fonctionne pas en autonomie complète. Il lui faut un objectif, un contexte, un public, une intention. Sans ça, elle produit la moyenne statistique de tout ce qu'elle a vu, ce qu'on appelle aujourd'hui le **slop** : lisse, générique, interchangeable, oubliable.

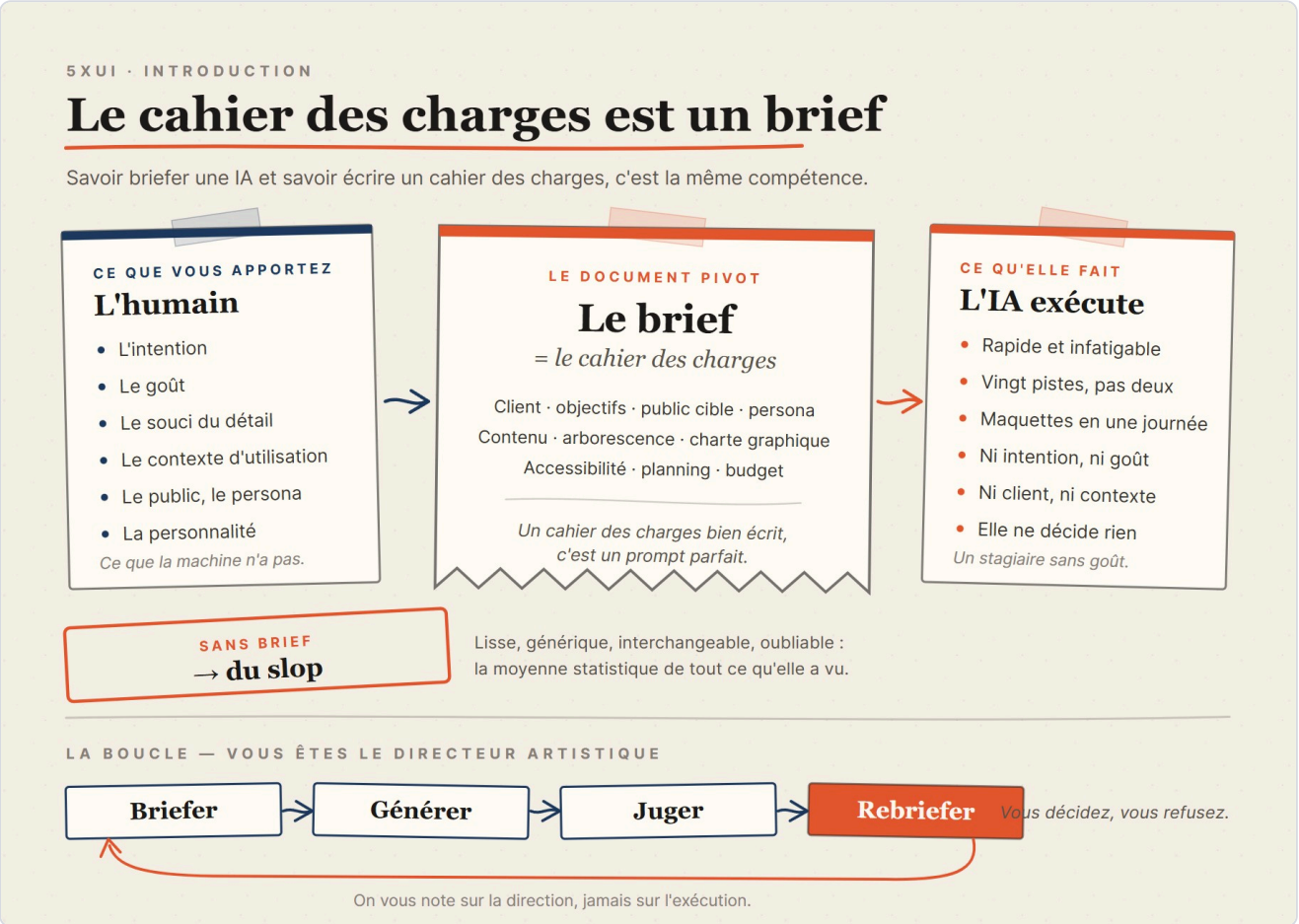
Ce qui manque à la machine, c'est exactement ce que ce cours vous apprend.

Le cahier des charges est un brief

Voici le pont entre les deux volets de ce cours. Pendant des années, on a enseigné le cahier des charges comme un document administratif : le client, les objectifs, le public cible, le contenu, la charte, le planning, le budget. C'est toujours vrai. Mais en 2026, ce document est devenu autre chose : c'est **le brief que vous donnez à vos outils**.

Un cahier des charges bien écrit, c'est un prompt parfait. Un objectif clair, un persona précis, une tonalité définie, une arborescence pensée, une charte graphique argumentée : donnez ça à une IA et elle produit quelque chose de juste. Donnez-lui « fais-moi un site sympa » et elle produit du slop.

Savoir briefer une IA et savoir écrire un cahier des charges, c'est **la même compétence**. Celui qui sait formuler une intention dirige la machine. Celui qui ne sait pas la subit.



3. Exercice : le prompt révisé

Reprenez votre sujet. Cette fois, **décidez** avant d'écrire. Répondez à ces questions sur papier, en cinq minutes :

- **Pour qui ?** Une personne précise : son âge, ce qu'elle cherche, ce qu'elle craint, où elle est quand elle ouvre le site.
- **Pour quoi faire ?** Une seule action attendue du visiteur : appeler, réserver, venir, commander.

- **Quelle ambiance ?** Trois adjectifs, et trois que vous refusez.
- **Quelle référence ?** Une image de votre téléphone, un lieu, une affiche, une pochette de disque que le site devrait évoquer.
- **Quel détail ?** Une idée qui n'appartient qu'à ce projet : le camion est orange, le coiffeur collectionne les vinyles, le club a 60 ans.

Réécrivez le prompt avec ces décisions. Relancez. Comparez avec la première version, et avec les résultats des autres.

Ce que vous devriez constater : les pages ne se ressemblent plus. Certaines sont ratées, mais elles sont ratées **de façon personnelle**. Et c'est vous qui avez fait la différence, pas l'outil.

4. L'IA, partenaire d'idéation et d'expérimentation

Diriger, ce n'est pas seulement filtrer. L'IA est aussi le meilleur partenaire d'**exploration** que vous aurez jamais, précisément parce qu'elle est rapide et qu'elle ne se vexe pas :

- Demandez **dix logos** différents, pas un. Gardez les deux qui vous parlent, demandez dix variantes de chacun.
- Demandez **vingt palettes** de couleurs à partir d'une photo de votre carnet d'inspiration.
- Demandez **cinq maquettes** de la même page, dans cinq directions opposées.
- Puis **croisez** : la typographie de la troisième avec la grille de la première, le ton de la cinquième. Itérez sur ce que vous préférez. Jetez le reste sans regret.

Explorer vingt directions au lieu de deux, produire des maquettes en une journée et passer le reste de la semaine à les rendre bonnes : voilà ce que ces outils changent pour vous. Mais à chaque étape, c'est **vous qui choisissez, qui croisez, qui tranchez**. L'outil propose, l'étudiant dirige.

5. Diriger, c'est aussi équiper

Il n'y a pas que diriger qu'il faut savoir faire. Un bon directeur artistique donne à son équipe les moyens de travailler. Avec un agent, c'est pareil : il faut **se mettre à sa place**.

- **Connaître ses capacités et ses limites.** Qu'est-ce qu'il fait bien, qu'est-ce qu'il rate, qu'est-ce qu'il ne peut pas voir ?
- **Se demander comment il va résoudre le problème.** Quelles infos lui manquent ? Quelles décisions avez-vous prises dans votre tête sans les lui dire ?
- **Lui fournir un vrai environnement.** Un ordinateur avec ce qu'il faut pour travailler, un navigateur pour tester ce qu'il produit, un projet bien structuré.
- **L'orienter vers les bons outils.** Un bon framework, une bonne bibliothèque, un bon package. Un agent qui part sur un mauvais outil produit du mauvais travail, vite.

C'est pour ça qu'on apprend aussi les **bases de Vue.js à la main** dans ce cours. Pas pour concurrencer la machine, mais pour savoir de quoi on parle : pour diriger, pour l'aider quand elle est coincée (ça arrive de moins en moins, mais ça arrive), et surtout pour l'orienter vers les bons outils pour le travail qu'on lui demande. Celui qui n'a jamais écrit un composant ne peut pas juger un composant.

6. À la main, avec la machine, ou les deux ?

Il y a trois façons de traverser ce moment, et elles sont toutes légitimes.

Tout à la main. Certains designers refusent les générateurs et mettent le cœur à l'ouvrage : dessin, collage, typographie, photo, code écrit ligne par ligne. Ce n'est pas un repli nostalgique. Regardez les tendances 2026 : l'imperfection assumée, le collage, la photocopie, le fait-main deviennent des codes de marque recherchés, **précisément parce que** le monde se noie dans le contenu généré. Quand tout est lisse, la trace de la main vaut cher. Cette voie exige d'être excellent, patient, et d'assumer d'être plus lent.

Tout avec la machine. D'autres embrassent les outils à fond : agents, générateurs, itération massive. C'est la voie de la vitesse et de l'exploration. Son risque est évident : sans œil, sans intention, elle mène droit au slop. Elle ne vaut que si la direction est forte.

Les deux. La plupart d'entre vous, et la plupart des studios, seront ici : la machine pour explorer, itérer, produire vite ; la main pour décider, finir, donner la texture, faire ce que le prompt ne donne pas. L'IA génère, vous finissez à la main.

Choisissez votre voie, mais choisissez-la **en connaissance**. C'est pour ça que ce cours vous fait travailler les deux mains : p5.js, le dessin, Figma et le code d'un côté ; les agents et les générateurs de l'autre. Et regardez votre position : vous êtes jeunes, vous n'avez pas encore de nom, pas de réputation à protéger. Vous n'avez rien à défendre et tout à essayer. C'est le meilleur moment pour tester les trois.

Ce qui a de la valeur sur le marché en 2026

Quelle que soit la voie, ce qui se paie est le même. Ce que la machine n'a pas :

- **L'intention.** Savoir ce qu'on veut dire, à qui, et pourquoi.
- **Le goût.** Reconnaître ce qui est juste et ce qui ne l'est pas, avant de pouvoir l'expliquer.
- **Le souci du détail.** Le pixel qui dépasse, la ligne trop longue, le contraste insuffisant. Le slop est fait de détails que personne n'a regardés.
- **Creuser une idée.** Ne pas s'arrêter au premier résultat. Le premier résultat, c'est la moyenne.
- **La personnalité.** Une direction artistique qui vous appartient, qu'on reconnaît.
- **Se démarquer.** Quand tout le monde a accès aux mêmes outils, la différence, c'est vous.

À ça s'ajoutent deux choses qu'on oublie souvent. D'abord la **communication** : parler à un client, écrire clairement, présenter et défendre un choix. Ensuite le **craft** : Photoshop, Figma, le dessin restent pertinents dès qu'on veut plus de contrôle que ce que donne un prompt. Continuez à apprendre à dessiner : la main pense.

7. Le contrat de l'année

Voici comment on va travailler. L'IA est votre stagiaire ultra-rapide, infatigable et sans goût. Vous êtes son **directeur artistique** : vous décidez, vous briefez, vous équipez, vous jugez, vous corrigez, vous refusez. On vous note sur la direction, jamais sur l'exécution.

Pour diriger, il faut un œil. Et un œil, ça se cultive. C'est l'objet des deux chapitres suivants : le carnet d'inspiration, pour remplir votre base de données visuelle, et le test du slop, pour apprendre à mettre des mots sur ce que vous voyez.

Soyez curieux. Allez à des expos, lisez, regardez des œuvres, consommez de la culture sous toutes ses formes. Travaillez une direction artistique qui soit la vôtre. Vous avez une année, un portfolio à faire et un TFE à préparer. Creusez.

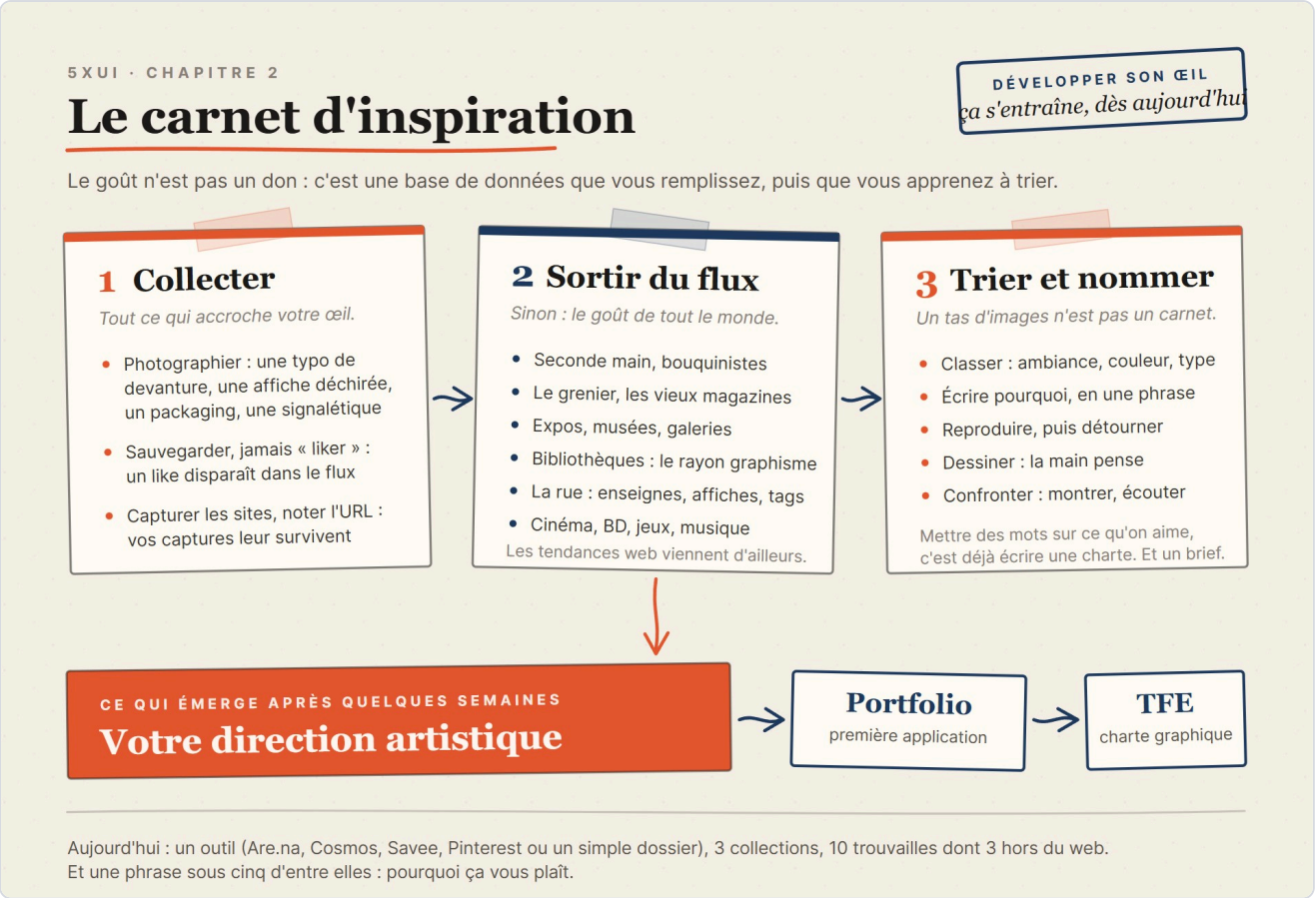
1.1 Le carnet d'inspiration — développer son œil

> *Chapitre 2 du cours 5XUI · Projet Web UI/UX · 2026-2027 · Activité en classe*

Le goût n'est pas un don

On croit souvent que le goût est inné : certains l'ont, d'autres pas. C'est faux. Le goût est une **base de données** que vous remplissez toute votre vie, et que vous apprenez à trier. Plus elle est riche et variée, plus vos choix sont justes, et plus vite vous reconnaissez le slop quand vous le voyez.

Un designer avec un œil, c'est quelqu'un qui a **regardé beaucoup de choses avec attention**. Rien de plus, rien de moins. La bonne nouvelle : ça s'entraîne, dès ce matin.



1. Collecter

Prenez l'habitude de **capturer tout ce qui accroche votre œil**, sans vous demander si c'est utile :

- **Photographiez** : une typo de devanture, une affiche déchirée, un packaging, un ticket de caisse, un carrelage, une signalétique, un vieux néon. Votre téléphone est votre premier outil de designer.
- **Sauvegardez** : une image, une œuvre, un site, une animation que vous croisez sur les réseaux. Ne likez pas, **enregistrez**. Un like disparaît dans le flux, une sauvegarde reste.
- **Capturez des sites** : quand une page web vous plaît, faites une capture d'écran complète et notez l'URL. Les sites disparaissent, vos captures restent.

2. Sortir du flux

Les réseaux sociaux vous montrent ce que tout le monde voit. Si vous ne vous nourrissez que là, vous aurez le goût de tout le monde, c'est-à-dire le goût moyen, celui que l'IA reproduit déjà parfaitement. Pour avoir un œil à vous, il faut aller chercher ailleurs :

- **Les boutiques de seconde main et les bouquinistes** : vieux livres de graphisme, magazines des années 70 à 2000, pochettes de disques, cartes postales. Pour quelques euros, des décennies de mise en page.

- **Le grenier** : les vieux magazines de vos parents ou grands-parents. Regardez la typographie, les couleurs, les publicités. C'est une mine.
- **Les expos, les musées, les galeries** : peinture, photo, architecture, design d'objet, mode. Le webdesign emprunte à tout ça, en retard de quelques années.
- **Les bibliothèques** : le rayon graphisme, typographie, photographie, architecture. Gratuit.
- **La rue** : la typographie urbaine, les enseignes, les affiches de concert, les tags, la signalétique.
- **Le reste de la culture** : cinéma, BD, jeux vidéo, musique, littérature. Un générique de film, une interface de jeu, une couverture de roman : tout est design.

Le web n'est pas la seule source de webdesign. Les tendances web viennent presque toujours d'ailleurs, avec quelques années de retard. Celui qui regarde ailleurs voit les tendances arriver.

3. Trier et nommer

Un tas d'images n'est pas un carnet d'inspiration. Ce qui construit le goût, c'est le **tri** et la **verbalisation** :

- **Classez** : par ambiance, par couleur, par type (typo, layout, photo, animation, logo), par projet.
- **Écrivez pourquoi** : sous chaque image, une phrase. « J'aime le contraste entre la serif énorme et le petit texte en monospace. » « Cette palette est chaude mais pas kitsch grâce au gris. » Mettre des mots sur ce qu'on aime, c'est exactement ce qu'on vous demandera dans une charte graphique, et dans un brief à une IA.
- **Reproduisez puis détournez** : copiez à la main un logo, une mise en page, une palette que vous admirez. Comprenez-la de l'intérieur. Puis faites-en quelque chose à vous.
- **Dessinez** : croquis, wireframes papier, thumbnails. Même mal. La main pense différemment de la souris.
- **Confrontez** : montrez vos trouvailles, dites pourquoi, écoutez ce qu'en pensent les autres. De temps en temps, je vous demanderai de présenter une trouvaille à la classe.

4. Construire votre direction artistique

Après quelques semaines, votre carnet va parler de vous. Des couleurs reviennent, des typos reviennent, une ambiance se dégage. Ce n'est pas un hasard : **c'est votre direction artistique qui apparaît**. Le portfolio que vous allez concevoir cette année en sera la première application, et la charte graphique de votre TFE la deuxième.

Activité : ouvrez votre carnet ce matin

Choisissez **un** outil et créez votre carnet pendant la séance :

Outil	Pour qui	Points forts
Are.na	Les curieux, les chercheurs	Canaux connectés, communauté de designers et d'artistes, zéro algorithme, très respecté dans le milieu
Cosmos	Les visuels	Interface magnifique, extension navigateur, collections, découverte de qualité
Pinterest	Les pragmatiques	Volume immense, tableaux, mais algorithme et beaucoup de slop : triez fort
Savee	Les minimalistes	Grille propre, extension, apprécié des studios
Un dossier + un fichier notes	Les indépendants	Aucun compte, aucune plateforme, vous gardez tout

Ensuite, en 30 minutes :

1. **Créez 3 collections** minimum : par exemple « Typo », « Couleurs », « Sites », ou par ambiance (« Chaud », « Brut », « Éditorial »).
2. **Ajoutez 10 trouvailles** dont **au moins 3 qui ne viennent pas du web**. Fouillez la galerie de votre téléphone : les photos prises ces derniers mois, votre dernière visite d'expo, vos vacances, une devanture, un livre, une affiche. Vous avez déjà un carnet sans le savoir.
3. **Écrivez une phrase** sous 5 d'entre elles : pourquoi ça vous plaît.
4. **Partagez le lien** de votre carnet dans le canal Teams du cours.

À partir de maintenant, ce carnet vit. On y reviendra pour le logo, la charte graphique et les maquettes de votre portfolio.

Galeries et sources pour démarrer

- **Sites** : Siteinspire, Godly, Awwwards, Minimal Gallery, Dark Mode Design, Brutalist Websites
- **Typographie** : Fonts In Use, Typewolf, Google Fonts
- **Graphisme et culture** : It's Nice That, Creative Boom, Eye on Design, Behance
- **Livres** : *Steal Like an Artist* (Austin Kleon), *Ways of Seeing* (John Berger), *Design is a Job* (Mike Monteiro), *Grid Systems* (Josef Müller-Brockmann)

1.2 Le test du slop — mettre des mots sur ce qu'on voit

> *Chapitre 3 du cours 5XUI · Projet Web UI/UX · 2026-2027 · Classe inversée*

C'est quoi, le slop ?

Le mot vient de l'anglais, il désigne la pâtée, la bouillie. Depuis 2024, il désigne **la production générique et sans intention** des outils d'IA : les images lisses aux reflets partout, les textes qui disent tout et rien, les sites qui ont un hero, trois cartes, des témoignages et un footer, et qui pourraient vendre n'importe quoi à n'importe qui.

Le slop n'est pas « fait par une IA ». Le slop, c'est **l'absence de décision**. Un humain pressé qui copie un template produit du slop. Une IA bien briefée par un designer avec un œil n'en produit pas.

Apprendre à reconnaître le slop, c'est apprendre à reconnaître **où il manque une intention**. Et c'est la compétence la plus utile de votre carrière : celle qui vous permet de refuser un résultat et de demander mieux.

Les symptômes

Un design qui sent le slop présente souvent plusieurs de ces signes :

- **Interchangeable** : on pourrait remplacer le logo par celui d'un concurrent, rien ne changerait.
- **Sur-fini, sous-pensé** : tout brille, dégradés, reflets, ombres, mais rien n'a de raison d'être là.
- **La palette de tout le monde** : violet-bleu sur fond sombre, dégradé néon, glassmorphism. Ce n'était pas du slop en 2021. Ça l'est devenu.
- **Le texte remplissage** : « Solutions innovantes pour un monde en mouvement ». Qui parle ? À qui ?
- **Le détail abandonné** : alignements approximatifs, hiérarchie plate, contrastes insuffisants, une typo pour tout, images stock qui ne disent rien.
- **Sans contexte** : rien n'indique pour qui c'est fait, ni ce que le visiteur doit faire.
- **Le trop** : quand on ne sait pas ce qu'on veut dire, on met tout. Le slop est rarement sobre.

À l'inverse, un design avec une intention a **une idée**, on la voit tout de suite, et chaque détail la sert.

Activité : à vous de chasser

C'est vous qui constituez le corpus. Pour la prochaine séance :

1. **Trouvez 6 images** : des pages web, des affiches, des logos, des visuels de réseaux sociaux, des publicités. Un mélange de **slop** et de **pas slop**, dans les proportions que vous voulez. Ne dites pas lesquelles sont lesquelles.
2. Cherchez partout : les galeries de sites, vos réseaux, les pubs qu'on vous impose, les sites des commerces de votre quartier, les templates gratuits, les générateurs d'images, votre carnet d'inspiration.

3. **Déposez-les dans le dossier Teams** prévu pour ça, nommées avec votre prénom et un numéro : `prenom-01.png` à `prenom-06.png`. Pas de légende, pas d'indice.

Pendant la séance, on projette les images une par une, et **on juge ensemble** :

1. **Slop ou pas slop ?** Vote à main levée.
2. **Pourquoi ?** Un ou deux symptômes précis, avec les mots de la liste ci-dessus ou les vôtres.
3. **Pour qui c'est fait ?** Si personne ne sait répondre, c'est déjà un indice.
4. Celui qui a apporté l'image révèle ce qu'il en pensait, et d'où elle vient.

Je glisserai quelques images à moi dans le lot, dont au moins une générée par une IA **bien briefée**. Essayez de les repérer.

On ne cherche pas la bonne réponse, on cherche **les mots**. Le but est que vous sortiez de la séance avec un vocabulaire pour dire ce qui va et ce qui ne va pas : hiérarchie, contraste, rythme, tonalité, cohérence, intention.

Débrief

Trois choses à retenir :

- **L'outil ne fait pas le slop, l'absence de brief le fait.** Si une image générée par une IA bien briefée vous a échappé, c'est le point.
- **Vous savez déjà voir.** Vous avez trouvé la plupart des réponses. Ce qui vous manquait, c'étaient les mots. Ils viendront avec le carnet d'inspiration.
- **Le slop est une tendance qui vieillit vite.** Ce qui était frais en 2021 est du slop en 2026. D'où l'importance de regarder ailleurs que dans le flux.

Pour la suite

Chaque fois que vous générez quelque chose cette année, logo, image, maquette, composant, passez-le au test : slop ou pas slop, pourquoi, pour qui. Si vous ne pouvez pas répondre, ne le gardez pas. Rebriefez. Creusez.

CHAPITRE 2

Introduction à p5.js

> 🎯 ***Objectif*** *: Comprendre ce qu'est p5.js, pourquoi on l'utilise dans ce cours, le mettre en place et découvrir l'interface. À la fin de cette introduction, tu seras capable de lancer ton premier programme.*



Réviser JavaScript en s'amusant

Dans ce cours, tu as déjà appris les bases de **JavaScript** : variables, conditions, boucles, fonctions... Mais écrire du code qui affiche du texte dans la console, avouons-le, c'est pas le truc le plus excitant du monde.

Et si on révisait tout ça en **dessinant** ? En créant des animations, des formes colorées, des programmes visuels ?

p5.js, c'est exactement ça : une **bibliothèque JavaScript** conçue pour rendre le code **visuel et créatif**. Tu écris du JavaScript, tu rafraîchis ta page, et tu vois directement le résultat à l'écran.

L'avantage ? Tu révises **toute la syntaxe JavaScript** que tu connais déjà (variables, `if`, `for`, fonctions...) mais avec des résultats immédiats et visuels. C'est le même langage, juste avec des super-pouvoirs graphiques !

Un peu d'histoire

p5.js est né du projet **Processing**, créé en **2001** au **MIT** (Massachusetts Institute of Technology) par **Ben Fry** et **Casey Reas**. Leur objectif : permettre aux **artistes**, designers et non-programmeurs de créer des œuvres visuelles avec du code, sans avoir besoin d'un diplôme en informatique.

Le projet a tellement bien marché qu'il a reçu le **Golden Nica** du prestigieux festival Ars Electronica en 2005.

En **2013**, **Lauren Lee McCarthy** a créé **p5.js**, une version de Processing qui tourne directement dans le **navigateur web** grâce à JavaScript. Aujourd'hui, p5.js est utilisé dans le monde entier pour l'art numérique, la visualisation de données, et surtout... **l'apprentissage de la programmation**.

En **2025**, p5.js a franchi une étape importante avec la sortie de la **version 2**, une refonte de la bibliothèque qui modernise sa façon de gérer les événements (souris, clavier) et le chargement des fichiers. C'est cette version qu'on utilise dans ce cours.

Pourquoi p5.js dans un cours de Scripts Clients ?

Voici le secret : p5.js, c'est du **JavaScript pur**. Tout ce que tu as appris en JavaScript fonctionne directement dans p5.js.

JavaScript classique	p5.js
`let x = 100;`	`let x = 100;`
`if (x > 50) { }`	`if (x > 50) { }`
`for (let i = 0; i < 10; i++)`	`for (let i = 0; i < 10; i++)`
`function maFonction() { }`	`function maFonction() { }`
`console.log("hello")`	`console.log("hello")`

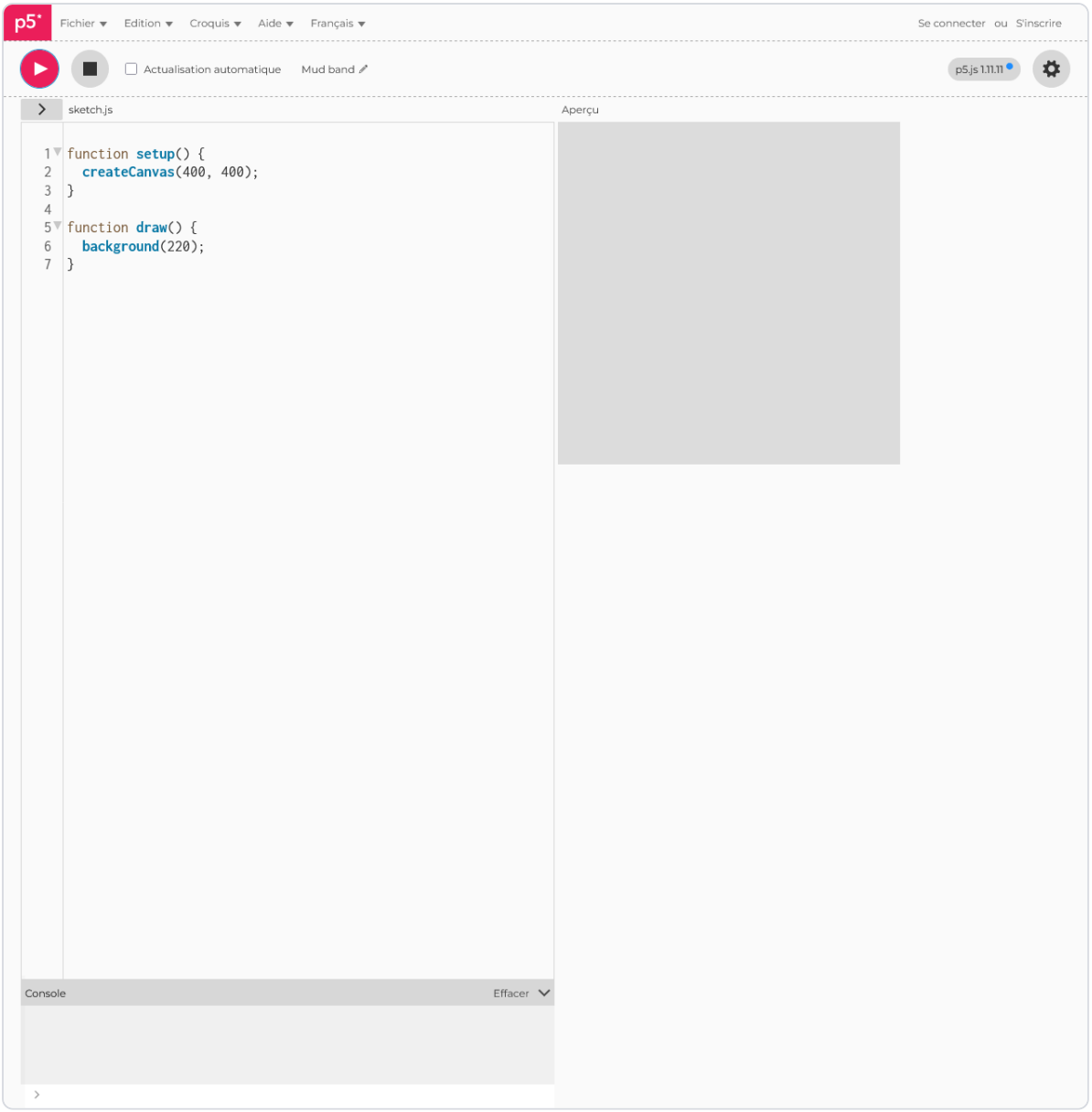
Tu vois ? C'est **exactement le même langage**. p5.js ajoute simplement des fonctions pour dessiner (`rect()`, `ellipse()`, `fill()`...) et une structure de base (`setup()` et `draw()`) pour organiser ton programme.

💡 **Fun fact** : p5.js peut interagir avec le DOM, gérer du son, de la vidéo, de la webcam, et même de la 3D — tout ça en JavaScript dans le navigateur !

Mettre en place p5.js

L'éditeur en ligne (le plus simple)

1. Va sur **https://editor.p5js.org**
2. Tu arrives directement sur un éditeur de code prêt à l'emploi
3. Clique sur ▶ **Play** pour lancer ton programme
4. C'est tout ! Pas besoin d'installer quoi que ce soit.



En local (dans ton projet)

Si tu préfères travailler en local, il suffit d'inclure la bibliothèque p5.js dans ton fichier HTML :

```xml
<!DOCTYPE html>
<html>
<head>
  <script src="https://cdn.jsdelivr.net/npm/p5@2/lib/p5.min.js"></script>
</head>
<body>
  <script src="sketch.js"></script>
</body>
</html>
```

Puis tu crées un fichier `sketch.js` avec ton code p5.js. Ouvre le fichier HTML dans ton navigateur et c'est parti !

Quelle version de p5.js ?

Ce cours est écrit pour **p5.js version 2** (v2). C'est la version par défaut de l'éditeur en ligne et du site officiel depuis 2025, et celle que tu obtiens avec le lien CDN ci-dessus (`p5@2`).

Pourquoi c'est important ? Parce que beaucoup de tutoriels, de vidéos YouTube et d'exemples générés par les IA ont été écrits pour la **version 1**, et certaines choses ont changé. Pour ce qu'on fait dans ce cours, il y a **une seule différence à retenir** :

Ce que tu veux faire	p5.js v1 (ancien)	p5.js v2 (ce cours)
Savoir quel bouton de la souris est enfoncé	`mouseButton == LEFT`	`mouseButton.left`

On y reviendra dans le chapitre sur les conditions. Le reste (`setup()` , `draw()` , formes, couleurs, variables, boucles...) est **identique** dans les deux versions.

> **ASTUCE**
>
> **Si tu trouves du code v1 sur Internet** : dans l'éditeur en ligne, le menu **Settings → Library Management** permet de choisir la version de p5.js du sketch et d'activer des "compatibility add-ons" qui font tourner l'ancien code. C'est pratique pour dépanner, mais dans ce cours on écrit du **v2** : c'est la version maintenue, et c'est celle que tu retrouveras partout demain.

> **ATTENTION**
>
> **Avec les IA** : quand tu demandes du code p5.js à un assistant, précise **"p5.js version 2"** dans ton prompt. Si le code qu'il te donne contient `preload()` , `mouseButton == LEFT` ou `keyCode == UP_ARROW` , c'est du v1 : demande-lui de le convertir en v2 (`async setup()` + `await` , `mouseButton.left` , `keyIsDown(UP_ARROW)`).

L'interface de l'éditeur en ligne

Quand tu ouvres l'éditeur p5.js, voici ce que tu vois :

Zone	Rôle
Zone de code (grande zone blanche)	C'est là que tu écris ton programme
Bouton ▶ Play (en haut à gauche)	Lance ton programme
Bouton ■ Stop	Arrête ton programme
Console (zone noire en bas)	Affiche les messages et les erreurs
Zone de prévisualisation	S'affiche à droite quand tu cliques sur Play — c'est ton canevas !

💡 **Astuce** : Si tu vois du texte rouge dans la console, c'est une **erreur**. Pas de panique ! Lis le message, il te dit souvent quelle ligne pose problème.

La structure d'un programme p5.js

Tout programme p5.js a la même structure de base :

```javascript
function setup() {
  // Code exécuté UNE SEULE FOIS au démarrage
}

function draw() {
  // Code exécuté EN BOUCLE (60 fois par seconde !)
}
```

Le parallèle avec JavaScript classique

Concept	JavaScript classique	p5.js	Rôle
Initialisation	`window.onload = ...`	`function setup() { }`	S'exécute une seule fois au début
Répétition	`setInterval(...)`	`function draw() { }`	S'exécute en boucle, encore et encore
Zone d'affichage	Le DOM / un `<canvas>`	Le **canevas** (canvas)	La zone où on dessine

Pour créer un canevas, on utilise `createCanvas()` dans le `setup()` :

```javascript
function setup() {
  createCanvas(400, 300);  // Crée un canevas de 400 pixels de large et 300 de haut
}

function draw() {
  // On dessinera ici
}
```

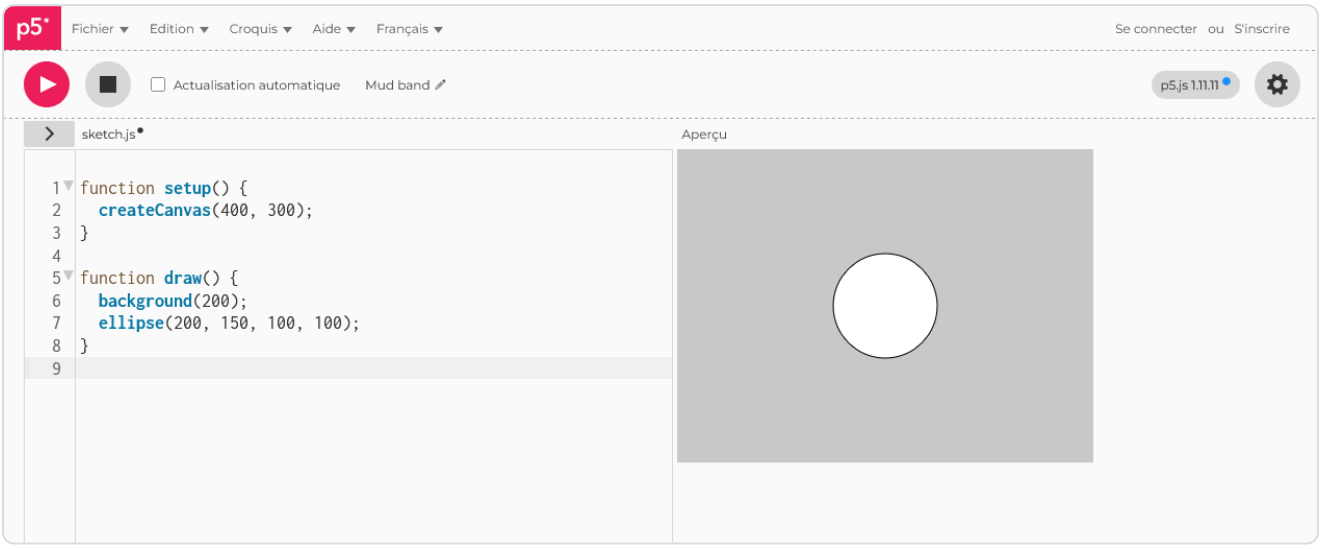
⚠️ **Attention** : `createCanvas()` doit toujours être dans `setup()`, idéalement comme **première instruction**.

Ton premier programme

Tape ce code dans l'éditeur p5.js et clique sur ▶ **Play** :

```javascript
function setup() {
  createCanvas(400, 300);
}

function draw() {
  background(200);
  ellipse(200, 150, 100, 100);
}
```



Tu devrais voir une fenêtre grise avec un cercle blanc au centre. Félicitations, tu viens d'écrire ton premier programme p5.js ! 🎉

Ne t'inquiète pas si tu ne comprends pas encore chaque ligne — on va tout décortiquer dans les prochains chapitres.

Ce qu'on va apprendre dans les prochains chapitres

Voici un aperçu de la suite. Chaque concept utilise du JavaScript que tu connais déjà :

Chapitre	Concept	En JavaScript classique	En p5.js
0b	**Outils de dessin**	Manipuler un `<canvas>`	`rect()`, `ellipse()`, `stroke()`, `fill()`
0c	**Couleurs**	`ctx.fillStyle = "red"`	`fill(255, 0, 0)`, `background()`
1	**Premières formes**	Canvas API complexe	Combiner formes et couleurs
2	**Variables**	`let x = 10;`	`let x = 10;` (identique !)
3	**Conditions**	`if (x > 5) { } else { }`	`if (x > 5) { } else { }` (identique !)

Chapitre	Concept	En JavaScript classique	En p5.js
4	**Boucles**	`for (let i = 0; i < 10; i++)`	`for (let i = 0; i < 10; i++)` (identique !)
5	**Fonctions**	`function dessiner() { }`	`function dessinerMaison(x, y) { }`
6	**Animation**	`requestAnimationFrame()`	Variable qui change dans `draw()`

Les types de données (un avant-goût)

En JavaScript, les variables sont déclarées avec `let` ou `const`. Contrairement à des langages comme Java ou C++, tu ne dois pas déclarer le type — JavaScript le détermine automatiquement. Mais il est important de connaître les types qui existent :

Type	Ce que ça stocke	Exemple
`number`	Un nombre (entier ou à virgule)	`let age = 25;` / `let prix = 9.99;`
`boolean`	**Vrai** ou **Faux**	`let estVisible = true;`
`string`	Du **texte**	`let nom = "Alice";`

💡 En JavaScript, pas besoin de préciser `int` ou `float` comme en Java/C++ — un `number` gère les deux :

```javascript
let x = 100;          // ✅ Nombre entier
let y = 3.14;         // ✅ Nombre à virgule
let nom = "Bonjour";  // ✅ Texte
let ok = true;        // ✅ Booléen
```

p5.js ajoute aussi la fonction `color()` pour manipuler les couleurs facilement :

```javascript
let rouge = color(255, 0, 0);
```

On verra tout ça en détail dans le chapitre sur les variables.

Pour aller plus loin

Si tu veux explorer p5.js par toi-même :

- **Documentation officielle** : https://p5js.org/reference — la liste complète de toutes les fonctions (version 2). Si tu suis un vieux tutoriel, l'ancienne référence v1 est sur https://v1.p5js.org/reference

- **Guide de transition v1 → v2** : https://p5js.org/tutorials/v2_transition/ — court et clair, pour comprendre ce qui a changé
- **The Coding Train** (YouTube) : Daniel Shiffman explique p5.js de façon fun et accessible (attention, beaucoup de vidéos datent de la v1)
- **Nature of Code** (https://natureofcode.com) : un livre gratuit pour les plus avancés, mêlant code et simulation physique
- **Éditeur en ligne** : https://editor.p5js.org — code et teste directement dans le navigateur

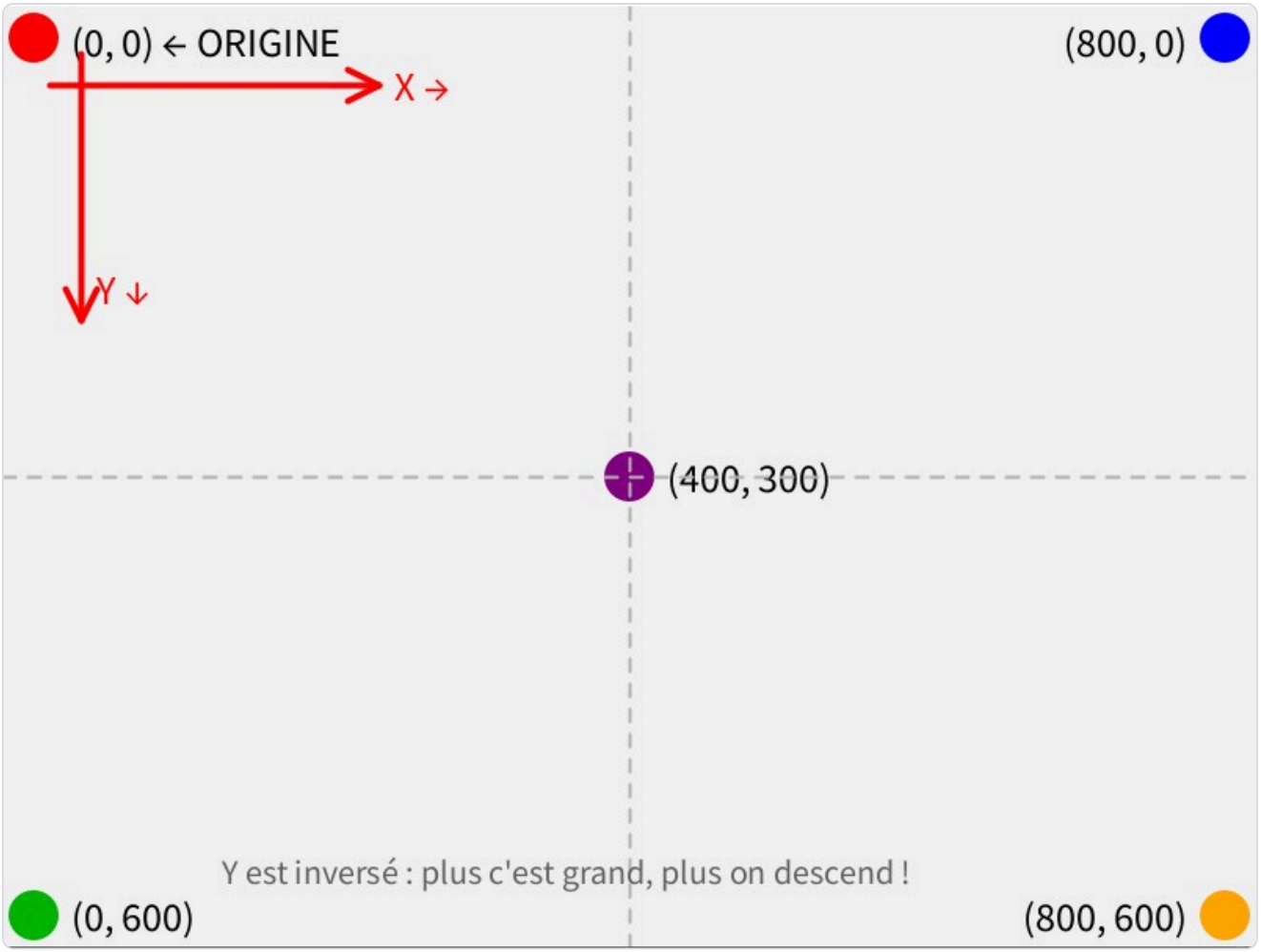
2.1 Les Outils de Dessin

> 🎯 ***Objectif*** *: Maîtriser le système de coordonnées, dessiner toutes les formes de base, et comprendre le fonctionnement du remplissage et du contour. À la fin de ce chapitre, tu seras capable de dessiner n'importe quelle combinaison de formes sur ton canevas.*

Le système de coordonnées

L'origine est en haut à gauche

C'est différent des maths ! En p5.js, le point **(0, 0)** est en **haut à gauche** du canevas. L'axe X va vers la **droite** et l'axe Y va vers le **bas**.



Quand on écrit `ellipse(100, 200, 50, 50)`, on dessine un cercle dont le centre est à **100 pixels depuis la gauche** et **200 pixels depuis le haut**.

⚠️ **Piège classique** : l'axe Y est **inversé** par rapport aux cours de maths. Plus la valeur de Y est grande, plus on **descend** !

Direction	Valeur qui augmente
→ Vers la droite	X augmente
↓ Vers le bas	Y augmente
← Vers la gauche	X diminue
↑ Vers le haut	Y diminue

Les formes de base

p5.js met à ta disposition plusieurs fonctions pour dessiner. Voici les principales :

Le point

```javascript
point(x, y);
```

Dessine un seul pixel à la position (x, y). Pas très visible ! Utilise `strokeWeight()` pour le rendre plus gros.

```javascript
strokeWeight(8);
point(200, 150);   // Un gros point au centre
```

La ligne

```javascript
line(x1, y1, x2, y2);
```

Trace une ligne entre le point (x1, y1) et le point (x2, y2).

```javascript
line(50, 50, 350, 250);  // Ligne diagonale
```

Le rectangle

```javascript
rect(x, y, largeur, hauteur);
```

Dessine un rectangle. Par défaut, **(x, y) est le coin supérieur gauche**.

```javascript
rect(100, 80, 200, 120);  // Rectangle à partir du coin (100, 80)
```

💡 Pour que (x, y) soit le **centre** du rectangle au lieu du coin, utilise :

```javascript
rectMode(CENTER);
rect(200, 150, 200, 120);  // (200, 150) est maintenant le CENTRE
```

L'ellipse (et le cercle)

```javascript
ellipse(x, y, largeur, hauteur);
```

Dessine une ellipse **centrée** sur (x, y). Si largeur = hauteur, c'est un **cercle**.

```javascript
ellipse(200, 150, 100, 100);  // Cercle de diamètre 100
ellipse(200, 150, 150, 80);   // Ellipse aplatie
```

Le triangle

```javascript
triangle(x1, y1, x2, y2, x3, y3);
```

Dessine un triangle en reliant trois points.

```javascript
triangle(200, 50, 100, 250, 300, 250);  // Triangle pointant vers le haut
```

L'arc (portion de cercle)

```javascript
arc(x, y, largeur, hauteur, angleDebut, angleFin);
```

Dessine un arc de cercle. Les angles sont en **radians**, pas en degrés.

Angle	Radians	Constante p5.js
0°	0	`0`
90°	π/2	`HALF_PI`
180°	π	`PI`
270°	3π/2	`PI + HALF_PI`
360°	2π	`TWO_PI`

```javascript
arc(200, 150, 160, 160, 0, PI);  // Demi-cercle (de 0° à 180°)
```

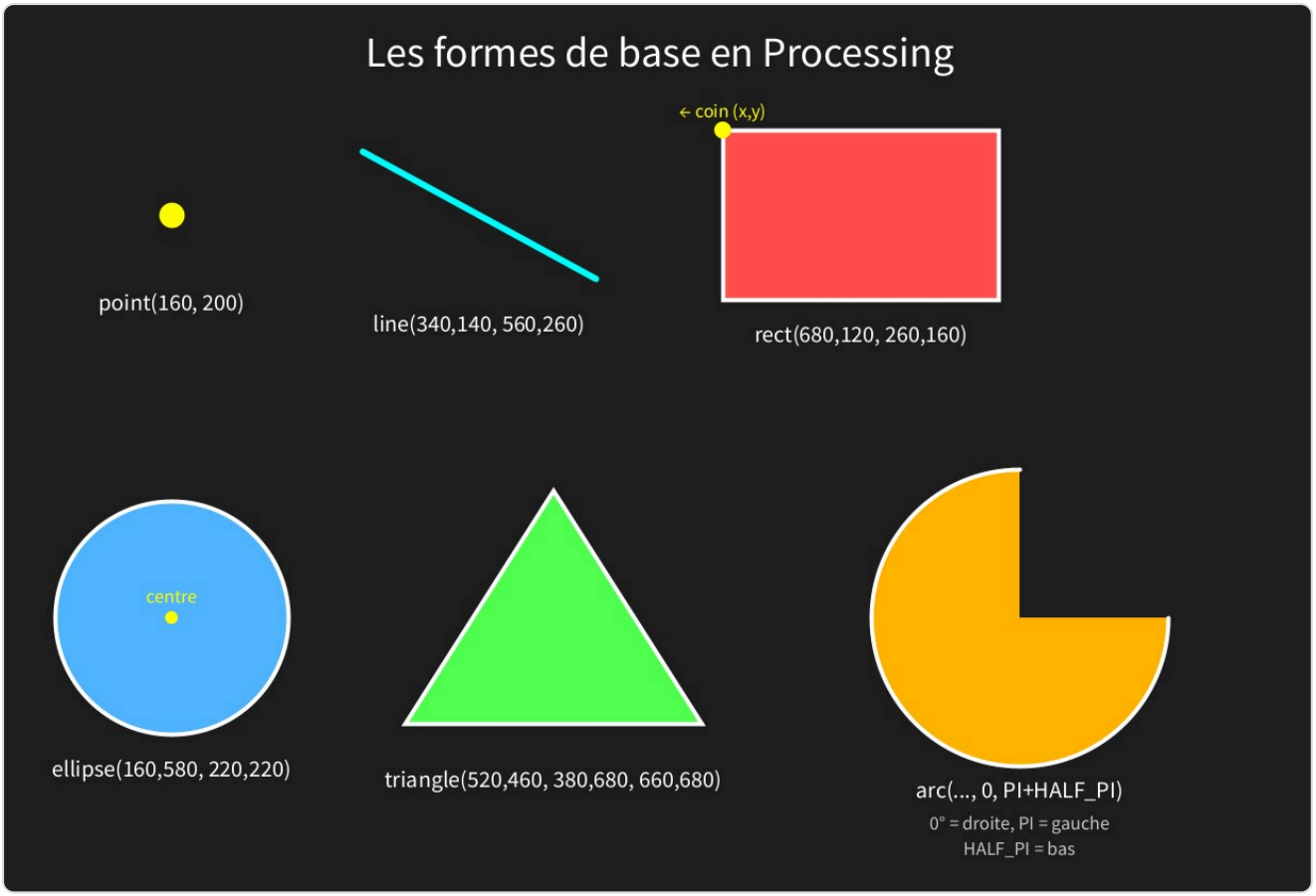


Tableau récapitulatif

Fonction	Paramètres	Dessine...
`point(x, y)`	Position	Un point
`line(x1, y1, x2, y2)`	Deux points	Une ligne
`rect(x, y, l, h)`	Coin + dimensions	Un rectangle
`ellipse(x, y, l, h)`	Centre + dimensions	Une ellipse/cercle
`triangle(x1,y1, x2,y2, x3,y3)`	Trois points	Un triangle
`arc(x, y, l, h, début, fin)`	Centre + dimensions + angles	Un arc
`background(couleur)`	Couleur	Remplit tout le fond

Le remplissage et le contour

p5.js fonctionne comme si tu avais deux outils dans la main :

🎨 Le pot de peinture : `fill()`

`fill()` définit la couleur de **remplissage** (l'intérieur) de toutes les formes dessinées après.

```javascript
fill(200);              // Remplissage gris clair
rect(50, 50, 100, 80);  // Ce rectangle sera gris clair à l'intérieur
```

Pour désactiver le remplissage (forme "vide", transparente) :

```javascript
noFill();
```

✏️ Le stylo : `stroke()` et `strokeWeight()`

`stroke()` définit la couleur du **contour** (le trait autour des formes).

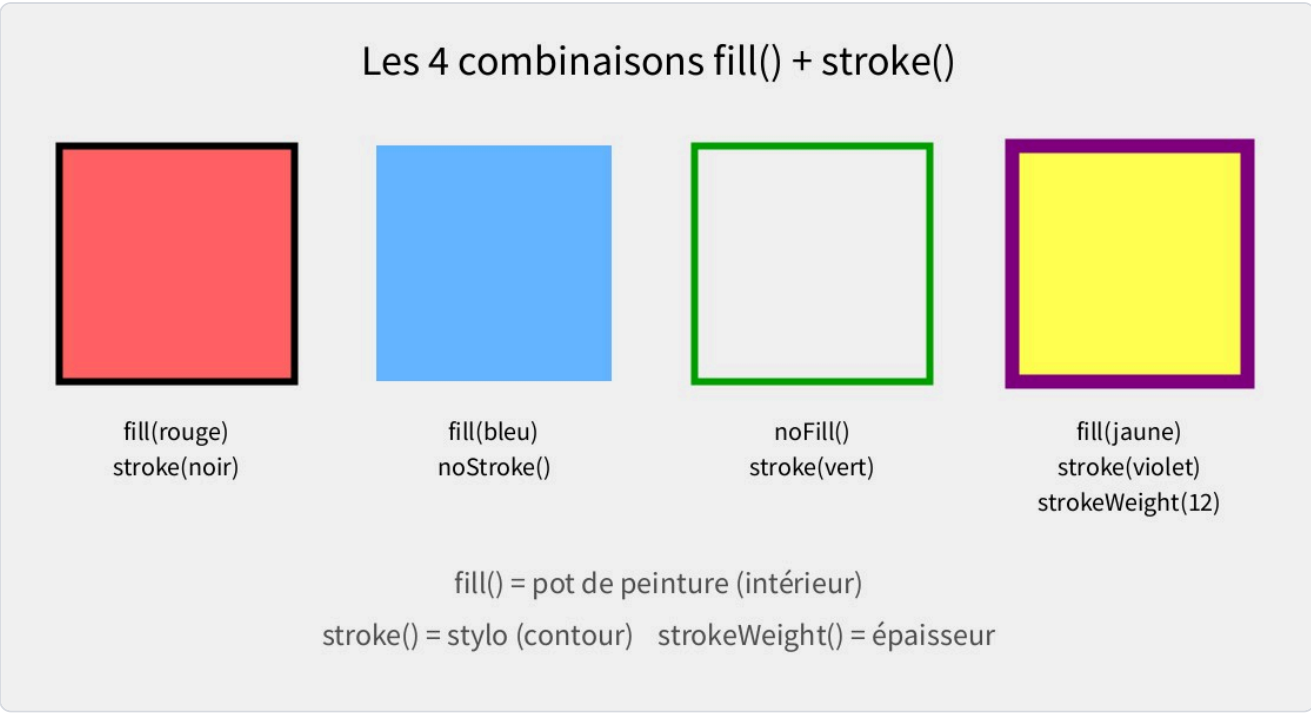
`strokeWeight()` définit l'**épaisseur** du trait en pixels.

```javascript
stroke(0);              // Contour noir
strokeWeight(4);         // Trait de 4 pixels d'épaisseur
rect(50, 50, 100, 80);  // Ce rectangle aura un gros contour noir
```

Pour désactiver le contour :

```javascript
noStroke();
```

Les 4 combinaisons possibles



Combinaison	Code	Résultat
Remplissage + contour	`fill(200); stroke(0);`	Forme grise avec bord noir
Remplissage seul	`fill(200); noStroke();`	Forme grise, pas de bord
Contour seul	`noFill(); stroke(0);`	Juste le contour, intérieur vide
Les deux personnalisés	`fill(200); stroke(0); strokeWeight(5);`	Forme grise avec gros bord noir

💡 Règle importante : l'effet "feutre"

`fill()` et `stroke()` s'appliquent à **tout ce qui est dessiné après**, jusqu'au prochain appel. C'est comme si tu changeais de feutre : une fois que tu prends le rouge, tout ce que tu dessines est rouge jusqu'à ce que tu changes.

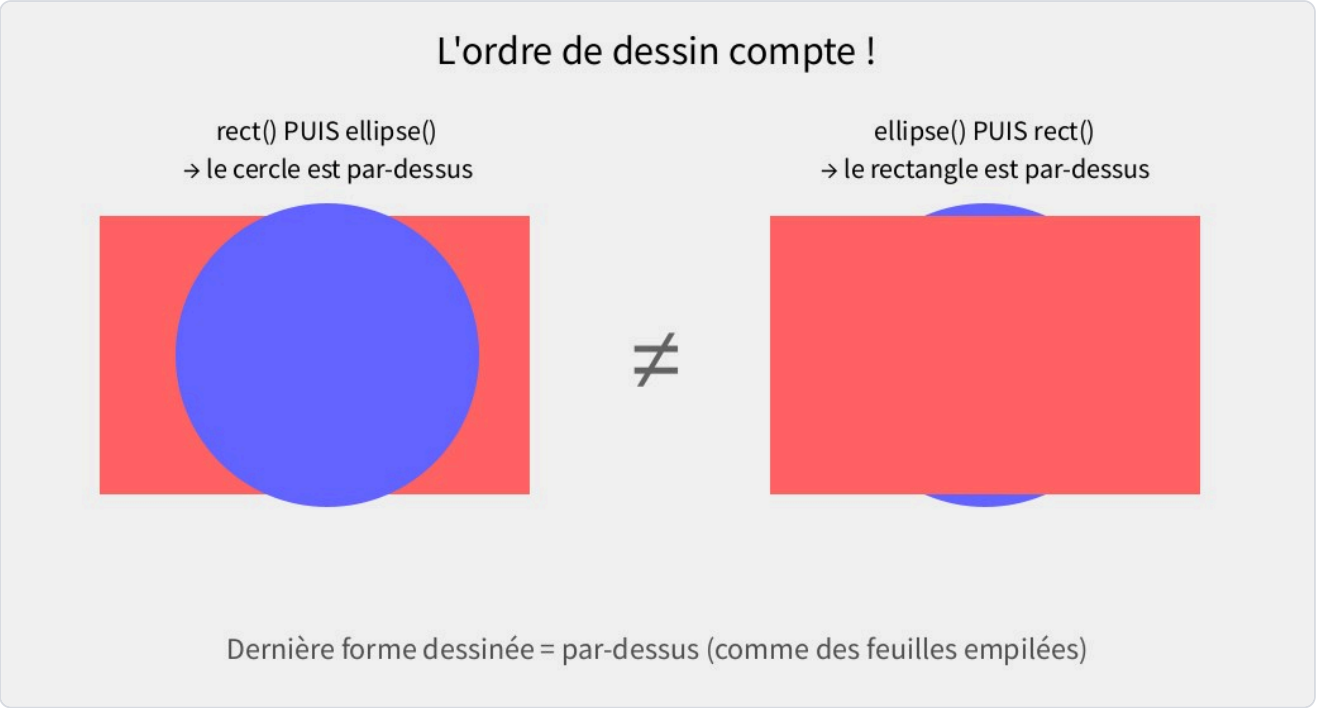
JAVASCRIPT

```
fill(200);                  // "Je prends le feutre gris"
rect(20, 20, 80, 80);       // Rectangle gris
ellipse(200, 60, 80, 80);   // Cercle gris AUSSI !

fill(100);                  // "Je change pour le feutre gris foncé"
rect(300, 20, 80, 80);      // Rectangle gris foncé
```

L'ordre des instructions compte !

En p5.js, les formes sont dessinées **les unes par-dessus les autres**, dans l'ordre du code. La dernière forme dessinée est **par-dessus** toutes les autres.



JAVASCRIPT

```javascript
// Le cercle sera PAR-DESSUS le rectangle
fill(200);
rect(50, 50, 200, 100);     // Dessiné en premier → en dessous

fill(100);
ellipse(150, 100, 120, 120); // Dessiné en dernier → par-dessus
```

C'est comme quand tu empiles des feuilles de papier : la dernière posée est celle qu'on voit.

⚠️ **Conséquence** : si tu veux qu'un fond soit derrière tout, dessine-le **en premier** ! C'est le rôle de `background()` qu'on met au début du `draw()`.

Résumé

Outil	Fonction	Ce que ça fait
🎨 Remplissage	`fill(couleur)`	Couleur de l'intérieur des formes
🎨 Pas de remplissage	`noFill()`	Formes transparentes
✏️ Contour	`stroke(couleur)`	Couleur du trait/bord
✏️ Épaisseur	`strokeWeight(pixels)`	Épaisseur du trait

Outil	Fonction	Ce que ça fait
✏️ Pas de contour	`noStroke()`	Pas de trait autour des formes
📐 Mode rectangle	`rectMode(CENTER)`	Change le point de référence du rectangle
🖼️ Fond	`background(couleur)`	Remplit tout le canevas

💡 **Astuce finale** : quand tu te perds, rappelle-toi que chaque forme a besoin de **trois choses** :

1. Un **remplissage** (`fill()` ou `noFill()`)
2. Un **contour** (`stroke()` / `strokeWeight()` ou `noStroke()`)
3. Sa **position et taille** (les paramètres de la fonction)

2.2 Les Couleurs

> 🎯 ***Objectif*** *: Maîtriser le système de couleurs de p5.js, des niveaux de gris jusqu'aux couleurs RGB avec transparence. À la fin de ce chapitre, tu seras capable de colorier n'importe quelle forme avec la couleur exacte que tu veux.*

Les niveaux de gris

La façon la plus simple d'utiliser les couleurs en p5.js : **une seule valeur** entre 0 et 255.

Valeur	Couleur
`0`	⬛ Noir complet
`64`	Gris foncé
`128`	Gris moyen
`192`	Gris clair
`255`	⬜ Blanc complet

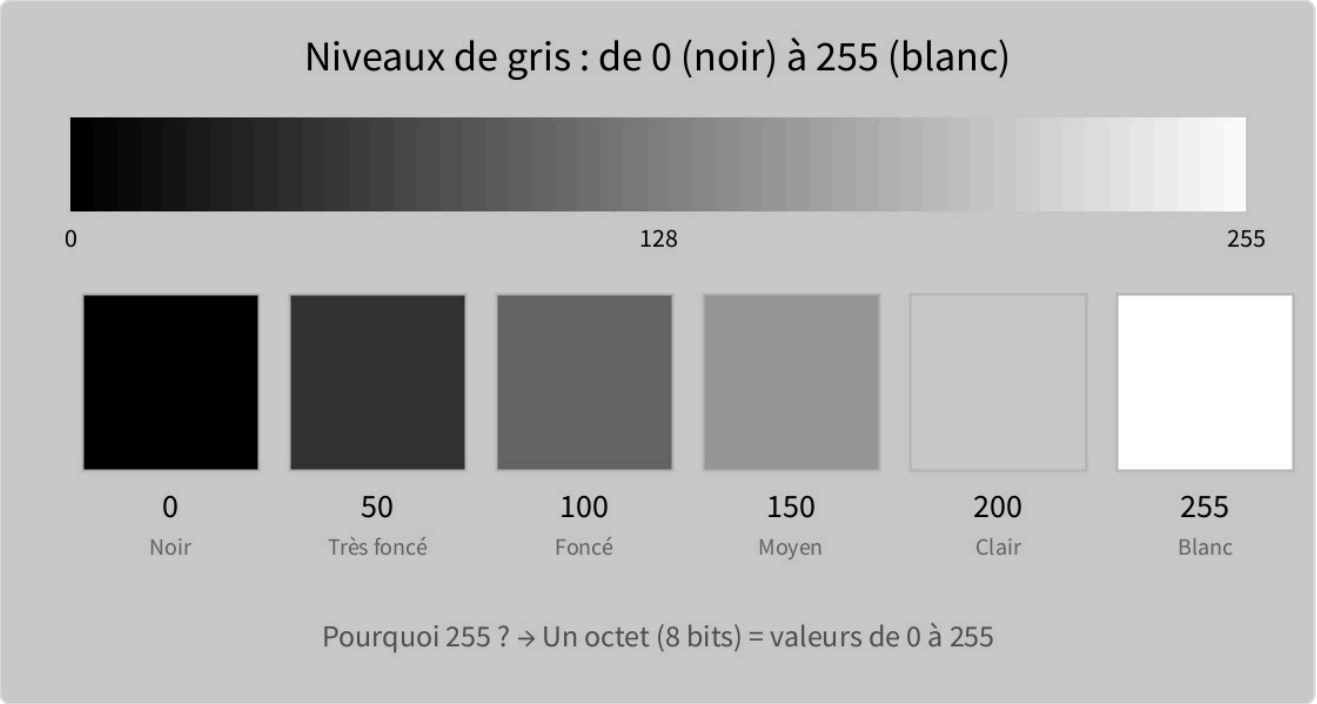
```javascript
background(240);        // Fond gris très clair

fill(0);                // Noir
rect(20, 20, 80, 80);

fill(128);              // Gris moyen
rect(120, 20, 80, 80);

fill(255);              // Blanc
rect(220, 20, 80, 80);
```

💡 **Pourquoi 255 et pas 100 ?** Parce qu'un octet (8 bits) peut stocker des valeurs de 0 à 255. C'est un standard en informatique que tu retrouveras partout : écrans, images, web... D'ailleurs, c'est exactement le même système que les couleurs CSS en `rgb()` !



Les couleurs RGB

Pour obtenir de vraies couleurs, on utilise le système **RGB** : **R**ouge, **V**ert (Green), **B**leu. Chaque composante va de 0 à 255.

```javascript
fill(rouge, vert, bleu);
```

Le principe est celui de la **synthèse additive** : on mélange de la lumière, pas de la peinture ! C'est comme les pixels de ton écran qui combinent trois petites LEDs (rouge, verte, bleue).

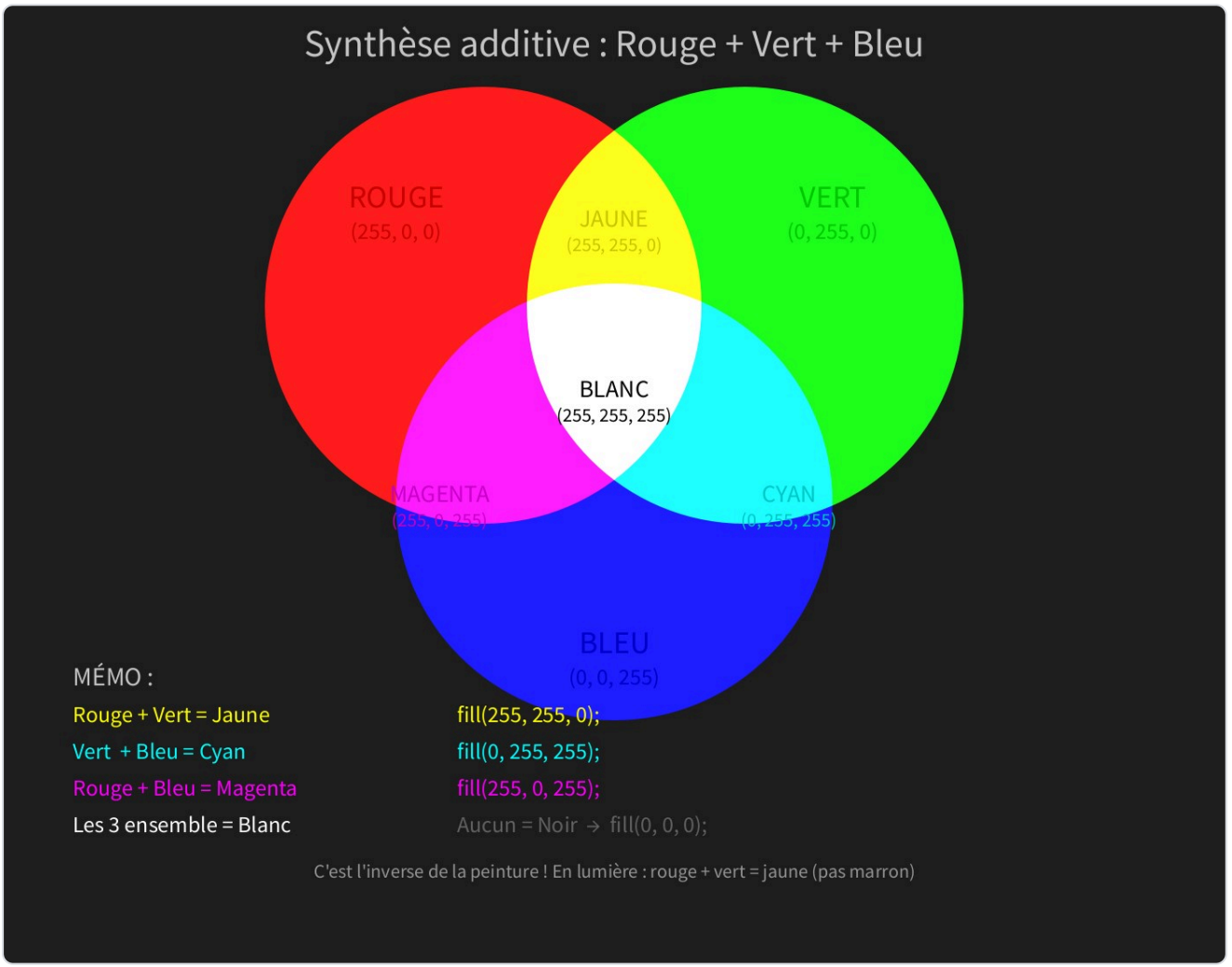
Les couleurs primaires

```javascript
fill(255, 0, 0);    // 🟥 Rouge pur — que du rouge, pas de vert, pas de bleu
fill(0, 255, 0);    // 🟩 Vert pur
fill(0, 0, 255);    // 🟦 Bleu pur
```

Les mélanges

```javascript
fill(255, 255, 0);   // 🟨 Rouge + Vert = Jaune
fill(0, 255, 255);   //      Vert + Bleu = Cyan
fill(255, 0, 255);   // 🟪 Rouge + Bleu = Magenta/Violet
fill(255, 255, 255); // ⬜ Les trois ensemble = Blanc
fill(0, 0, 0);       // ⬛ Aucun = Noir
```

⚠️ **Attention** : mélanger de la lumière, c'est l'inverse de la peinture ! En peinture, rouge + vert = marron. En lumière (RGB), rouge + vert = **jaune**. C'est surprenant mais c'est comme ça que fonctionnent tous les écrans.



Mémo des couleurs courantes

Voici les codes RGB des couleurs que tu utiliseras le plus souvent :

Couleur	Code RGB	Astuce pour retenir
⬛ Noir	`0, 0, 0`	Tout à zéro

Couleur	Code RGB	Astuce pour retenir
⬜ Blanc	`255, 255, 255`	Tout au max
🟥 Rouge	`255, 0, 0`	Premier à fond
🟩 Vert	`0, 255, 0`	Deuxième à fond
🟦 Bleu	`0, 0, 255`	Troisième à fond
🟨 Jaune	`255, 255, 0`	Rouge + Vert
🟧 Orange	`255, 165, 0`	Rouge + un peu de Vert
🟪 Violet	`128, 0, 128`	Rouge + Bleu (à moitié)
Cyan	`0, 255, 255`	Vert + Bleu
🦩 Rose	`255, 192, 203`	Beaucoup de tout, surtout Rouge
🤎 Brun	`139, 69, 19`	Rouge dominant, peu de Vert
🪨 Gris	`128, 128, 128`	Les trois égaux (entre noir et blanc)

💡 **Astuce** : pas besoin de retenir tout ça par cœur ! Tu peux utiliser n'importe quel **color picker** en ligne, ou même celui intégré aux DevTools de ton navigateur. Tape "color picker" dans Google et il t'en affiche un directement.

Le lien avec CSS

Tu connais déjà les couleurs en CSS ? Bonne nouvelle, c'est le **même système RGB** :

CSS	p5.js	Résultat
`color: rgb(255, 0, 0);`	`fill(255, 0, 0);`	🟥 Rouge
`background-color: rgb(0, 0, 0);`	`background(0, 0, 0);`	⬛ Noir
`border-color: rgb(0, 0, 255);`	`stroke(0, 0, 255);`	🟦 Contour bleu

Les valeurs sont exactement les mêmes — seule la syntaxe change !

Où utiliser les couleurs ?

Les couleurs s'appliquent à trois endroits :

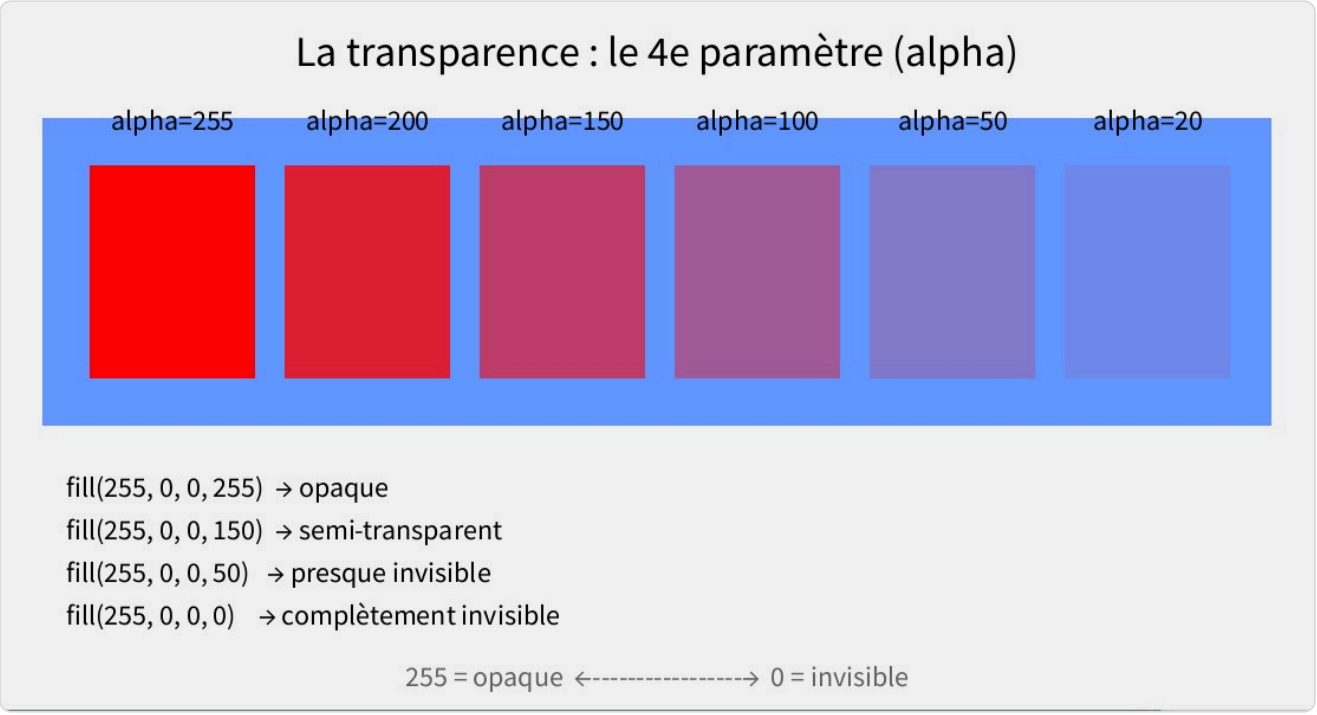
Fonction	Ce qu'elle colorie	Exemple
`background(r, g, b)`	Le **fond** du canevas	`background(135, 206, 235);` → ciel bleu

Fonction	Ce qu'elle colorie	Exemple
`fill(r, g, b)`	L'**intérieur** des formes	`fill(255, 0, 0);` → formes rouges
`stroke(r, g, b)`	Le **contour** des formes	`stroke(0, 0, 255);` → contour bleu

Chacune de ces fonctions accepte les mêmes formats :

```javascript
fill(128);            // 1 valeur  → niveau de gris
fill(255, 0, 0);      // 3 valeurs → couleur RGB
fill(255, 0, 0, 128); // 4 valeurs → couleur RGB + transparence
```

La transparence



On peut ajouter un **4e nombre** pour la transparence (appelée **alpha**), de 0 (invisible) à 255 (opaque) :

```javascript
fill(255, 0, 0, 255);  // Rouge complètement opaque (par défaut)
fill(255, 0, 0, 180);  // Rouge un peu transparent
fill(255, 0, 0, 100);  // Rouge assez transparent
fill(255, 0, 0, 30);   // Rouge presque invisible
```

C'est très utile pour superposer des formes et créer des effets de profondeur ou de verre :

```javascript
background(240);

// Cercle bleu
fill(0, 0, 255);
ellipse(180, 150, 150, 150);

// Cercle rouge semi-transparent par-dessus
fill(255, 0, 0, 120);
ellipse(250, 150, 150, 150);
```

💡 **Astuce** : la transparence fonctionne aussi avec `stroke()` :

```javascript
stroke(0, 0, 0, 100);  // Contour noir semi-transparent
```

Stocker une couleur dans une variable

p5.js possède une fonction `color()` pour créer des couleurs et les stocker dans des variables. C'est pratique quand tu utilises souvent la même couleur :

```javascript
let rouge = color(255, 0, 0);
let bleuCiel = color(135, 206, 235);

background(bleuCiel);
fill(rouge);
rect(50, 50, 100, 80);
```

On verra les variables en détail dans un prochain chapitre, mais sache que ça existe !

Mini-exemple : une scène colorée



Voici un exemple qui combine tout ce qu'on a vu dans les trois chapitres d'introduction — formes, couleurs, remplissage, contour et ordre de dessin :

```javascript
function setup() {
  createCanvas(500, 400);
}

function draw() {
  // Ciel
  background(135, 206, 235);

  // Soleil (jaune, sans contour)
  noStroke();
  fill(255, 220, 50);
  ellipse(420, 60, 80, 80);

  // Sol (vert)
  fill(80, 180, 80);
  rect(0, 300, 500, 100);

  // Maison : mur (marron avec contour)
  stroke(100, 60, 30);
  strokeWeight(2);
  fill(200, 120, 80);
  rect(150, 200, 200, 120);

  // Maison : toit (triangle rouge)
  fill(180, 40, 40);
  triangle(130, 200, 250, 120, 370, 200);

  // Maison : porte (marron foncé)
  fill(100, 60, 30);
  rect(220, 250, 50, 70);

  // Fenêtres (bleu clair)
  fill(180, 220, 255);
  rect(170, 230, 35, 35);
  rect(295, 230, 35, 35);
}
```

Remarque l'**ordre** : le ciel en premier (tout derrière), puis le sol, puis la maison, puis les détails par-dessus. Chaque élément recouvre les précédents.

Résumé

Concept	Syntaxe	Exemple
Niveau de gris	`fill(valeur)`	`fill(128)` → gris moyen
Couleur RGB	`fill(r, g, b)`	`fill(255, 0, 0)` → rouge
Avec transparence	`fill(r, g, b, a)`	`fill(255, 0, 0, 128)` → rouge transparent

Concept	Syntaxe	Exemple
Fond du canevas	`background(r, g, b)`	`background(135, 206, 235)` → ciel
Stocker une couleur	`let nom = color(r, g, b)`	`let c = color(255, 0, 0);`
Sélecteur de couleurs	Color picker Google / DevTools	Trouve n'importe quel code RGB

Valeur RGB	Règle
`0`	Rien de cette composante
`128`	Moitié
`255`	Maximum de cette composante
Les 3 égales	Niveau de gris (du noir au blanc)
Les 3 à 255	Blanc
Les 3 à 0	Noir

2.3 Premières Formes

> 🎯 ***Objectif*** *: Combiner formes, couleurs, remplissage et contour pour créer des compositions visuelles. À la fin de ce chapitre, tu seras capable de dessiner une scène complète en contrôlant précisément l'apparence de chaque élément.*

Rappel express

Dans les chapitres précédents, tu as découvert trois familles d'outils :

Famille	Ce que tu sais faire	Fonctions
📐 **Formes**	Dessiner des formes géométriques	`rect()`, `ellipse()`, `triangle()`, `line()`, `point()`, `arc()`
🎨 **Remplissage**	Colorier l'intérieur des formes	`fill()`, `noFill()`
✏️ **Contour**	Contrôler le trait autour des formes	`stroke()`, `noStroke()`, `strokeWeight()`

Ce chapitre, c'est la **mise en pratique**. On va combiner tout ça pour créer de vraies compositions, comme un artiste qui sort enfin ses pinceaux après avoir appris à les tenir.

Exercice 1 — Placement de formes

Consigne

Crée un canevas de **400×300** pixels avec un fond de la couleur de ton choix. Place dessus :

- Un **carré** (rappel : un carré, c'est un rectangle avec la même largeur et la même hauteur)
- Un **triangle**
- Une **ligne**

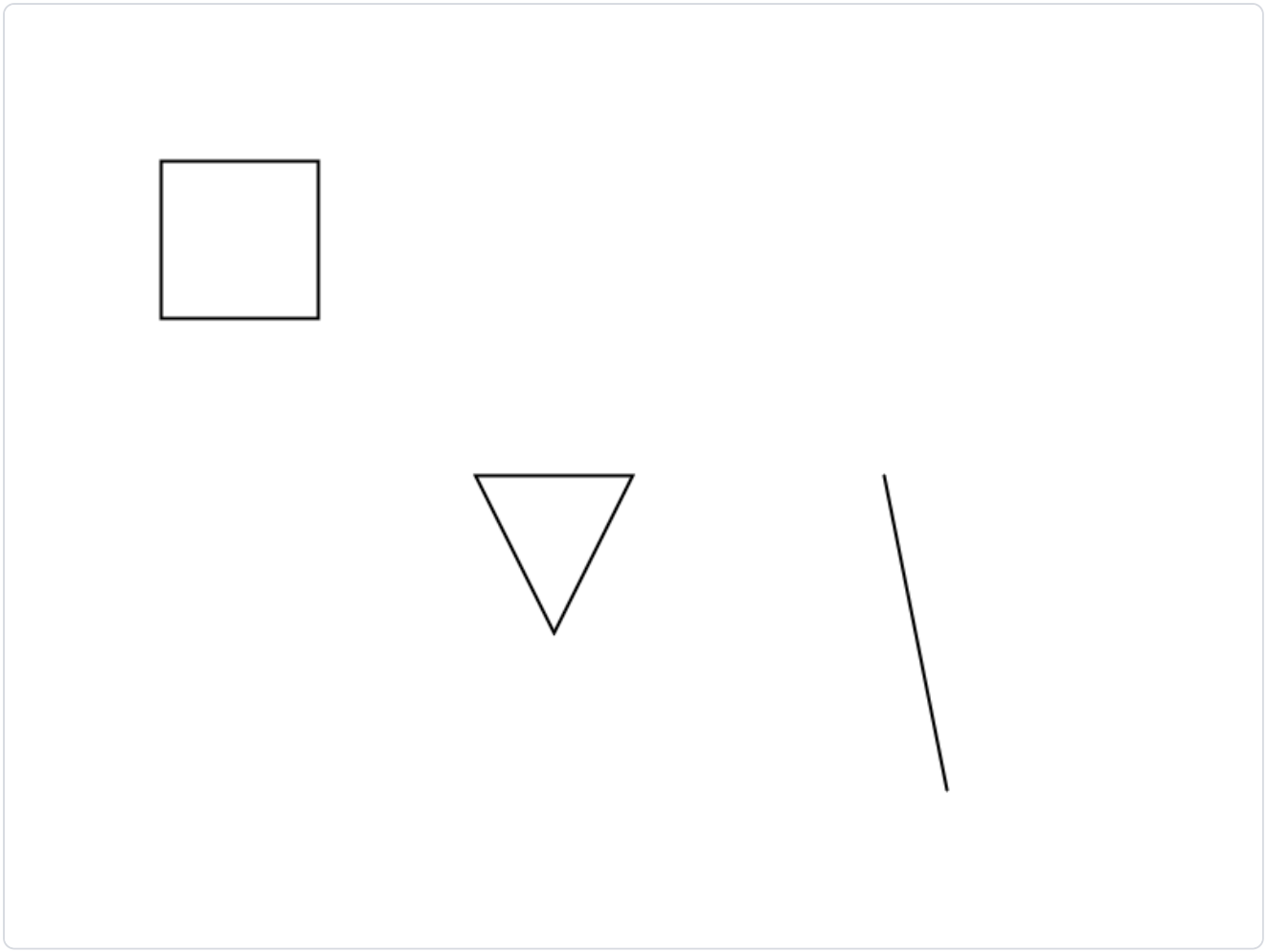
Les trois formes **ne doivent pas se chevaucher**.

Contraintes

- Utilise `createCanvas(400, 300)` dans le `setup()`
- Choisis un `background()` qui te plaît
- Laisse les couleurs par défaut (on les changera dans l'exercice 2)

Ce que tu devrais obtenir

Un canevas avec trois formes bien séparées. Par exemple :



Indices

💡 **Indice 1** — Un carré est un `rect()` dont la largeur et la hauteur sont **identiques** : `rect(50, 50, 80, 80)` fait un carré de 80×80 pixels.

💡 **Indice 2** — Ton canevas fait 400 pixels de large. Divise mentalement l'espace en trois zones : gauche (0 à ~130) pour le carré, centre (~130 à ~270) pour le triangle, droite (~270 à 400) pour la ligne.

💡 **Indice 3** — Structure du programme :

```javascript
function setup() {
  createCanvas(400, 300);
}

function draw() {
  background(???);

  // Carré
  rect(???, ???, ???, ???);

  // Triangle
  triangle(???, ???, ???, ???, ???, ???);

  // Ligne
  line(???, ???, ???, ???);
}
```

Exercice 2 — Changement de couleurs

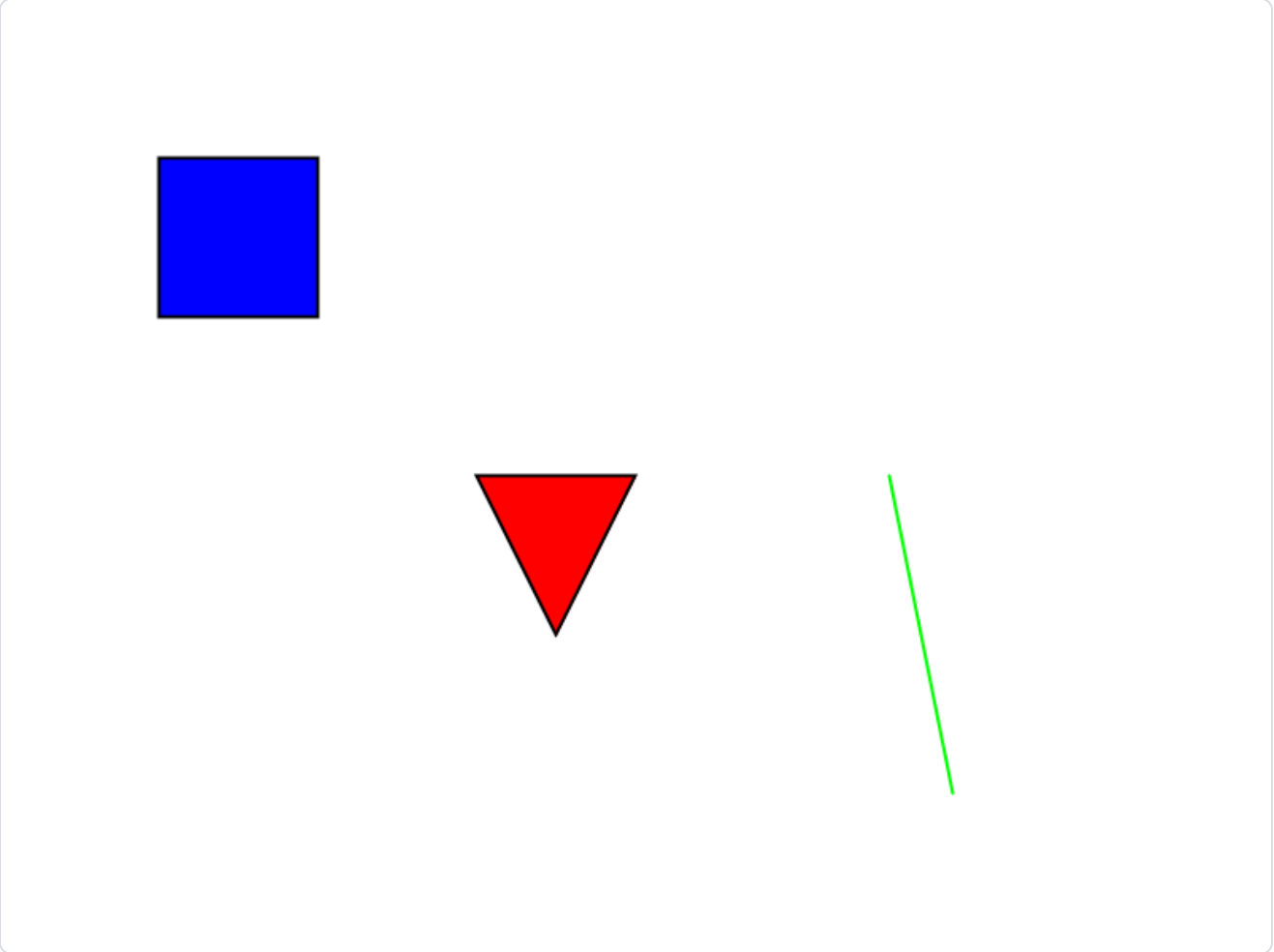
Consigne

Reprends ton code de l'exercice 1 et ajoute des couleurs :

- Le carré doit être rempli en **bleu** 🟦
- Le triangle doit être rempli en **rouge** 🟥
- La ligne doit être **verte** 🟩 et plus épaisse (au moins 3 pixels)

Contraintes

- Le carré doit avoir un **contour noir** visible
- Le triangle doit avoir un **contour noir** visible
- La ligne doit être **épaisse** (utilise `strokeWeight()`)



Indices

💡 **Indice 1** — Rappel des couleurs de base : `fill(0, 0, 255)` pour le bleu, `fill(255, 0, 0)` pour le rouge, `stroke(0, 255, 0)` pour un contour/ligne vert.

💡 **Indice 2** — Une ligne n'a **pas de remplissage** (`fill()` n'a aucun effet sur elle). Pour changer la couleur d'une ligne, c'est `stroke()` qu'il faut utiliser. Et pour l'épaissir : `strokeWeight()` .

💡 **Indice 3** — Si tu veux que le carré et le triangle aient un contour noir mais que la ligne soit verte, tu dois changer le `stroke()` entre les formes :

JAVASCRIPT

```
stroke(0);            // Contour noir pour le carré et le triangle
fill(0, 0, 255);
rect(...);

fill(255, 0, 0);
triangle(...);

stroke(0, 255, 0);    // On passe au vert pour la ligne
strokeWeight(4);
line(...);
```

🎯 Défi bonus — Le drapeau

Tu as terminé les exercices 1 et 2 ? Dessine le drapeau d'un pays de ton choix ! Par exemple :

- 🇧🇪 **Belgique** : trois rectangles verticaux (noir, jaune, rouge)
- 🇫🇷 **France** : trois rectangles verticaux (bleu, blanc, rouge)
- 🇯🇵 **Japon** : fond blanc + cercle rouge au centre
- 🇩🇪 **Allemagne** : trois rectangles horizontaux (noir, rouge, jaune)
- 🇮🇹 **Italie** : trois rectangles verticaux (vert, blanc, rouge)

💡 **Astuce** : pour un drapeau, utilise `noStroke()` pour éviter les traits entre les bandes.

💡 **Astuce** : pour un drapeau à 3 bandes verticales qui remplit tout le canevas, chaque bande fait un tiers de la largeur. Par exemple avec `createCanvas(300, 200)`, chaque bande fait 100 pixels.

Résumé

Ce chapitre t'a permis de mettre en pratique les trois piliers du dessin en p5.js :

Pilier	Ce qu'il faut retenir
📐 **Formes**	Chaque fonction de dessin a ses propres paramètres (position, taille, points...)
🎨 **Couleurs**	`fill()` pour l'intérieur, `stroke()` pour le contour — ils restent actifs jusqu'au prochain appel
📋 **Ordre**	Ce qui est dessiné en dernier apparaît par-dessus — comme des couches de peinture

Et les bonnes habitudes à garder :

- Définis le style (`fill`, `stroke`, `strokeWeight`) **juste avant** chaque forme
- Pense en **couches** : fond → éléments lointains → éléments proches

Dans le prochain chapitre, on va découvrir les **variables** — et ça va tout changer. Au lieu d'écrire des nombres fixes partout, on pourra utiliser des noms pour stocker et réutiliser des valeurs. C'est le premier pas vers des programmes vraiment dynamiques !

2.4 Les Variables

> 🎯 ***Objectif*** *: Comprendre ce qu'est une variable, savoir en créer, et les utiliser pour positionner et dimensionner des formes de manière intelligente. À la fin de ce chapitre, tu seras capable de dessiner des formes qui s'adaptent à la taille du canevas.*

C'est quoi une variable ?

Imagine une **boîte** avec :

- une **étiquette** (le nom de la variable)
- une **valeur** à l'intérieur (le contenu)

```
┌──────────┐
│   150    │  ← la valeur
└──────────┘
  let posX      ← le mot-clé + le nom (l'étiquette)
```

En JavaScript (et donc en p5.js), ça s'écrit :

```javascript
let posX = 150;
```

Cette ligne fait trois choses d'un coup :

1. Elle crée une boîte avec `let`
2. Elle colle l'étiquette `posX` dessus
3. Elle met la valeur `150` à l'intérieur

Tu connais déjà ça !

Tu as déjà utilisé des variables en JavaScript. En p5.js, c'est **exactement la même syntaxe** :

JavaScript classique	p5.js
`let posX;`	`let posX;`
`posX = 150;`	`posX = 150;`
`let posX = 150;`	`let posX = 150;`
`posX = posX + 10;`	`posX = posX + 10;`

Zéro surprise : c'est du JavaScript pur et simple.

`let` vs `const`

En JavaScript, tu as deux façons principales de déclarer une variable :

Mot-clé	Quand l'utiliser	Exemple
`let`	La valeur peut **changer** au fil du temps	`let posX = 150;`

Mot-clé	Quand l'utiliser	Exemple
`const`	La valeur ne changera **jamais**	`const TAILLE_MAX = 500;`

```javascript
let score = 0;          // ✅ score va évoluer
score = 10;             // ✅ OK, on peut modifier un let

const GRAVITE = 9.81;   // ✅ la gravité ne change pas
GRAVITE = 10;           // ❌ Erreur ! On ne peut pas modifier un const
```

💡 **Bonne pratique** : utilise `const` quand la valeur ne doit pas changer, `let` pour le reste. Ça rend ton code plus clair et évite des bugs.

⚠️ **Et `var` ?** Tu as peut-être vu `var` dans du code JavaScript. C'est l'ancienne façon de déclarer des variables. On ne l'utilise plus — préfère toujours `let` ou `const`.

Les types en JavaScript

Contrairement à des langages comme Java ou C++, JavaScript ne demande **pas** de préciser le type. Il le devine tout seul. Mais il est important de connaître les types qui existent :

Type	Ce que ça stocke	Exemple
`number`	Un nombre (entier **ou** à virgule)	`let age = 25;` / `let prix = 9.99;`
`string`	Du texte	`let nom = "Alice";`
`boolean`	Vrai ou faux	`let estVisible = true;`

```javascript
let largeur = 200;      // ✅ number (entier)
let vitesse = 3.5;      // ✅ number (à virgule)
let nom = "Alice";      // ✅ string
let actif = true;       // ✅ boolean
```

💡 **Avantage JavaScript** : pas besoin de choisir entre `int` et `float` comme dans d'autres langages. Un `number` gère les deux !

⚠️ **Attention** : en programmation, la virgule s'écrit avec un **point** ! On écrit `3.5` et pas `3,5`.

Modifier une variable

Une fois créée avec `let`, tu peux **changer** la valeur d'une variable autant de fois que tu veux. Par contre, tu ne remets **pas** le `let` :

```javascript
let score = 0;        // Création : on met let
score = 10;           // Modification : PAS de let
score = score + 5;    // score vaut maintenant 15
```

⚠️ **Piège classique** : écrire `let score = 10;` puis plus loin `let score = 20;`. JavaScript va te dire que la variable existe déjà ! Le `let` ne se met qu'**une seule fois**, à la création.

Les opérations mathématiques

Les variables numériques supportent les opérations de base :

Opération	Symbole	Exemple	Résultat
Addition	`+`	`10 + 3`	`13`
Soustraction	`-`	`10 - 3`	`7`
Multiplication	`*`	`10 * 3`	`30`
Division	`/`	`10 / 3`	`3.3333...`

💡 **Bonne nouvelle** : contrairement à d'autres langages (Java, C++...), JavaScript ne fait **pas** de division entière. `10 / 3` donne bien `3.3333...` et pas `3`. Un souci en moins !

Tu peux bien sûr combiner variables et opérations :

```javascript
let cote = 100;
let demiCote = cote / 2;    // 50
let doubleCote = cote * 2;  // 200
```

Les variables spéciales de p5.js

p5.js te fournit des variables déjà toutes faites, prêtes à l'emploi. Pas besoin de les créer, elles existent automatiquement :

`width` et `height` — les dimensions du canevas

Variable	Ce qu'elle contient
`width`	La **largeur** du canevas (en pixels)
`height`	La **hauteur** du canevas (en pixels)

Si tu as écrit `createCanvas(400, 300)`, alors `width` vaut `400` et `height` vaut `300`.

Pourquoi c'est utile ? Parce que ça te permet de **positionner des formes par rapport au canevas**, sans connaître sa taille à l'avance :

```javascript
function setup() {
  createCanvas(400, 300);
}

function draw() {
  background(240);

  // Centre du canevas, quelle que soit la taille
  ellipse(width / 2, height / 2, 50, 50);
}
```

Essaie de changer `createCanvas(400, 300)` en `createCanvas(600, 400)` — le cercle reste **toujours au centre** ! Sans `width` et `height`, il faudrait recalculer à la main à chaque fois.

`mouseX` et `mouseY` — la position de la souris

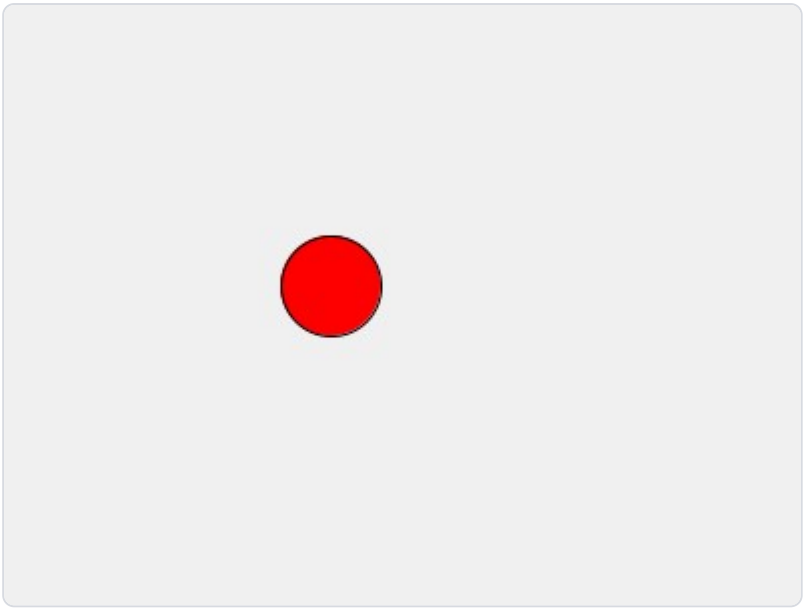
Variable	Ce qu'elle contient
`mouseX`	La position **horizontale** de la souris (en pixels depuis la gauche)
`mouseY`	La position **verticale** de la souris (en pixels depuis le haut)

Ces variables changent **60 fois par seconde** (à chaque passage dans `draw()`). C'est ce qui permet de créer de l'interactivité :

```javascript
function setup() {
  createCanvas(400, 300);
}

function draw() {
  background(240);

  // Un cercle qui suit la souris
  fill(255, 0, 0);
  ellipse(mouseX, mouseY, 50, 50);
}
```



Essaie ce code ! Tu devrais voir un cercle rouge qui suit ta souris partout sur le canevas.

💡 **Expérience** : que se passe-t-il si tu **enlèves** la ligne `background(240)` ? Essaie, le résultat est surprenant ! (On reviendra sur cette astuce dans le chapitre sur l'animation.)

Bien nommer ses variables

Le nom d'une variable, c'est comme le nom d'un fichier sur ton ordinateur : il doit être **clair et descriptif**. Quelques règles :

Règle	✅ Bon	❌ Mauvais
Nom descriptif	`posX`, `largeurCarre`	`a`, `x1`, `truc`
Commence par une minuscule	`tailleCercle`	`TailleCercle`
Pas d'espaces ni d'accents	`couleurFond`	`couleur fond`, `côté`
Plusieurs mots → camelCase	`vitesseMax`	`vitesse_max`

La convention **camelCase** (première lettre en minuscule, puis majuscule à chaque nouveau mot) est le standard en JavaScript. Tu la retrouveras partout : dans les frameworks, les API, la doc…

Exercice 3 — Carré centré

Consigne

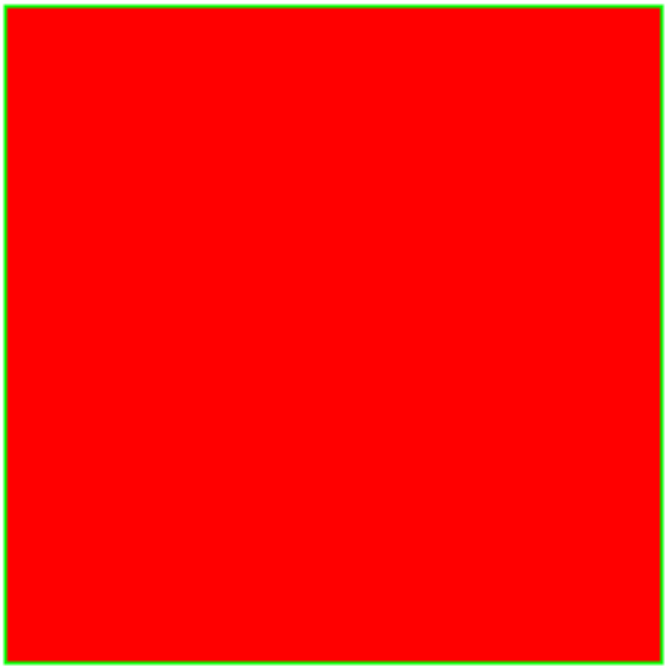
Dessine un **carré rouge** avec un **contour vert** parfaitement **au centre** de l'écran, en utilisant `width` et `height`.

Contraintes

- Le canevas fait `createCanvas(400, 300)`
- Le carré fait **100×100** pixels
- Le carré doit être **centré** sur le canevas
- Remplissage **rouge**, contour **vert**, épaisseur de contour de **3** pixels

Ce que tu devrais obtenir

Un carré rouge avec un contour vert, pile au milieu du canevas.



Indices

💡 **Indice 1** — Le centre du canevas est à la position `(width/2, height/2)`. Mais attention : par défaut, `rect()` dessine à partir du **coin supérieur gauche**, pas du centre !

💡 **Indice 2** — Tu as deux façons de centrer un rectangle :

- **Méthode calcul** : soustraire la moitié de la taille au point central → `rect(width/2 - 50, height/2 - 50, 100, 100)`

- **Méthode** `rectMode` : cherche `rectMode` dans la documentation de p5.js (https://p5js.org/reference). Il y a un mode qui change le comportement de `rect()` pour dessiner à partir du centre...

💡 **Indice 3** — `rectMode(CENTER)` fait en sorte que le premier paramètre de `rect()` soit le **centre** du rectangle, pas le coin. C'est beaucoup plus pratique pour centrer !

Exercice 4 — Dessiner dans le carré

Consigne

Reprends ton carré centré de l'exercice 3, et trace des **lignes à l'intérieur** en utilisant des variables pour tous les calculs de position.

Étape par étape :

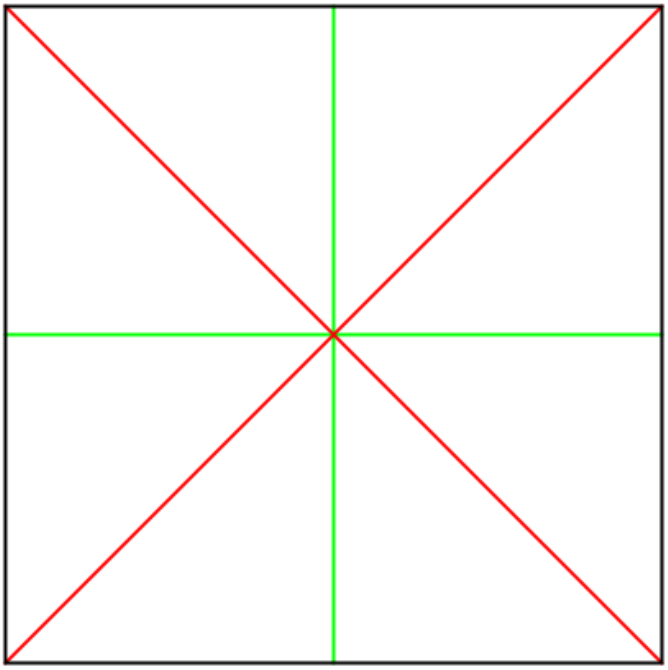
1. Crée des variables `x`, `y` pour la position du coin supérieur gauche du carré, `cote` pour la taille du côté, et `demiCote` pour la moitié
2. Dessine le carré en utilisant ces variables
3. Trace la **médiane verticale** (ligne du milieu haut au milieu bas du carré)
4. Trace la **médiane horizontale** (ligne du milieu gauche au milieu droit)
5. Trace les deux **diagonales** (coin à coin)

Contraintes

- Utilise **uniquement des variables** pour les positions (pas de nombres "en dur" sauf pour la taille initiale)
- Chaque ligne a une **couleur différente**
- Joue avec la **transparence** pour voir les lignes qui se croisent

Ce que tu devrais obtenir

Un carré avec 4 lignes colorées à l'intérieur : les deux médianes (en croix +) et les deux diagonales (en croix ×).



Indices

💡 **Indice 1** — Commence par créer tes variables. Si tu utilises `rectMode(CENTER)`, le coin supérieur gauche du carré est à `(width/2 - cote/2, height/2 - cote/2)`. Si tu préfères, tu peux aussi utiliser `rectMode(CORNER)` (le mode par défaut) et calculer `x` et `y` directement.

💡 **Indice 2** — La médiane verticale va du point `(x + demiCote, y)` au point `(x + demiCote, y + cote)`. La médiane horizontale va de `(x, y + demiCote)` à `(x + cote, y + demiCote)`.

💡 **Indice 3** — Les diagonales relient les coins opposés. La première va de `(x, y)` à `(x + cote, y + cote)`. La deuxième va de `(x + cote, y)` à `(x, y + cote)`.

🎯 Défi bonus — Le suiveur de souris

Reprends l'exercice 4, mais remplace les variables `x` et `y` par `mouseX` et `mouseY`. Ton carré (avec ses lignes intérieures) suivra la souris !

💡 **Astuce** : n'oublie pas le `background()` au début de `draw()`, sinon tu verras toutes les positions précédentes du carré.

Résumé

Concept	Syntaxe	Exemple
Créer une variable	`let nom = valeur;`	`let posX = 200;`
Créer une constante	`const nom = valeur;`	`const GRAVITE = 9.81;`
Modifier une variable	`nom = nouvelleValeur;`	`posX = posX + 10;`
Largeur du canevas	`width`	`ellipse(width/2, ...)`
Hauteur du canevas	`height`	`ellipse(..., height/2, ...)`
Position souris X	`mouseX`	`ellipse(mouseX, ...)`
Position souris Y	`mouseY`	`ellipse(..., mouseY, ...)`

Les points clés à retenir :

- Une variable se crée avec `let`, une constante avec `const`
- Le `let` / `const` ne se met qu'**une seule fois** (à la création)
- JavaScript devine le type automatiquement (`number`, `string`, `boolean`)
- `width`, `height`, `mouseX`, `mouseY` sont des variables **fournies par p5.js**
- Utiliser des variables rend ton code **flexible** : change une valeur, tout s'adapte

Dans le prochain chapitre, on va découvrir les **conditions** — le moyen de faire réagir ton programme différemment selon la situation. C'est le `if` qui ouvre la porte à l'interactivité !

2.5 Les Conditions

> 🎯 ***Objectif*** *: Faire réagir ton programme différemment selon la situation grâce à* `if`*,* `else if` *et* `else`*. À la fin de ce chapitre, tu seras capable de créer un dessin interactif qui change en fonction de la position de la souris et des clics.*

C'est quoi une condition ?

Dans la vie de tous les jours, tu prends des décisions en permanence :

- **SI** il pleut **ALORS** je prends un parapluie, **SINON** je mets mes lunettes de soleil
- **SI** le feu est rouge **ALORS** je m'arrête
- **SI** j'ai faim **ALORS** je mange

En programmation, c'est exactement pareil. Une **condition** permet à ton programme de faire des choix : exécuter certaines instructions **uniquement si** quelque chose est vrai.

Tu connais déjà ça !

Les conditions en p5.js sont **exactement les mêmes** qu'en JavaScript classique. Pas de nouvelle syntaxe à apprendre :

JavaScript classique	p5.js
`if (condition) { }`	`if (condition) { }`
`if (condition) { } else { }`	`if (condition) { } else { }`
`if ... else if ... else`	`if ... else if ... else`

La seule nouveauté, ce sont les **variables spéciales** de p5.js (`mouseX`, `mouseIsPressed`...) qu'on va utiliser dans nos conditions pour créer de l'interactivité.

La syntaxe du `if`

`if` seul — "si ... alors"

```javascript
if (condition) {
  // Ce code s'exécute UNIQUEMENT si la condition est vraie
}
```

Exemple concret :

```javascript
let temperature = 35;

if (temperature > 30) {
  fill(255, 0, 0);  // Rouge si c'est chaud
}
rect(50, 50, 100, 100);
```

Si `temperature` est supérieure à 30, le rectangle sera rouge. Sinon, il gardera la couleur par défaut (blanc).

`if ... else` — "si ... alors ... sinon"

```javascript
if (condition) {
  // Exécuté si la condition est VRAIE
} else {
  // Exécuté si la condition est FAUSSE
}
```

```javascript
let temperature = 15;

if (temperature > 30) {
  fill(255, 0, 0);  // Rouge si chaud
} else {
  fill(0, 0, 255);  // Bleu sinon
}
rect(50, 50, 100, 100);
```

C'est **l'un ou l'autre**, jamais les deux. Soit la condition est vraie et on passe dans le premier bloc, soit elle est fausse et on passe dans le `else`.

`if ... else if ... else` — plusieurs cas

Quand tu as plus de deux possibilités :

```javascript
let temperature = 22;

if (temperature > 30) {
  fill(255, 0, 0);    // Rouge : chaud
} else if (temperature > 20) {
  fill(255, 165, 0);  // Orange : tiède
} else if (temperature > 10) {
  fill(0, 255, 0);    // Vert : frais
} else {
  fill(0, 0, 255);    // Bleu : froid
}
rect(50, 50, 100, 100);
```

Les conditions sont testées **de haut en bas** et le programme s'arrête à la **première** qui est vraie. Si aucune n'est vraie, il passe dans le `else` (s'il y en a un).

⚠️ **Piège classique** : l'ordre des `else if` compte ! Si tu mets `temperature > 10` avant `temperature > 30`, la condition `> 10` sera vraie dès 11°C et le programme n'ira jamais tester `> 30`. Commence toujours par la condition la plus **restrictive**.

Les opérateurs de comparaison

Pour écrire des conditions, tu as besoin de **comparer** des valeurs :

Opérateur	Signification	Exemple	Vrai si...
`==`	Est égal à	`x == 10`	x vaut 10
`!=`	Est différent de	`x != 10`	x ne vaut pas 10

Opérateur	Signification	Exemple	Vrai si...
`<`	Est plus petit que	`x < 10`	x est strictement inférieur à 10
`>`	Est plus grand que	`x > 10`	x est strictement supérieur à 10
`⩽`	Est plus petit ou égal	`x ⩽ 10`	x vaut 10 ou moins
`⩾`	Est plus grand ou égal	`x ⩾ 10`	x vaut 10 ou plus

⚠️ **ATTENTION** : pour comparer, c'est **trois** signes égal `===` en JavaScript. Un seul `=`, c'est pour **assigner** une valeur. C'est l'erreur n°1 des débutants !

JAVASCRIPT

```
let x = 10;       // ✅ Assigner : "x reçoit la valeur 10"
if (x === 10) {   // ✅ Comparer : "est-ce que x vaut 10 ?"
if (x = 10) {     // ❌ ERREUR ! Ça assigne au lieu de comparer
```

💡 `===` vs `==` en JavaScript : tu verras parfois `==` (deux signes) dans du code. La différence : `==` convertit les types avant de comparer (`"5" == 5` est vrai), tandis que `===` compare aussi le type (`"5" === 5` est faux). Préfère toujours `===`, c'est plus fiable et prévisible.

Combiner des conditions

Tu peux combiner plusieurs conditions avec les opérateurs logiques :

Opérateur	Signification	Exemple
`&&`	**ET** (les deux doivent être vraies)	`x > 0 && x < 100`
`||`	**OU** (au moins une doit être vraie)	`x < 0 || x > 100`
`!`	**NON** (inverse la condition)	`!mouseIsPressed`

JAVASCRIPT

```
// Le clic est dans le quart supérieur gauche ?
if (mouseX < width / 2 && mouseY < height / 2) {
  fill(255, 0, 0);  // Rouge
}
```

C'est comme en français : "SI la souris est à gauche **ET** en haut **ALORS** c'est rouge."

Détecter la souris

p5.js fournit des variables et fonctions spéciales pour l'interactivité :

Variable/Fonction	Ce qu'elle fait
`mouseX`	Position horizontale de la souris
`mouseY`	Position verticale de la souris
`mouseIsPressed`	`true` si un bouton de la souris est enfoncé, `false` sinon
`mouseButton`	Un objet avec trois cases : `mouseButton.left`, `mouseButton.right` et `mouseButton.center`, chacune `true` ou `false`

⚠️ **Différence importante** : en p5.js, c'est `mouseIsPressed` (avec "Is" au milieu). C'est une des rares différences de nom par rapport à Processing qui utilise `mousePressed`.

Exemple : changer de couleur au clic

JAVASCRIPT

```javascript
function setup() {
  createCanvas(400, 300);
}

function draw() {
  background(240);

  if (mouseIsPressed) {
    fill(255, 0, 0);    // Rouge quand on clique
  } else {
    fill(200);           // Gris sinon
  }

  ellipse(width / 2, height / 2, 150, 150);
}
```

Exemple : détecter gauche vs droite

```javascript
function setup() {
  createCanvas(400, 300);
}

function draw() {
  if (mouseX < width / 2) {
    background(100, 150, 255);  // Bleu à gauche
  } else {
    background(255, 150, 100);  // Orange à droite
  }

  // Ligne de séparation
  stroke(255);
  strokeWeight(2);
  line(width / 2, 0, width / 2, height);
}
```

Essaie ce code : le fond change de couleur quand ta souris passe d'un côté à l'autre !

Quel bouton de souris ?

`mouseIsPressed` détecte **tous** les boutons de la souris (gauche, droit, molette). Pour savoir **quel** bouton est enfoncé, on utilise `mouseButton`.

`mouseButton` n'est pas un simple nombre ou un simple texte : c'est un **objet**. Imagine une boîte avec **trois cases**, une par bouton. Chaque case contient `true` (le bouton est enfoncé) ou `false` (il ne l'est pas). Pour lire une case, on écrit le nom de la boîte, un **point**, puis le nom de la case :

Expression	Vaut `true` quand...
`mouseButton.left`	Le bouton **gauche** est enfoncé
`mouseButton.right`	Le bouton **droit** est enfoncé
`mouseButton.center`	La **molette** (clic du milieu) est enfoncée

Comme chaque case est déjà un booléen (`true` ou `false`), tu peux la mettre **directement** dans un `if`, sans `===` :

```javascript
if (mouseIsPressed && mouseButton.left) {
  // Le bouton GAUCHE est enfoncé
}
```

💡 **Pourquoi combiner avec** `mouseIsPressed` **?** En pratique, `mouseButton.left` suffit : il repasse à `false` dès qu'on relâche. Mais garder `mouseIsPressed` rend l'intention plus lisible : "SI on clique ET que c'est le bouton gauche".

💡 **Bonus** : comme les trois cases sont indépendantes, tu peux détecter **deux boutons enfoncés en même temps** (`mouseButton.left && mouseButton.right`). Pratique pour un mini-Paint avec des outils différents selon le bouton !

> **ATTENTION**
>
> **Ancienne syntaxe (p5.js v1)** : dans les vieux tutoriels, les vidéos YouTube et le code généré par certaines IA, tu verras `if (mouseButton === LEFT)`. C'était la syntaxe de p5.js **version 1**. Depuis la **version 2** (celle de l'éditeur en ligne aujourd'hui), `mouseButton` est un objet, donc cette comparaison est **toujours fausse** : ton programme ne dessine rien, et **aucune erreur** ne s'affiche dans la console. Si un exemple trouvé sur Internet "ne fait rien" au clic, c'est probablement ça. Remplace-le par `mouseButton.left`.

Exemple : dessiner au clic gauche

JAVASCRIPT

```
function setup() {
  createCanvas(400, 300);
  background(240);
}

function draw() {
  if (mouseIsPressed && mouseButton.left) {
    noStroke();
    fill(255, 0, 0);
    ellipse(mouseX, mouseY, 20, 20);
  }
}
```

Essaie ce code ! Tant que tu maintiens le clic gauche, tu dessines des cercles rouges comme un pinceau. `draw()` s'exécute 60 fois par seconde, donc le tracé est **fluide**.

💡 **Pourquoi pas de** `background()` dans `draw()` ? — Parce qu'on veut que les cercles **restent affichés**. Si on repeint le fond à chaque frame, les cercles précédents disparaissent. En mettant `background()` uniquement dans `setup()`, les dessins s'accumulent.

Exercice 5 — Dessin interactif par zones

Consigne

Le canevas est divisé en **4 zones** (quadrants). Quand on **maintient le clic gauche**, des cercles apparaissent à la position de la souris, mais leur **couleur dépend de la zone** :

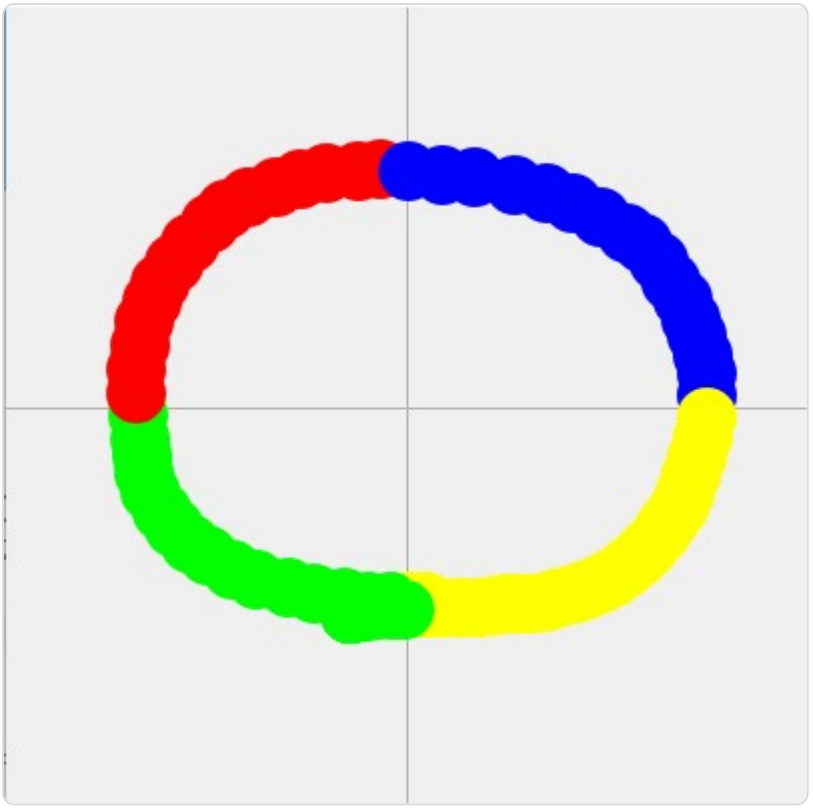
Zone	Position	Couleur
Haut-gauche	`mouseX < width/2` et `mouseY < height/2`	🟥 Rouge
Haut-droite	`mouseX ⩾ width/2` et `mouseY < height/2`	🟦 Bleu
Bas-gauche	`mouseX < width/2` et `mouseY ⩾ height/2`	🟩 Vert
Bas-droite	`mouseX ⩾ width/2` et `mouseY ⩾ height/2`	🟨 Jaune

Contraintes

- Canevas de `createCanvas(400, 400)`
- Le fond est dessiné **une seule fois** dans `setup()` (pas dans `draw()`) → les cercles restent visibles
- On dessine en continu avec `mouseIsPressed && mouseButton.left` dans `draw()`
- Chaque cercle fait **30 pixels** de diamètre
- Dessine les lignes de séparation des 4 zones (croix au centre de l'écran)

Ce que tu devrais obtenir

Un canevas divisé en 4 zones visibles. En maintenant le clic gauche et en déplaçant la souris, des cercles colorés apparaissent comme un pinceau, avec la couleur correspondant à la zone.



Indices

💡 **Indice 1** — Pour que les cercles restent affichés et ne disparaissent pas, **ne mets pas** `background()` dans `draw()`. Mets-le dans `setup()` à la place. Comme `draw()` ne repeint pas le fond, les cercles s'accumulent.

💡 **Indice 2** — Pour détecter le quadrant, combine deux conditions avec `&&` :

```javascript
if (mouseX < width / 2 && mouseY < height / 2) {
  // Haut-gauche
}
```

💡 **Indice 3** — Structure globale du programme :

```javascript
function setup() {
  createCanvas(400, 400);
  background(240);

  // Lignes de séparation
  stroke(180);
  line(width / 2, 0, width / 2, height);
  line(0, height / 2, width, height / 2);
}

function draw() {
  if (mouseIsPressed && mouseButton.left) {
    noStroke();

    // Détecte la zone et choisis la couleur
    if (mouseX < width / 2 && mouseY < height / 2) {
      fill(255, 0, 0);  // Rouge
    } else if (???) {
      // ...à toi de compléter !
    }

    ellipse(mouseX, mouseY, 30, 30);
  }
}
```

🎯 Défi bonus — La zone secrète

Ajoute une **5e zone circulaire** au centre de l'écran (un cercle de 100 pixels de diamètre). Si on dessine dans cette zone, les cercles sont **blancs**.

💡 **Astuce** : pour savoir si un point est dans un cercle, il faut calculer la **distance** entre le point (le clic) et le centre du cercle. p5.js a une fonction `dist()` toute prête :

```javascript
let d = dist(mouseX, mouseY, width / 2, height / 2);
if (d < 50) {
  // Le clic est dans la zone centrale (rayon = 50 pixels)
}
```

Pense à tester cette condition **en premier** (avant les quadrants), sinon les dessins au centre seront capturés par les quadrants avant d'arriver à la zone centrale !

🎯 Défi bonus 2 — La gomme

Reprends l'exercice 5 : le clic **droit** efface au lieu de dessiner (des cercles de la couleur du fond). Tu as maintenant deux outils, un par bouton.

💡 **Astuce** : c'est `mouseButton.right`. Et pour empêcher le navigateur d'ouvrir son menu contextuel au clic droit, ajoute cette ligne dans `setup()` :

```javascript
document.oncontextmenu = () => false;
```

Résumé

Concept	Syntaxe	Exemple
Si	`if (condition) { }`	`if (x > 10) { ... }`
Si ... sinon	`if (condition) { } else { }`	`if (x > 10) { ... } else { ... }`
Si ... sinon si ... sinon	`if ... else if ... else`	Tester plusieurs cas
Égal	`==`	`if (x == 10)`
Différent	`!=`	`if (x != 10)`
Plus petit / plus grand	`<`, `>`, `<=`, `>=`	`if (x < 100)`
ET logique	`&&`	`if (x > 0 && x < 100)`
OU logique	`||`	`if (x < 0 || x > 100)`
Souris enfoncée	`mouseIsPressed`	`if (mouseIsPressed) { }`
Bouton gauche	`mouseButton.left`	`if (mouseButton.left) { }`
Bouton droit	`mouseButton.right`	`if (mouseButton.right) { }`
Molette	`mouseButton.center`	`if (mouseButton.center) { }`
Distance	`dist(x1, y1, x2, y2)`	`dist(mouseX, mouseY, 200, 200)`

JavaScript classique	p5.js
`if (condition) { }`	`if (condition) { }` (identique !)
`element.addEventListener('click', ...)`	`mouseIsPressed` / `mouseButton`
`event.clientX` / `event.clientY`	`mouseX` / `mouseY`
`event.button == 0`	`mouseButton.left`

Les points clés à retenir :

- `==` pour comparer, `=` pour assigner — ne les confonds pas !
- Teste les conditions de la plus **restrictive** à la plus **générale**
- `&&` = les deux conditions doivent être vraies, `||` = au moins une suffit

- `mouseIsPressed` détecte le clic, `mouseButton.left` / `.right` / `.center` précise **quel** bouton (chacun vaut `true` ou `false`)
- `mouseButton === LEFT`, c'est de l'ancien p5.js (v1) : ça ne marche plus, sans message d'erreur

Dans le prochain chapitre, on va découvrir les **boucles** — le moyen de répéter des instructions sans copier-coller, pour dessiner des motifs, des grilles et des dégradés en quelques lignes !

2.6 Les Boucles

> 🎯 ***Objectif*** *: Comprendre pourquoi et comment répéter des instructions automatiquement avec la boucle `for`. À la fin de ce chapitre, tu seras capable de dessiner des motifs répétitifs, des cercles concentriques et même un damier, en quelques lignes de code.*

Pourquoi des boucles ?

Imagine qu'on te demande de dessiner **10 cercles** les uns à côté des autres. Sans boucle, tu écrirais :

```javascript
ellipse(30, 150, 40, 40);
ellipse(80, 150, 40, 40);
ellipse(130, 150, 40, 40);
ellipse(180, 150, 40, 40);
ellipse(230, 150, 40, 40);
ellipse(280, 150, 40, 40);
ellipse(330, 150, 40, 40);
ellipse(380, 150, 40, 40);
ellipse(430, 150, 40, 40);
ellipse(480, 150, 40, 40);
```

Dix lignes quasi identiques 😩 Et si on t'en demande 100 ? 1000 ? Tu vas copier-coller pendant des heures.

En programmation, **copier-coller, c'est un signal d'alarme**. Chaque fois que tu te retrouves à écrire plusieurs fois la même chose en changeant juste un nombre, c'est qu'il te faut une **boucle**.

Tu connais déjà ça !

La boucle `for` en p5.js est **exactement la même** qu'en JavaScript classique. La seule différence avec d'autres langages : on déclare le compteur avec `let` au lieu de `int` :

Autre langage (Java, C++...)	JavaScript / p5.js
`for (int i = 0; i < 10; i++)`	`for (let i = 0; i < 10; i++)`

Le résultat est le même : le code à l'intérieur s'exécute 10 fois. Et tu as un **compteur** (`i`) qui te dit à quelle répétition tu en es.

Anatomie d'une boucle `for`

```javascript
for (let i = 0; i < 10; i++) {
  // Ce code s'exécute 10 fois
  // i vaut 0, puis 1, puis 2, ... puis 9
}
```

Il y a trois parties séparées par des **points-virgules** :

```bash
for ( initialisation ; condition ; incrémentation )
       ↓                 ↓            ↓
   let i = 0         i < 10         i++
   "On commence      "Tant que      "À chaque tour,
    à 0"              i < 10,        on ajoute 1
                      on continue"   à i"
```

Partie	Ce qu'elle fait	Dans notre exemple
Initialisation	Crée le compteur et lui donne une valeur de départ	`let i = 0` → on commence à 0
Condition	Vérifie si on continue la boucle	`i < 10` → tant que i est plus petit que 10
Incrémentation	Modifie le compteur à chaque tour	`i++` → on ajoute 1 à i

💡 `i++` est un raccourci pour `i = i + 1`. C'est la même chose, juste plus court.

Ce qui se passe à chaque tour

Voici le déroulé complet d'un `for (let i = 0; i < 5; i++)` :

Tour	Valeur de `i`	Condition `i < 5` ?	Action
1	0	✅ Oui	Exécute le code
2	1	✅ Oui	Exécute le code
3	2	✅ Oui	Exécute le code
4	3	✅ Oui	Exécute le code
5	4	✅ Oui	Exécute le code

Tour	Valeur de `i`	Condition `i < 5` ?	Action
6	5	❌ Non	STOP, on sort de la boucle

La boucle s'exécute **5 fois** (pour i = 0, 1, 2, 3, 4). Quand `i` atteint 5, la condition `i < 5` est fausse → on s'arrête.

⚠️ **Piège classique** : la boucle commence à **0**, pas à 1. Et elle s'arrête **avant** d'atteindre la limite. `i < 5` donne les valeurs 0, 1, 2, 3, 4 → c'est bien **5 tours**, mais le dernier tour est à `i = 4`, pas `i = 5`.

La puissance du compteur `i`

Le compteur `i` n'est pas juste un numéro de tour : tu peux l'**utiliser dans tes calculs**. C'est ça qui rend la boucle `for` si puissante.

Reprenons nos 10 cercles du début :

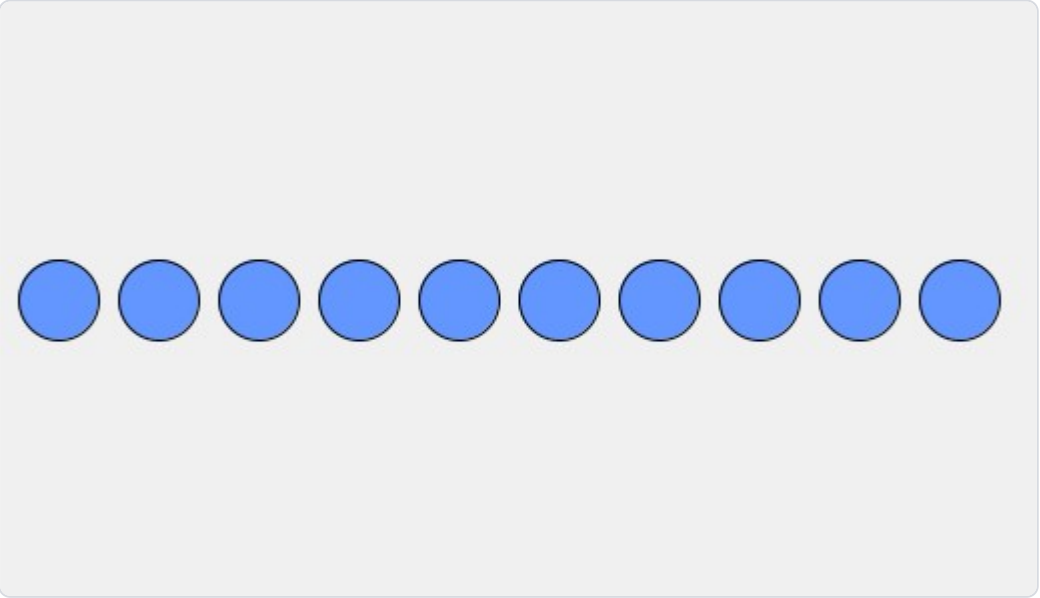
JAVASCRIPT

```javascript
function setup() {
  createCanvas(520, 300);
}

function draw() {
  background(240);

  fill(100, 150, 255);
  stroke(0);

  for (let i = 0; i < 10; i++) {
    ellipse(30 + i * 50, 150, 40, 40);
  }
}
```



Décortiquons `30 + i * 50` :

Tour	`i`	`30 + i * 50`	Position X
1	0	30 + 0 × 50	**30**
2	1	30 + 1 × 50	**80**
3	2	30 + 2 × 50	**130**
…	…	…	…
10	9	30 + 9 × 50	**480**

Deux lignes de code au lieu de dix. Et si tu veux 100 cercles ? Change juste `i < 10` en `i < 100` 😎

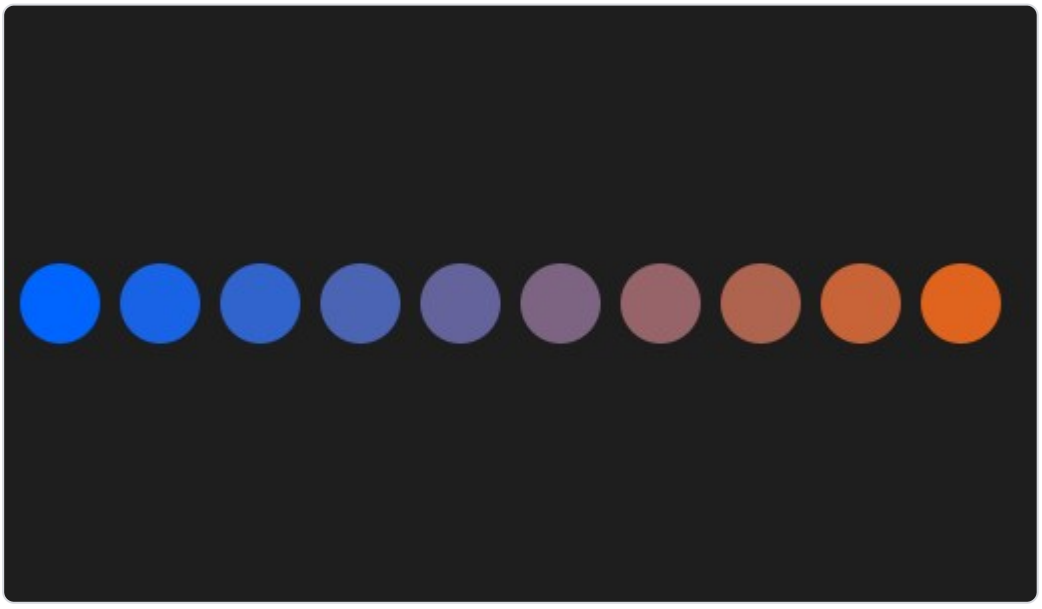
Utiliser `i` pour varier les couleurs

Le compteur `i` peut aussi servir à calculer des couleurs qui changent à chaque tour :

```javascript
function setup() {
  createCanvas(520, 300);
}

function draw() {
  background(30);
  noStroke();

  for (let i = 0; i < 10; i++) {
    fill(i * 25, 100, 255 - i * 25);  // La couleur varie !
    ellipse(30 + i * 50, 150, 40, 40);
  }
}
```



Ici, `i * 25` fait varier la composante rouge de 0 à 225, et `255 - i * 25` fait varier le bleu en sens inverse. Résultat : un joli dégradé du bleu au rouge !

Utiliser une variable pour le nombre de répétitions

Plutôt que d'écrire un nombre "en dur" dans la boucle, utilise une **variable** :

JAVASCRIPT

```
let nbCercles = 10;

for (let i = 0; i < nbCercles; i++) {
  ellipse(30 + i * 50, 150, 40, 40);
}
```

L'avantage ? Tu changes **une seule valeur** et tout s'adapte. C'est exactement le même principe que les variables du chapitre précédent : rendre ton code **flexible**.

Les boucles imbriquées (grilles 2D)

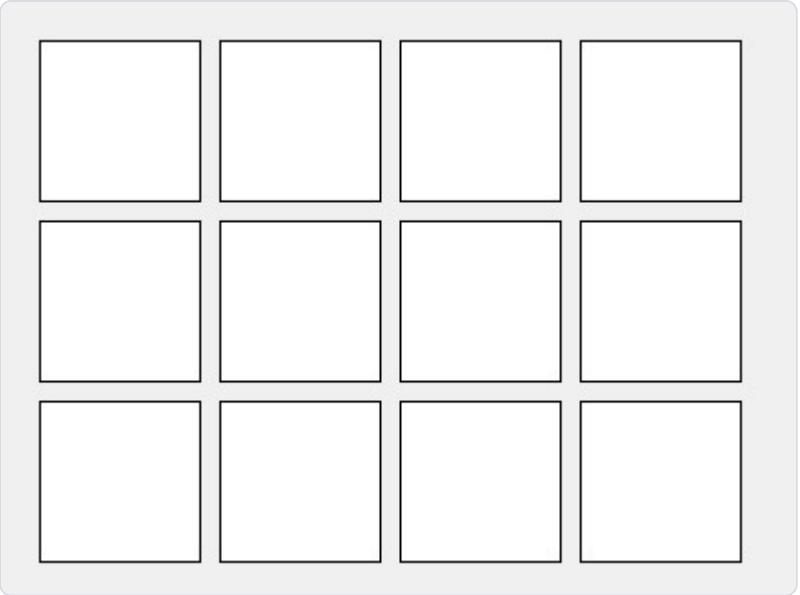
Une boucle `for`, c'est une répétition en **une dimension** (une rangée). Deux boucles `for` imbriquées, c'est une répétition en **deux dimensions** (une grille).

JAVASCRIPT

```
function setup() {
  createCanvas(400, 300);
}

function draw() {
  background(240);

  for (let ligne = 0; ligne < 3; ligne++) {
    for (let colonne = 0; colonne < 4; colonne++) {
      rect(20 + colonne * 90, 20 + ligne * 90, 80, 80);
    }
  }
}
```



Comment ça marche ? La boucle extérieure (`ligne`) parcourt les lignes, et pour **chaque ligne**, la boucle intérieure (`colonne`) dessine tous les éléments de cette ligne :

```
ligne=0 : colonne=0, colonne=1, colonne=2, colonne=3   → 1ère rangée
ligne=1 : colonne=0, colonne=1, colonne=2, colonne=3   → 2ème rangée
ligne=2 : colonne=0, colonne=1, colonne=2, colonne=3   → 3ème rangée
```

C'est comme lire un livre : on va de gauche à droite (colonnes), puis on passe à la ligne suivante.

💡 **Astuce** : nomme tes compteurs `ligne` et `colonne` (ou `row` et `col`) plutôt que `i` et `j` . C'est plus clair quand on relit le code.

Exercice 6 — L'arc-en-ciel (sans boucle)

Consigne

Dessine un **arc-en-ciel** : des demi-cercles concentriques de couleurs différentes. Le centre de l'arc-en-ciel est en **bas du canevas**. Ajoute un petit paysage autour (sol vert, ciel bleu, soleil...).

Les couleurs de l'arc-en-ciel, du plus grand au plus petit : rouge, orange, jaune, vert, bleu, indigo, violet.

Contraintes

- Canevas de `createCanvas(500, 400)`
- Utilise `arc()` avec les angles `0` et `PI` (demi-cercle supérieur) — ou bien des `ellipse()` que tu empiles
- **Pas de boucle** pour cet exercice (on fera la version boucle dans l'exercice suivant)

- `noStroke()` pour éviter les contours entre les arcs

Ce que tu devrais obtenir

Un arc-en-ciel avec 7 bandes de couleurs au-dessus d'un paysage simple.



Indices

💡 **Indice 1** — L'astuce est de dessiner des **ellipses** (ou arcs) du plus grand au plus petit, comme des poupées russes. La plus grande ellipse (rouge) est dessinée en premier, puis l'orange par-dessus (un peu plus petite), etc. La dernière ellipse (violette) est la plus petite.

💡 **Indice 2** — Centre tes demi-cercles en bas du canevas, par exemple à la position `(250, 400)`. Commence avec un diamètre de 400, puis diminue de 50 à chaque bande.

💡 **Indice 3** — Tu peux utiliser `arc(250, 400, 400, 400, PI, TWO_PI)` pour un demi-cercle vers le haut. Ou, plus simple : dessine des `ellipse()` complètes et cache le bas avec le rectangle du sol !

Exercice 7 — Cercles concentriques (avec boucle)

Consigne

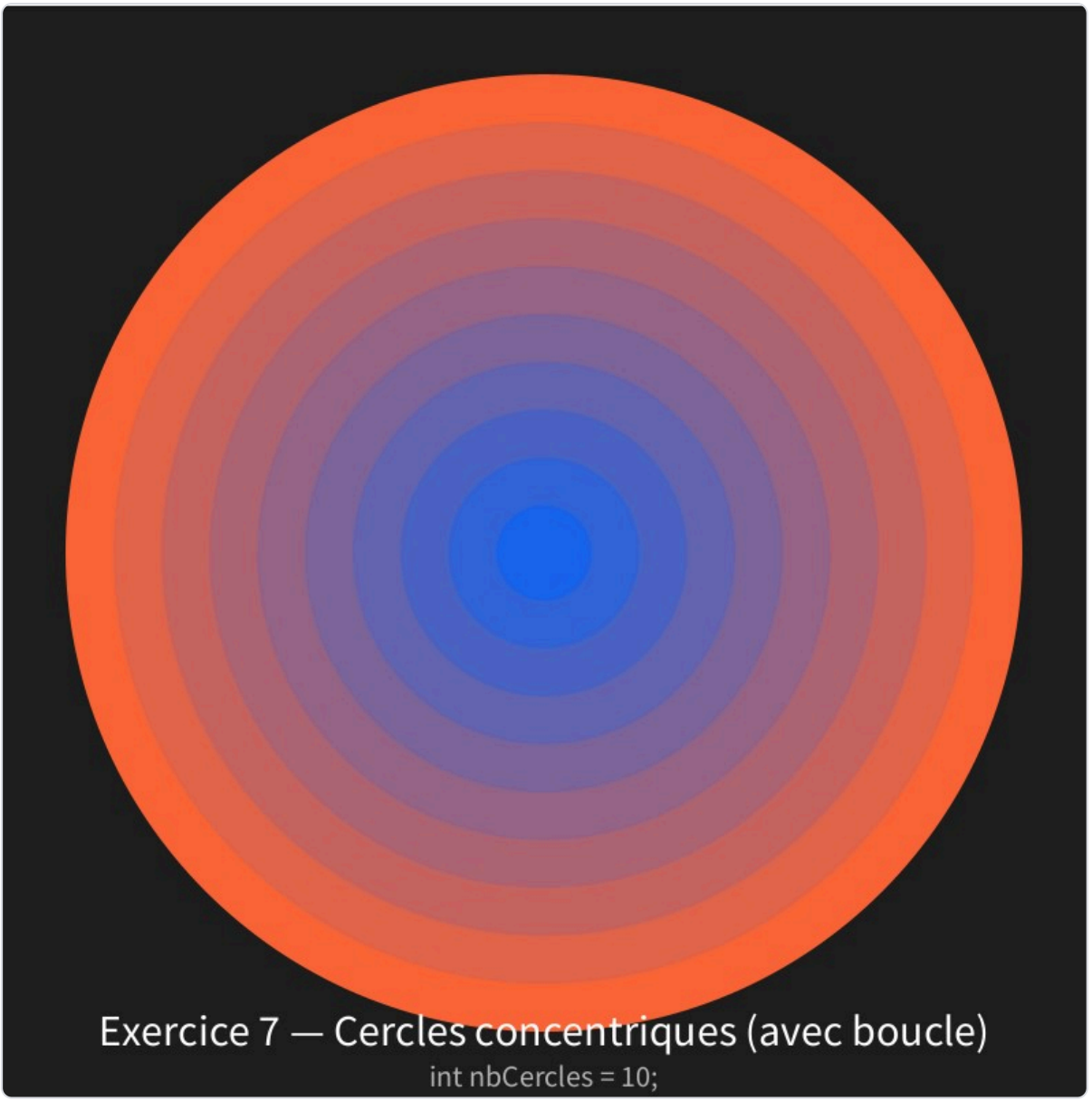
Reprends le principe des cercles concentriques, mais cette fois avec une **boucle** `for`. Le nombre de cercles est stocké dans une variable `nbCercles`. Chaque cercle a une **couleur différente** calculée automatiquement à partir du compteur `i`.

Contraintes

- Canevas de `createCanvas(400, 400)`
- Les cercles sont **centrés** au milieu du canevas
- Le nombre de cercles est dans une variable (commence avec `nbCercles = 8`)
- Chaque cercle a une couleur calculée avec `i` (pas de couleurs écrites à la main)
- `noStroke()` pour un rendu propre

Ce que tu devrais obtenir

Des cercles concentriques avec un dégradé de couleurs automatique, du plus grand (extérieur) au plus petit (intérieur).



Indices

💡 **Indice 1** — Attention à l'**ordre** ! Il faut dessiner du **plus grand au plus petit** (sinon les grands cercles cachent les petits). Astuce : fais ta boucle à l'envers avec `for (let i = nbCercles; i > 0; i--)`, ou bien calcule le rayon en partant du plus grand.

💡 **Indice 2** — Le diamètre du cercle `i` peut être `i * 40`. La couleur peut varier avec `fill(i * 30, 100, 255 - i * 30)`.

💡 **Indice 3** — Pour le bonus : remplace `nbCercles = 8` par `nbCercles = mouseX / 20`. Le nombre de cercles changera en fonction de la position horizontale de la souris !

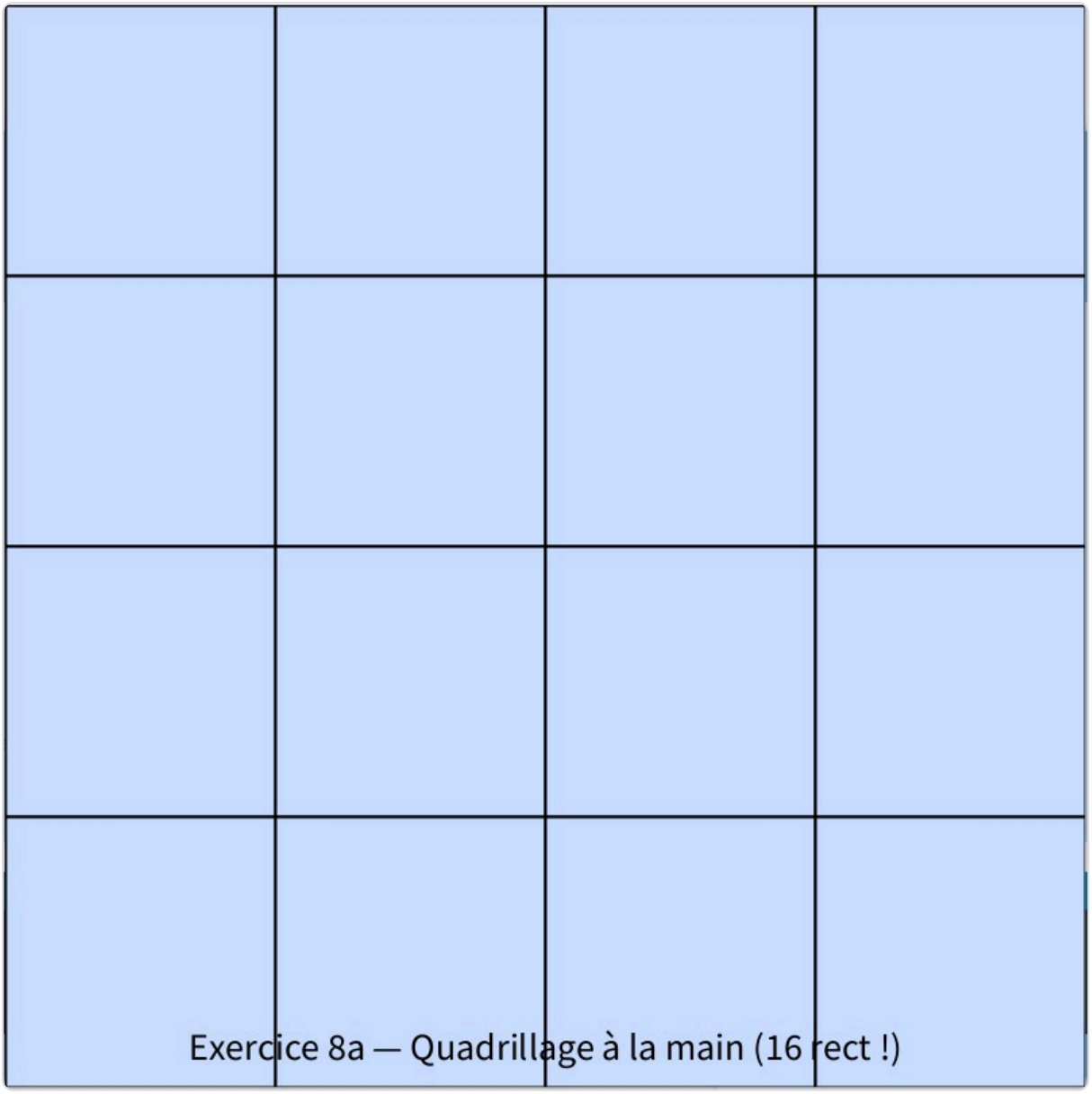
Exercice 8 — Le damier (boucles imbriquées)

Cet exercice se fait en **trois étapes progressives**.

Étape 1 — Quadrillage à la main (sans boucle)

Dessine un quadrillage de **4×4 carrés** de 100×100 pixels, tous de la même couleur. Oui, ça va être long et répétitif — c'est fait exprès 😉

Faites des copier-coller pour aller plus vite !



Étape 2 — Quadrillage avec boucles

Remplace tes 16 lignes de `rect()` par **2 boucles** `for` imbriquées. Le nombre de cases par côté est dans une variable `nbCases`.

Étape 3 — Damier

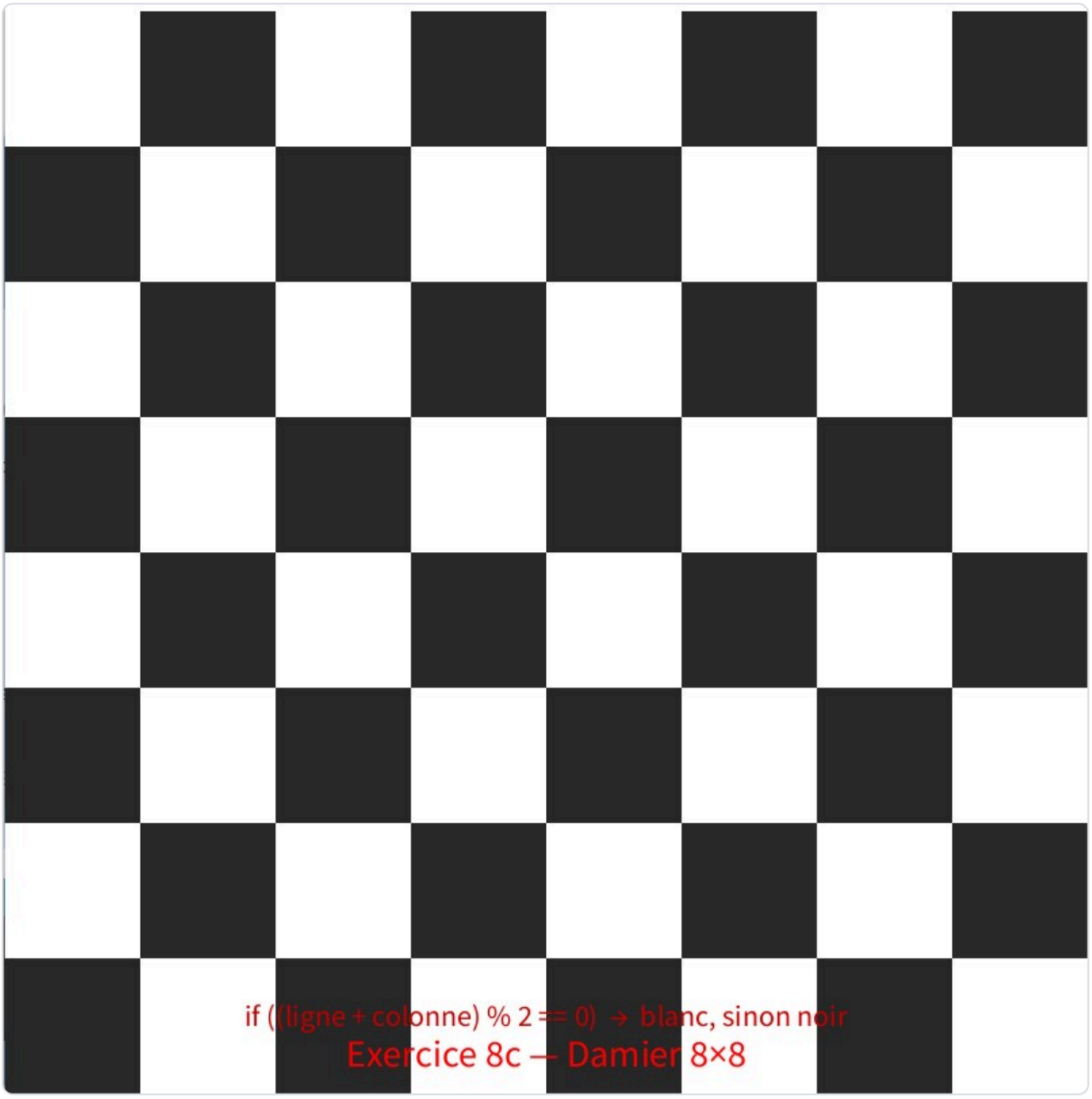
Alterne les couleurs **noir et blanc** comme un échiquier.

Contraintes

- Canevas de `createCanvas(400, 400)` (pour que les 4×4 carrés de 100px remplissent tout l'écran)
- Pour l'étape 2 : la taille des cases se calcule avec `width / nbCases`
- Pour l'étape 3 : utilise une condition `if` pour choisir la couleur

Ce que tu devrais obtenir

Étape 1 → une grille de carrés identiques. Étape 2 → la même grille, mais en 4 lignes de code. Étape 3 → un échiquier noir et blanc.



Indices

💡 **Indice 1 (étape 2)** — La position X d'une case est `colonne * taille` et la position Y est `ligne * taille`, avec `taille = width / nbCases`.

💡 **Indice 2 (étape 3)** — L'astuce du damier repose sur une observation mathématique : si la **somme** du numéro de ligne et du numéro de colonne est **paire**, la case est blanche ; si elle est **impaire**, la case est noire.

💡 **Indice 3 (étape 3)** — Pour savoir si un nombre est pair, on utilise l'opérateur **modulo** `%` (le reste de la division). `(ligne + colonne) % 2` donne `0` si la somme est paire, `1` si elle est impaire :

```javascript
if ((ligne + colonne) % 2 === 0) {
  fill(255);  // Blanc
} else {
  fill(0);    // Noir
}
```

Résumé

Concept	Syntaxe	Exemple
Boucle for	`for (let i = 0; i < n; i++)`	Répéter n fois
Compteur	`i`	Vaut 0, 1, 2, ... n-1
Incrémenter	`i++`	Raccourci pour `i = i + 1`
Position calculée	`debut + i * espacement`	`30 + i * 50`
Couleur calculée	`i * facteur`	`fill(i * 25, 0, 0)`
Boucles imbriquées	2 `for` = grille 2D	`for ligne { for colonne }`
Modulo	`a % b`	Reste de la division → pair/impair

JavaScript classique	p5.js
`for (let i = 0; i < 10; i++)`	`for (let i = 0; i < 10; i++)` (identique !)
`while (condition) { }`	`while (condition) { }` (identique !)

Les points clés à retenir :

- Une boucle **évite le copier-coller** et rend ton code adaptable
- Le compteur `i` est une **variable gratuite** que tu peux utiliser dans tes calculs
- Deux boucles imbriquées = une **grille** (lignes × colonnes)

- Mets le nombre de répétitions dans une **variable** pour pouvoir le changer facilement

Dans le prochain chapitre, on va découvrir les **fonctions** — le moyen de créer tes propres blocs réutilisables pour organiser et simplifier ton code !

2.7 Art génératif

Définition

L'art génératif, c'est concevoir un **système** (règles + algorithmes + hasard contrôlé) qui **produit** l'œuvre. L'artiste définit les paramètres; le programme explore l'«espace des possibles» et génère des variantes uniques (souvent à partir d'une *seed* aléatoire mais **reproductible**).

Origines

- **Années 1960–70** : premiers "algoristes" et **traceurs/plotters** (ex.: Vera Molnár).
- **Années 2000** : démocratisation via **Processing** (Reas/Fry) puis p5.js, openFrameworks, etc.
- **Aujourd'hui** : JavaScript dans le navigateur, shaders/WebGL, IA générative.

Le tournant blockchain

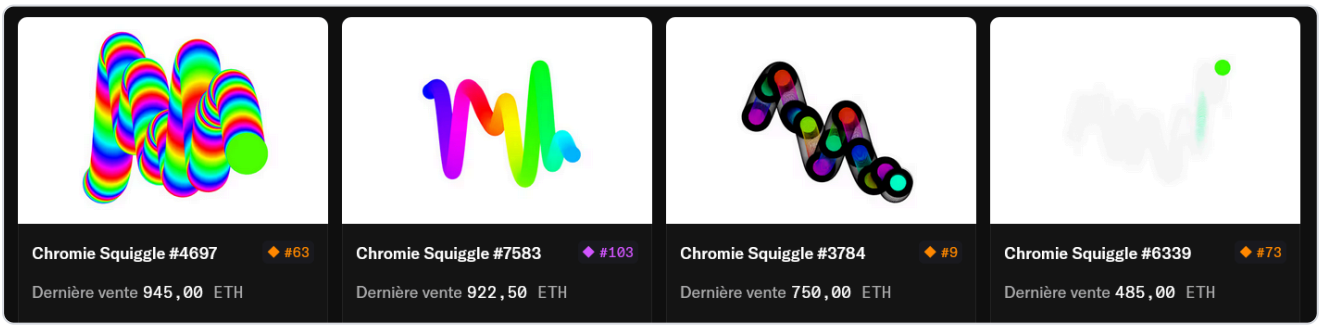
La blockchain apporte la **preuve d'unicité** et l'**exécution publique** d'algorithmes :

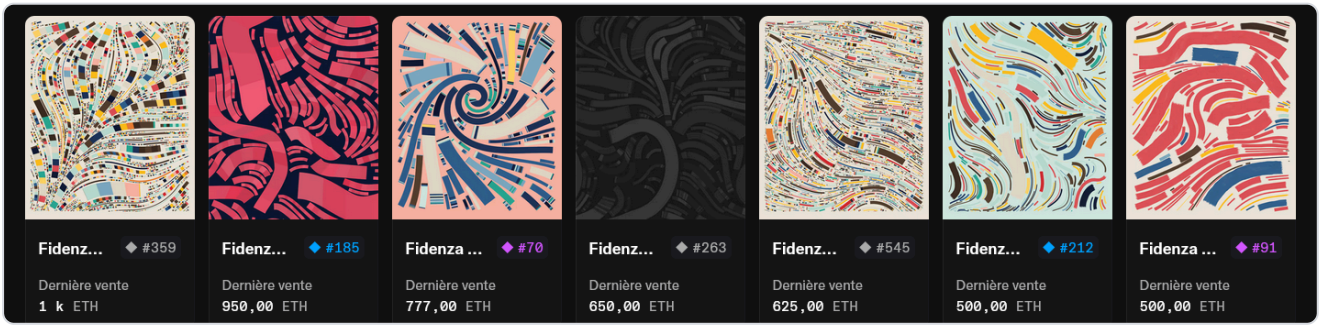
- **CryptoPunks (2017)** : 10k portraits générés par combinaison de traits, icône de la culture NFT.
- **Long-form** : une **seule** œuvre-algorithme génère **toute une série** d'outputs au mint, sans sélection manuelle (exécution autonome, paramètres variés, édition limitée).

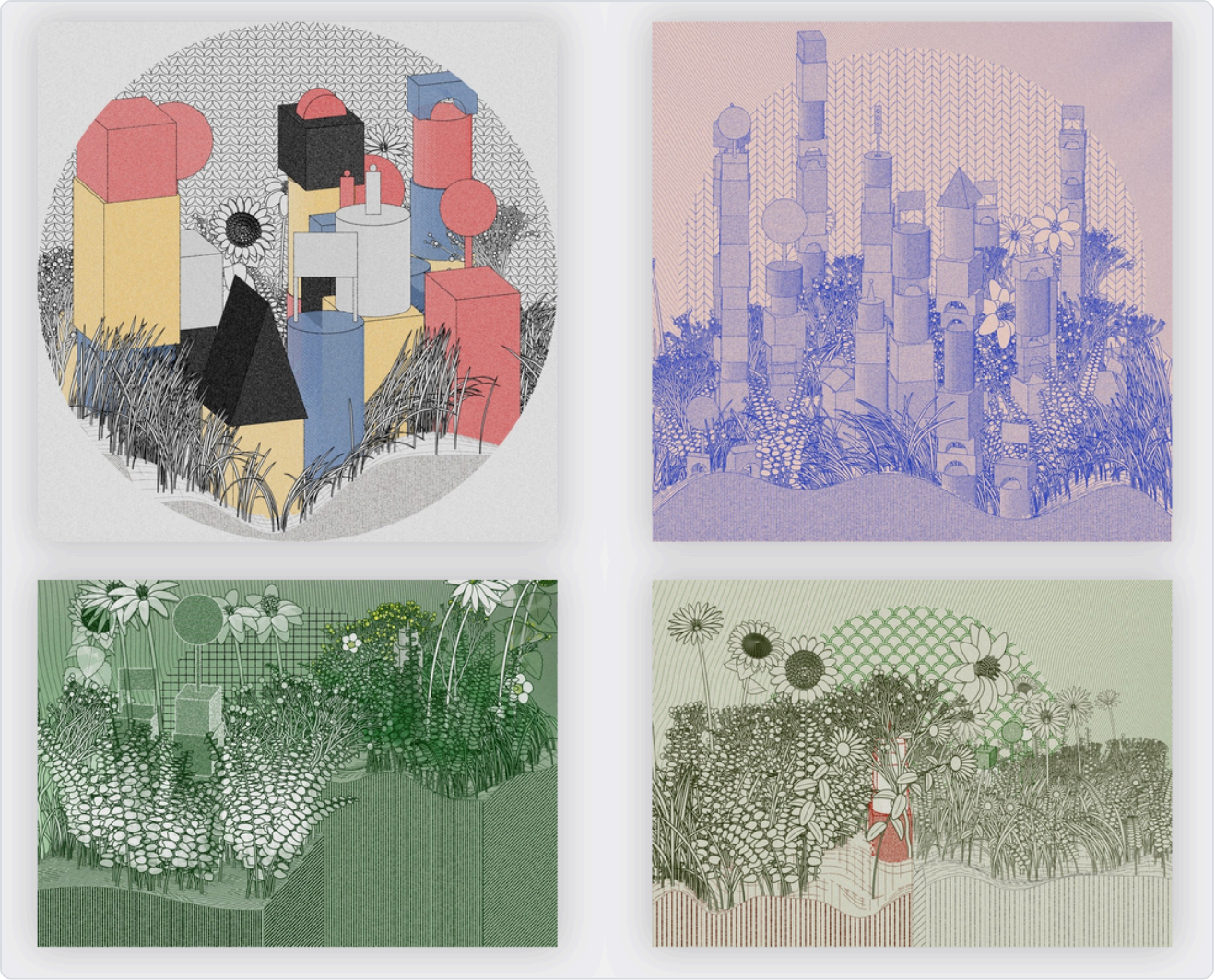


Références clés

- **Chromie Squiggles** (Snowfro, Art Blocks) : lignes ondulées, palette/épaisseur contrôlées par la seed, manifeste du long-form.
- **Fidenza** (Tyler Hobbs) : champs de formes “brossées”, variation riche via règles de composition/couleur/texture.
- **Zancan** : paysages génératifs inspirés du dessin technique (ex.: *Garden, Monoliths*), souvent sur Tezos/fxhash.







Pourquoi c’est utile pour vous

- **Penser en systèmes** (contraintes → styles cohérents).
- **Paramétrer la variation** (design, palettes, grilles).
- **Reproductibilité** (même seed → même visuel).
- **Perf web** : canvas/SVG, shaders, génération à la volée.

Projet — Œuvre générative personnelle (p5.js)

Objectif

Créer une **œuvre d’art génératif personnelle** avec p5.js. À **chaque lancement**, le rendu doit être **différent**. L’œuvre peut être **statique** ou **animée**.

Contraintes

- Réalisation **dans l'éditeur web p5.js** (editor.p5js.org).
- **GPT/IA explicitement encouragés** pour vous aider à atteindre un résultat **unique**.

Pistes de recherche

- **Rectangle packing** (remplissage non chevauchant, densité/échelle variables).
- **Palettes de couleurs** (HSL, harmonies, tirages pondérés).
- **Flow fields** (bruit/ `noise()` pour guider des particules ou des traits).
- **Grilles & perturbations** (symétries, jitter, subdivisions).
- **Textures procédurales** (hachures, pointillisme, "brosses" algorithmiques).
- **Masques & formes composées** (clips, superpositions, transparences).

Remise & présentation

- **À envoyer par e-mail** (le **code du projet**) à **thibault.six@ifosup.wavre.be**
- **Date limite : lundi 15 septembre 2025**
- **Présentation en classe** : démonstration en direct de votre œuvre.

CHAPITRE 3

Vue.js

Slides

Deck 3 — Vue.js et la toolchain (PDF) PDF

Objectif : apprendre à construire un site avec la boîte à outils d'un studio web en 2026 — terminal, Node.js, npm, Vite, Vue.js, Tailwind CSS, Git et VS Code — et s'en servir pour coder ton portfolio.

La toolchain, c'est le métier

Jusqu'ici tu écrivais un fichier `index.html`, tu l'ouvrais dans le navigateur, et ça marchait. C'est très bien pour apprendre. Ce n'est plus comme ça qu'on livre un site.

En agence, en freelance, dans n'importe quelle équipe produit, le travail passe par une **toolchain** : une chaîne d'outils qui s'enchaînent.

Outil	À quoi il sert
Le terminal	Donner des ordres à ta machine, en texte
Node.js	Faire tourner du JavaScript en dehors du navigateur
npm	Installer et lancer les outils du projet
Vite	Le serveur de développement : tu sauvegardes, la page se met à jour
Vue.js	Construire l'interface en composants réactifs
Tailwind CSS	Styler directement dans le HTML, sans fichier CSS à rallonge
Git	Garder l'historique du projet et le mettre en ligne
VS Code	L'éditeur, avec l'autocomplétion qui va avec

Il y a une deuxième raison d'apprendre tout ça, et elle est nouvelle. Les **agents IA** manipulent exactement ces outils : ils lancent des commandes, créent des fichiers, installent des paquets. Si tu ne sais pas ce qu'est un `package.json` ou pourquoi le serveur tourne sur le port 5173, tu ne peux pas relire ce qu'un agent a fait, ni lui dire qu'il s'est trompé. On apprend la chaîne à la main pour pouvoir la diriger.

NOTE

L'IA dans ce cours : autorisée, encouragée même. Mais tu es le directeur. Un directeur qui ne sait pas lire un plan ne dirige rien du tout : il valide au hasard. Ces chapitres te donnent le vocabulaire et les réflexes ; ensuite, délègue tant que tu veux. 🎬

Pourquoi Vue.js

Vue est un framework JavaScript qui règle deux problèmes précis.

1. La réactivité. Tu déclares une donnée, tu l'affiches, et quand la donnée change, l'affichage suit tout seul. Tu ne touches plus au DOM à la main.

2. Les composants réutilisables. Un bouton, une carte projet, une navbar : tu les écris une fois, tu les réutilises partout. Tu corriges à un seul endroit.

Un compteur, deux façons de faire

Voici un compteur en JavaScript classique, sans framework.

```xml
<div id="app">
  <button id="btn">Compteur : <span id="count">0</span></button>
</div>

<script>
  let count = 0;
  const btn = document.getElementById('btn');
  const countSpan = document.getElementById('count');

  btn.addEventListener('click', () => {
    count++;
    countSpan.textContent = count;
  });
</script>
```

Ça fonctionne. Mais remarque la charge de travail : tu dois aller chercher les éléments, écouter le clic, **et** penser à mettre à jour l'affichage toi-même. Sur une page avec trente éléments qui dépendent les uns des autres, tu vas finir par en oublier un. C'est le bug numéro un des sites codés à la main.

Le même compteur en Vue :

```XML
<script setup>
  import { ref } from 'vue'

  const count = ref(0)
</script>

<template>
  <button @click="count++">
    Compteur : {{ count }}
  </button>
</template>

<style scoped>
  button {
    font-weight: bold;
  }
</style>
```

Ce qui se passe, ligne par ligne :

- `ref(0)` crée une donnée **réactive** qui vaut 0 au départ. « Réactive » veut dire : Vue la surveille.
- `@click="count++"` attache un écouteur de clic. C'est l'équivalent de `addEventListener`, en une expression.
- `{{ count }}` affiche la valeur. Dès que `count` change, ce morceau de page se redessine. Tout seul.
- Aucun `getElementById` nulle part. Tu décris le résultat, Vue s'occupe du DOM.

Ce fichier s'appelle un **Single-File Component** (SFC), extension `.vue`. Il tient en trois blocs : `<script setup>` pour la logique, `<template>` pour le HTML, `<style scoped>` pour le CSS. Le mot `scoped` signifie que ce CSS ne s'applique qu'à ce composant : pas de style qui déborde sur le reste du site.

> **ASTUCE**
>
> Retiens la phrase, on va y revenir tout le trimestre : **en vanilla tu décris comment changer la page ; en Vue tu décris ce que la page doit afficher.** 🧠

Le programme de ce chapitre

Dans l'ordre des sous-chapitres :

1. **Le terminal** — se déplacer, créer des dossiers, lancer des commandes.
2. **Node.js et npm** — installer le moteur et le gestionnaire de paquets.
3. **Installation** — créer ton premier projet Vue et le lancer.

4. **Extensions VS Code** — configurer l'éditeur pour qu'il t'aide vraiment.
5. **Structure d'un projet** — comprendre chaque dossier généré.
6. **Tailwind CSS** — l'installer, puis premiers pas guidés.
7. **Les briques de Vue** — composants, `ref`, `v-model`, `v-if`, `v-for`, events, Vue Router.
8. **Le déploiement** — mettre le site en ligne sur Vercel.

Tout ça sert le **fil rouge de l'année** : ton portfolio. Le logo et les maquettes viennent du volet design ; ici, tu les transformes en site réel, en ligne, à ton nom.

Au programme de la première séance

Trois objectifs, pas un de plus :

1. **Te servir du terminal** sans avoir peur de l'écran noir.
2. **Installer ton premier projet Vue** et le voir tourner dans le navigateur.
3. **Configurer VS Code** avec les bonnes extensions.

Récap

- La toolchain moderne (terminal, Node, npm, Vite, Vue, Tailwind, Git, VS Code) est le standard du métier — et ce que les agents IA manipulent.
- Vue apporte la **réactivité** (l'affichage suit les données) et les **composants réutilisables**.
- Un composant Vue est un fichier `.vue` en trois blocs : `<script setup>`, `<template>`, `<style scoped>`.
- Le fil rouge : ton portfolio, codé en Vue + Tailwind, déployé sur Vercel.



3.1 Le terminal (CLI)

> ***Objectif*** *: savoir ouvrir un terminal sur ton ordinateur, comprendre où tu te trouves dans l'arborescence, naviguer entre les dossiers et en créer — sur Windows comme sur macOS.*

Pourquoi le terminal ?

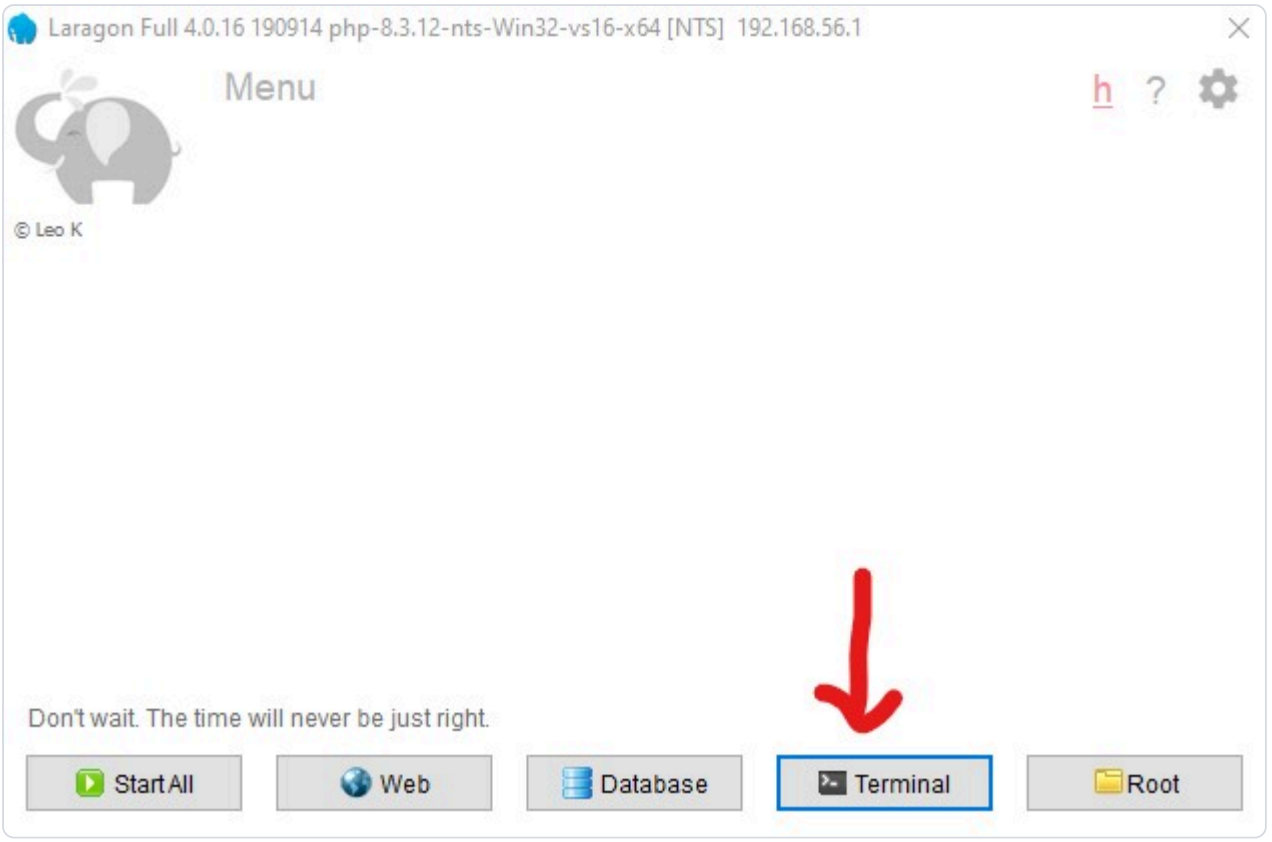
- Les outils qu'on va utiliser toute l'année — **npm**, **git**, **Vite** — se pilotent au terminal, pas à la souris.
- Une commande bien tapée remplace une dizaine de clics dans l'explorateur de fichiers.
- Les tutos, la documentation et les réponses sur Stack Overflow sont écrits en commandes : il faut savoir les lire.
- C'est la même logique sur tous les systèmes : ce que tu apprends ici te resservira partout.

Le terminal s'appelle aussi **CLI** (*Command Line Interface*, interface en ligne de commande). Tu tapes une instruction, tu valides, la machine répond. C'est tout.

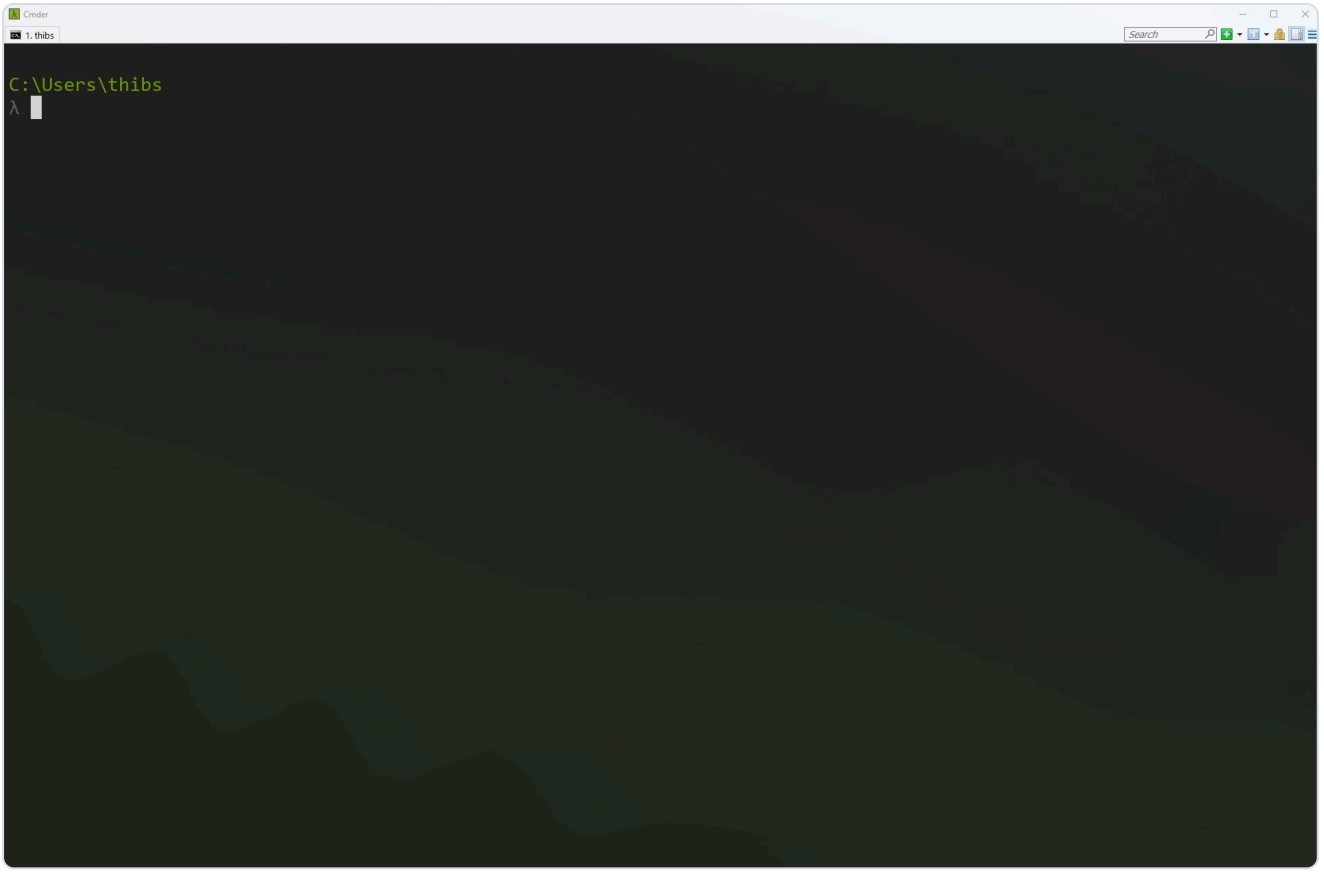
Ouvrir son terminal

Windows : le terminal de Laragon

Dans Laragon, clique sur le bouton **Terminal** en bas à droite de la fenêtre.



Une fenêtre s'ouvre : c'est **Cmder**, le terminal fourni avec Laragon. Il comprend les commandes Unix (`ls`, `touch`...) en plus des commandes Windows, ce qui nous arrange bien.



Le symbole `λ` (lambda) en début de ligne signale que le terminal attend une commande :

```bash
C:\laragon\www
λ
```

> **ATTENTION**
>
> Le terminal de Laragon s'ouvre **par défaut dans** `C:\laragon\www`, pas dans ton dossier utilisateur. C'est le dossier des projets PHP. Pense à en sortir quand tu travailles sur un projet JavaScript.

macOS : Terminal ou VS Code

Deux possibilités :

- **Terminal.app** : appuie sur `Cmd + Espace` (Spotlight), tape « Terminal », valide.
- **Le terminal intégré de VS Code** : menu *Terminal* → *Nouveau terminal*, ou le raccourci `Ctrl + ù`. Pratique, il s'ouvre directement dans le dossier du projet ouvert.

Le shell par défaut est **zsh**, son invite se termine par `%` :

```bash
prenom@MacBook ~ %
```

zsh est le shell par défaut de macOS depuis **Catalina (10.15, 2019)**. Si ton invite se termine par `$` et non par `%`, c'est que tu es sous **bash** : Mac plus ancien, ou réglage conservé lors d'une mise à jour. Aucune importance ici, toutes les commandes de ce chapitre fonctionnent dans les deux shells.

> **NOTE**
>
> Sur Mac, le terminal s'ouvre **dans ton dossier utilisateur** (`/Users/prenom` , aussi noté `~`). C'est exactement là qu'on veut être pour ce cours.

Où suis-je ?

Un terminal est toujours *dans* un dossier : c'est le **dossier courant**. Toutes tes commandes s'appliquent à ce dossier.

La commande `pwd` (*print working directory*) répond à la question « où suis-je ? » :

```bash
pwd
```

Sur Windows :

```bash
C:\laragon\www
λ pwd
C:\laragon\www
```

Sur Mac :

```bash
prenom@MacBook ~ % pwd
/Users/prenom
```

L'invite de commande (le **prompt**) affiche souvent déjà le dossier courant. Prends le réflexe de la lire avant de taper quoi que ce soit : la moitié des erreurs de débutant viennent du fait qu'on n'est pas dans le bon dossier.

Les touches [ENTER] et [TAB]

Deux touches à connaître. On les note `[ENTER]` et `[TAB]` dans ce cours : ce sont des **touches physiques**, tu ne tapes pas les lettres.

- `[ENTER]` (Entrée) : valide et exécute la commande. Tant que tu n'appuies pas dessus, rien ne se passe.
- `[TAB]` (↹, au-dessus de la touche Verr. Maj) : complète automatiquement le nom que tu as commencé à taper. Si plusieurs noms correspondent, appuie deux fois pour voir la liste.

Exemple : tu veux entrer dans `Documents`. Tu tapes `cd Do`, tu appuies sur `[TAB]`, le terminal complète en `cd Documents`, tu appuies sur `[ENTER]`.

```bash
cd Do        # puis [TAB] → cd Documents/ ; puis [ENTER]
```

`[TAB]` t'évite les fautes de frappe, gère les majuscules à ta place et te prouve que le dossier existe vraiment. Utilise-la systématiquement.

À partir d'ici, on n'écrit plus `[ENTER]` dans les exemples : c'est sous-entendu à la fin de chaque commande.

Les commandes

`pwd` — afficher le dossier courant

```bash
pwd
```

`ls` — lister le contenu du dossier

```bash
ls
```

Trois variantes utiles :

```bash
ls -l    # affiche les détails : taille, date de modification
ls -a    # affiche aussi les fichiers cachés (ceux qui commencent par un point)
ls -al   # les deux à la fois : détails + fichiers cachés
```

`ls -al` et `ls -la` font exactement la même chose : l'ordre des lettres n'a pas d'importance. C'est la commande qu'on tape le plus souvent, parce qu'elle montre tout.

Voici à quoi ressemble sa sortie (sur Cmder comme sur Mac, l'affichage est le même) :

```bash
prenom@MacBook repos % ls -al
total 8
drwxr-xr-x   5 prenom  staff   160 20 sep 10:12 .
drwxr-xr-x  18 prenom  staff   576 20 sep 10:11 ..
-rw-r--r--   1 prenom  staff    24 20 sep 10:12 .gitignore
drwxr-xr-x   7 prenom  staff   224 20 sep 10:12 portfolio
-rw-r--r--   1 prenom  staff  1042 20 sep 10:12 notes.md
```

Comment la lire, colonne par colonne :

- **1re colonne** : les droits. Un `d` au début = c'est un dossier, un `-` = c'est un fichier.
- **taille** en octets (la 5e colonne), puis la **date de dernière modification**.
- **dernière colonne** : le nom. `.gitignore` ne s'affiche qu'avec `-a` : il commence par un point, donc il est caché.
- Les deux premières lignes, `.` et `..`, sont le dossier courant et son parent — les mêmes raccourcis que pour `cd`.

`cd` — changer de dossier

```bash
cd nom_du_dossier
```

Les raccourcis, identiques sur les deux plateformes :

- `..` = le dossier parent (un niveau au-dessus)
- `.` = le dossier courant
- `~` = ton dossier utilisateur

```bash
cd ..        # remonter d'un niveau
cd ../..     # remonter de deux niveaux
cd ~         # revenir dans ton dossier utilisateur
cd           # sans argument : même effet que cd ~
```

Aller à la racine du disque — c'est ici que les deux systèmes diffèrent :

```bash
cd \         # Windows : racine du lecteur C:
```

```bash
cd /         # macOS : racine du disque
```

Chemin absolu : il part de la racine et décrit tout le trajet. Il marche depuis n'importe où.

```bash
cd C:\laragon\www\mon-projet      # Windows
```

```bash
cd /Users/prenom/repos/mon-projet # macOS
```

Chemin relatif : il part du dossier où tu es déjà.

```bash
cd www\mon-projet                 # Windows, si tu es dans C:\laragon
```

```bash
cd repos/mon-projet               # macOS, si tu es dans /Users/prenom
```

> **ASTUCE**
>
> Séparateur de dossiers : `\` (antislash) sur Windows, `/` (slash) sur Mac. Cmder accepte souvent les deux, mais garde la convention de ton système, ça t'évitera des surprises.

`mkdir` — créer un dossier

```bash
mkdir nom_du_dossier
mkdir projets tests brouillons   # plusieurs d'un coup
```

`touch` — créer un fichier vide

```bash
touch index.html
touch style.css script.js
```

`clear` — nettoyer l'écran

```bash
clear
```

Ça n'efface rien sur le disque, ça vide juste l'affichage. Pratique quand la fenêtre devient illisible.

Flèche haut — retrouver une commande

La touche `↑` rappelle les commandes déjà tapées, de la plus récente à la plus ancienne. `↓` revient en avant. Corrige, appuie sur `[ENTER]`, c'est reparti.

`code .` — ouvrir le dossier courant dans VS Code

```bash
code .
```

Le point signifie « le dossier où je suis ». C'est la façon la plus rapide d'ouvrir un projet dans l'éditeur.

> **NOTE**
>
> Sur Mac, si `code` n'existe pas encore : ouvre VS Code, appuie sur `Cmd + Shift + P`, tape « shell command » et choisis **Shell Command: Install 'code' command in PATH**. Ferme et rouvre ton terminal, la commande est disponible.

La carte du disque

Windows

```
C:\                          racine du lecteur C:
├── Users\
│   ├── prenom\              ton dossier utilisateur (~)
│   │   ├── Desktop\
│   │   ├── Documents\
│   │   ├── Downloads\
│   │   └── repos\           tes projets JavaScript (Vue, Vite)
│   │       ├── portfolio\
│   │       └── exercices-vue\
│   └── Public\
├── Program Files\
└── laragon\
    └── www\                 tes projets PHP
        ├── mon-site-php\
        └── wordpress-test\
```

macOS

```
/                           racine du disque
├── Applications/
│   └── MAMP/
│       └── htdocs/         tes projets PHP
│           ├── mon-site-php/
│           └── wordpress-test/
├── Users/
│   └── prenom/             ton dossier utilisateur (~)
│       ├── Desktop/
│       ├── Documents/
│       ├── Downloads/
│       └── repos/          tes projets JavaScript (Vue, Vite)
│           ├── portfolio/
│           └── exercices-vue/
└── System/
```

NOTE

La convention du cours : un projet = un dossier.

- `~/repos/` → les **projets JavaScript** de ce cours (Vue, Vite). Un dossier par projet.
- `C:\laragon\www\` (Windows) ou `/Applications/MAMP/htdocs/` (Mac) → les **projets PHP** (vanilla, WordPress...). Un dossier par projet aussi.

On ne mélange jamais les deux. Un projet Vue ne va pas dans `www`, et un site WordPress ne va pas dans `repos`.

Exercices

Exercice 1 — Où suis-je ?

1. Ouvre ton terminal (bouton Terminal de Laragon sur Windows, Spotlight → Terminal ou VS Code sur Mac).
2. Lis le prompt : quel dossier affiche-t-il ?
3. Tape `pwd` et compare avec ce que tu as lu.
4. Tape `ls` pour voir ce que contient ce dossier.

Windows

```bash
C:\laragon\www
λ pwd
C:\laragon\www

C:\laragon\www
λ ls
mon-site-php/  wordpress-test/
```

Laragon t'a déposé dans le dossier des projets PHP. C'est normal.

macOS

```bash
prenom@MacBook ~ % pwd
/Users/prenom

prenom@MacBook ~ % ls
Desktop    Documents    Downloads    Movies    Music    Pictures
```

Tu es dans ton dossier utilisateur, celui que `~` désigne.

Exercice 2 — Retour à la maison

Objectif : aller jusqu'à la racine du disque, puis redescendre jusqu'à ton dossier utilisateur, **en utilisant `[TAB]` à chaque étape**.

Windows : depuis `C:\laragon\www`, remonte jusqu'à la racine `C:\`, puis descends dans `Users`, puis dans ton propre dossier.

macOS : depuis `~`, va à la racine `/`, puis redescends dans `Users`, puis dans ton propre dossier.

Vérifie le résultat avec `pwd` à chaque étape.

Windows

```bash
C:\laragon\www
λ cd ..
C:\laragon
λ cd ..
C:\
λ pwd
C:\

C:\
λ cd Us         # [TAB] complète en Users\
C:\Users
λ cd pre        # [TAB] complète ton prénom
C:\Users\prenom
λ pwd
C:\Users\prenom
```

Tu peux aussi remonter d'un coup avec `cd \`, puis `cd Users`.

macOS

```bash
prenom@MacBook ~ % cd /
prenom@MacBook / % pwd
/

prenom@MacBook / % cd Us        # [TAB] complète en Users/
prenom@MacBook Users % cd pre   # [TAB] complète ton prénom
prenom@MacBook prenom % pwd
/Users/prenom
```

Raccourci : `cd ~` (ou `cd` tout seul) te ramène directement à la maison, depuis n'importe où.

Exercice 3 — Aller voir les projets PHP

Objectif : atteindre le dossier des projets PHP et constater l'organisation « un projet = un dossier ».

Windows : va dans `C:\laragon\www`, puis liste son contenu.

macOS : va dans `/Applications/MAMP/htdocs`, puis liste son contenu. Si MAMP n'est pas installé sur ta machine, le terminal te répondra que le dossier n'existe pas : c'est normal, note-le et passe à l'exercice suivant.

Windows

```bash
C:\Users\prenom
λ cd \laragon\www
C:\laragon\www
λ ls
mon-site-php/  wordpress-test/
```

macOS

```bash
prenom@MacBook ~ % cd /Applications/MAMP/htdocs
prenom@MacBook htdocs % ls
mon-site-php    wordpress-test
```

Si MAMP n'est pas installé :

```bash
prenom@MacBook ~ % cd /Applications/MAMP/htdocs
cd: no such file or directory: /Applications/MAMP/htdocs
```

Dans les deux cas : chaque sous-dossier est **un projet PHP complet**. On n'y met rien d'autre.

Exercice 4 — Remonter dans l'arborescence

Pas de terminal ici, juste de la lecture de chemins.

Windows — tu es dans `C:\laragon\www\mon-projet\assets\images`.

1. Quelle commande pour revenir dans `assets` ?
2. Quelle commande pour revenir dans `www` ?
3. Quelle commande pour aller à la racine `C:\` ?

macOS — tu es dans `/Users/prenom/repos/mon-projet/assets/images`.

1. Quelle commande pour revenir dans `assets` ?
2. Quelle commande pour revenir dans `repos` ?
3. Quelle commande pour revenir dans ton dossier utilisateur ?

Windows

```bash
cd ..              # → C:\laragon\www\mon-projet\assets
cd ..\..\..        # → C:\laragon\www
cd \               # → C:\
```

macOS

```bash
cd ..              # → /Users/prenom/repos/mon-projet/assets
cd ../../..        # → /Users/prenom/repos
cd ~               # → /Users/prenom
```

Chaque `..` remonte d'un cran. On les enchaîne avec le séparateur du système : `\` sur Windows, `/` sur Mac.

Exercice 5 — Prépare ton poste de travail

C'est l'exercice qui compte : tu crées maintenant le dossier dans lequel tu travailleras toute l'année.

1. Reviens dans ton dossier utilisateur.
2. Vérifie avec `pwd` que tu y es bien.
3. Crée un dossier `repos` avec `mkdir`.
4. Entre dedans avec `cd` (utilise `[TAB]`).
5. Vérifie avec `pwd`, puis avec `ls` : le dossier doit être vide.
6. Ouvre-le dans VS Code avec `code .`.

Windows

```bash
C:\laragon\www
λ cd ~
C:\Users\prenom
λ pwd
C:\Users\prenom

C:\Users\prenom
λ mkdir repos

C:\Users\prenom
λ cd rep       # [TAB] complète en repos\
C:\Users\prenom\repos
λ pwd
C:\Users\prenom\repos

C:\Users\prenom\repos
λ ls
# rien : le dossier est vide, c'est normal

C:\Users\prenom\repos
λ code .
```

macOS

```bash
prenom@MacBook ~ % cd ~
prenom@MacBook ~ % pwd
/Users/prenom

prenom@MacBook ~ % mkdir repos
prenom@MacBook ~ % cd rep      # [TAB] complète en repos/
prenom@MacBook repos % pwd
/Users/prenom/repos

prenom@MacBook repos % ls
# rien : le dossier est vide, c'est normal

prenom@MacBook repos % code .
```

> **ASTUCE**
>
> Au chapitre **Installation**, tu lanceras `npm create vue@latest` **depuis ce dossier** `repos`. La commande y créera le dossier de ton premier projet Vue. C'est pour ça qu'on le prépare maintenant.

Récap des commandes

Commande	Action	Remarque
`pwd`	Afficher le dossier courant	Identique Windows / Mac
`ls`	Lister le contenu	`ls -l` détaillé, `ls -a` fichiers cachés
`ls -al`	Lister tout, avec les détails	`ls -la` fait pareil
`cd dossier`	Entrer dans un dossier	Utilise `[TAB]`
`cd ..`	Remonter d'un niveau	Enchaînable : `cd ../..`
`cd ~`	Revenir au dossier utilisateur	`cd` seul fait pareil
`cd \` / `cd /`	Aller à la racine	`\` Windows, `/` Mac
`mkdir nom`	Créer un dossier	Plusieurs noms possibles
`touch fichier`	Créer un fichier vide	Fonctionne dans Cmder
`clear`	Nettoyer l'affichage	N'efface aucun fichier
`code .`	Ouvrir le dossier dans VS Code	Le `.` = dossier courant
`[TAB]`	Autocompléter un nom	Touche physique
`[ENTER]`	Exécuter la commande	Touche physique
`↑`	Rappeler la commande précédente	Touche physique

Erreurs fréquentes

Le dossier n'est pas trouvé alors qu'il existe. Les majuscules comptent : `cd documents` n'est pas `cd Documents`. Utilise `[TAB]`, le problème disparaît.

Mauvais séparateur. `\` sur Windows, `/` sur Mac. Si un chemin est refusé, vérifie d'abord ça.

Rien ne se passe après avoir tapé la commande. Tu as oublié `[ENTER]`.

Tu crées ton projet Vue dans `www`. Sur Windows, Laragon ouvre le terminal dans `C:\laragon\www`. Avant de créer un projet JavaScript, fais toujours `cd ~/repos` et vérifie avec `pwd`.

Tu ne sais plus où tu es. `pwd` répond. `cd ~` te ramène à la maison.

Ressources

- Cheat sheet Bash — antisèche des commandes
- Explainshell — colle une commande, il t'explique chaque morceau
- Cmder — le terminal fourni avec Laragon

3.2 Node.js et npm

Slides

Deck 4 — Node.js et npm (PDF) PDF

Objectif : installer Node.js sur ta machine, comprendre à quoi sert npm, et connaître les cinq commandes que tu vas taper toute l'année.

En une phrase : **Node.js fait tourner du JavaScript en dehors du navigateur, et npm installe les outils dont tes projets ont besoin.**

C'est quoi Node.js ?

Le JavaScript que tu connais s'exécute dans le navigateur. Node.js, c'est le même langage, mais exécuté **directement par ton ordinateur**, sans navigateur.

Pourquoi ça t'intéresse ? Parce que les outils de développement web modernes sont eux-mêmes écrits en JavaScript. Vite, Vue, Tailwind, ESLint : tous ont besoin de Node pour tourner.

Tu ne vas quasiment jamais écrire de code « pour Node ». Tu l'installes une fois, et il fait tourner tes outils en arrière-plan.

C'est quoi npm ?

npm = *Node Package Manager*, le gestionnaire de paquets de Node.

C'est le magasin d'applications des outils de dev. Tu veux Tailwind ? Une commande. Vue ? Une commande. Un paquet, c'est du code écrit par quelqu'un d'autre que tu ajoutes à ton projet.

```bash
npm install tailwindcss
npm install vue
```

Bonne nouvelle : **npm s'installe automatiquement avec Node.js.** Rien à faire de plus.

Vérifier si Node.js est déjà là

Ouvre ton terminal et tape :

```bash
node --version
npm --version
```

Sur Mac, ça ressemble à ça :

```bash
~ % node --version
v24.21.0
~ % npm --version
11.6.2
```

(Chez toi les numéros seront peut-être différents : ce qui compte, c'est que les deux commandes répondent.)

Sur Windows avec le terminal de Laragon :

```bash
C:\Users\prenom
λ node --version
v24.21.0
```

Si tu vois deux numéros de version, c'est installé. Passe directement aux commandes npm.

Si tu vois `command not found` ou `n'est pas reconnu en tant que commande`, c'est que Node n'est pas installé (ou pas trouvé) : suis la section correspondant à ton système ci-dessous.

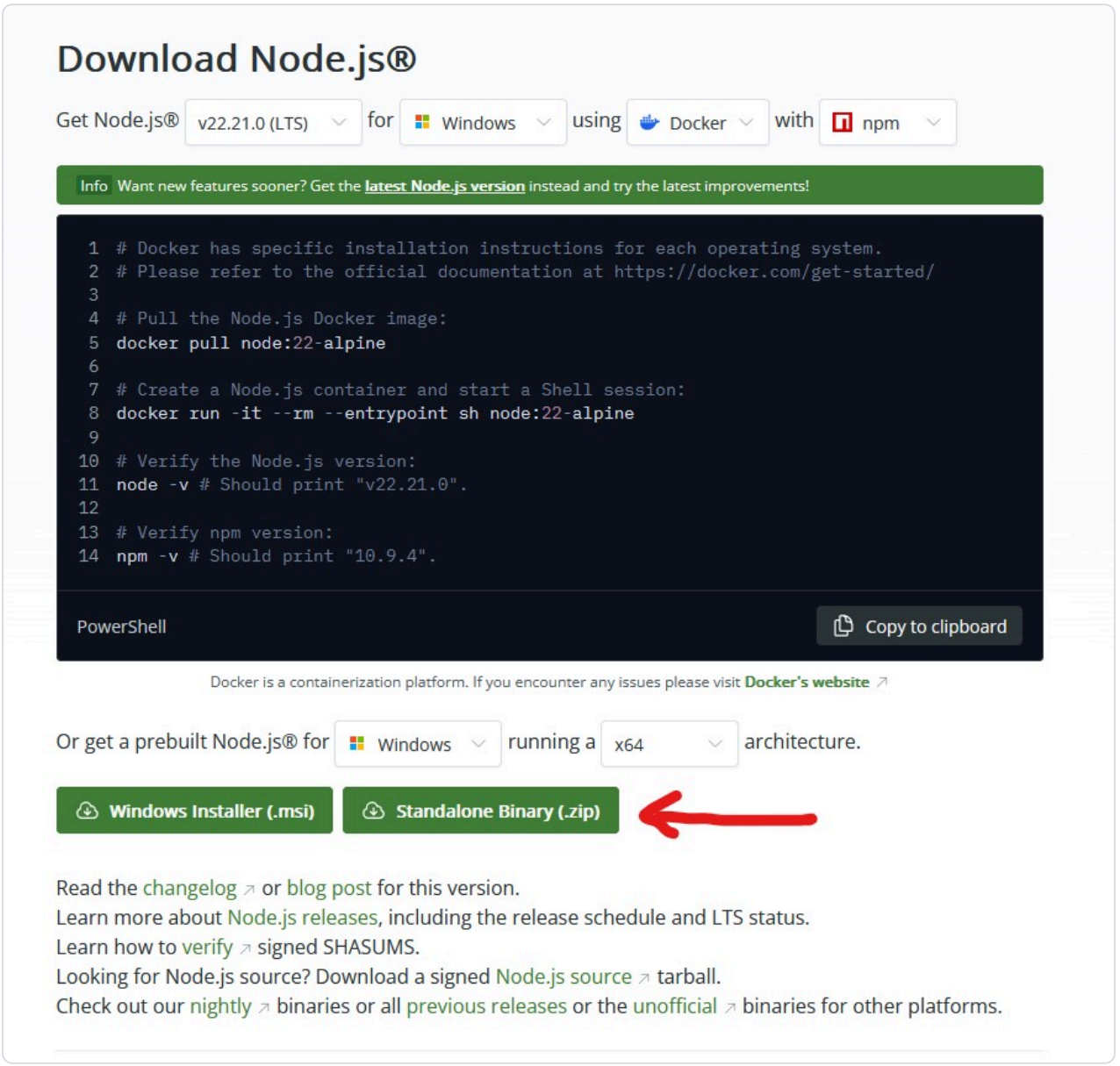
> **NOTE**
>
> **Quelle version choisir ?** Toujours la **LTS** (*Long Term Support*), la version stable maintenue sur la durée. C'est celle que nodejs.org met en avant à gauche du bouton de téléchargement. En septembre 2026, la LTS est la branche **Node 24**. Évite la version « Current » : plus récente, mais moins stable.

Installer Node.js sur Windows (Laragon)

Laragon est livré avec une version de Node, souvent ancienne. On va en ajouter une à jour.

Étape 1 — Télécharger le zip

1. Va sur nodejs.org.
2. Choisis la version **LTS**.
3. **Attention** : prends le **Standalone Binary (.zip)** pour **Windows x64**, pas l'installeur `.msi`. Laragon a besoin du zip pour gérer plusieurs versions.



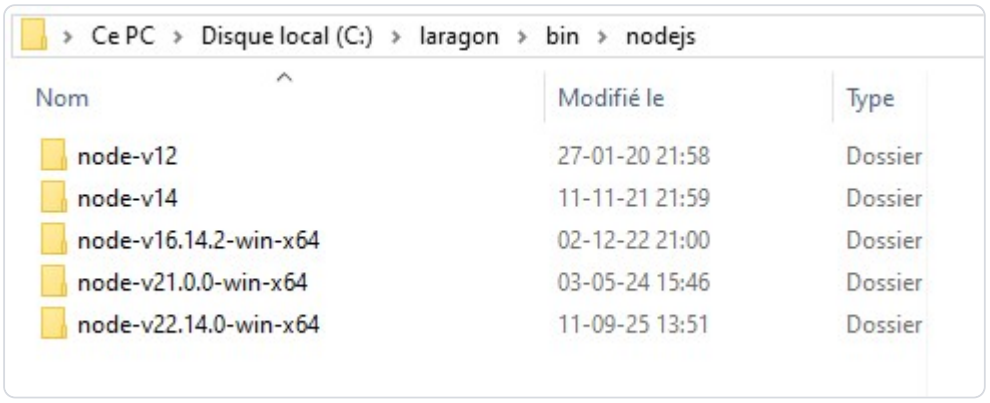
Étape 2 — Décompresser

Extrais le `.zip`. Tu obtiens un dossier qui contient `node.exe`, `npm.cmd` et compagnie.

Étape 3 — Placer le dossier dans Laragon

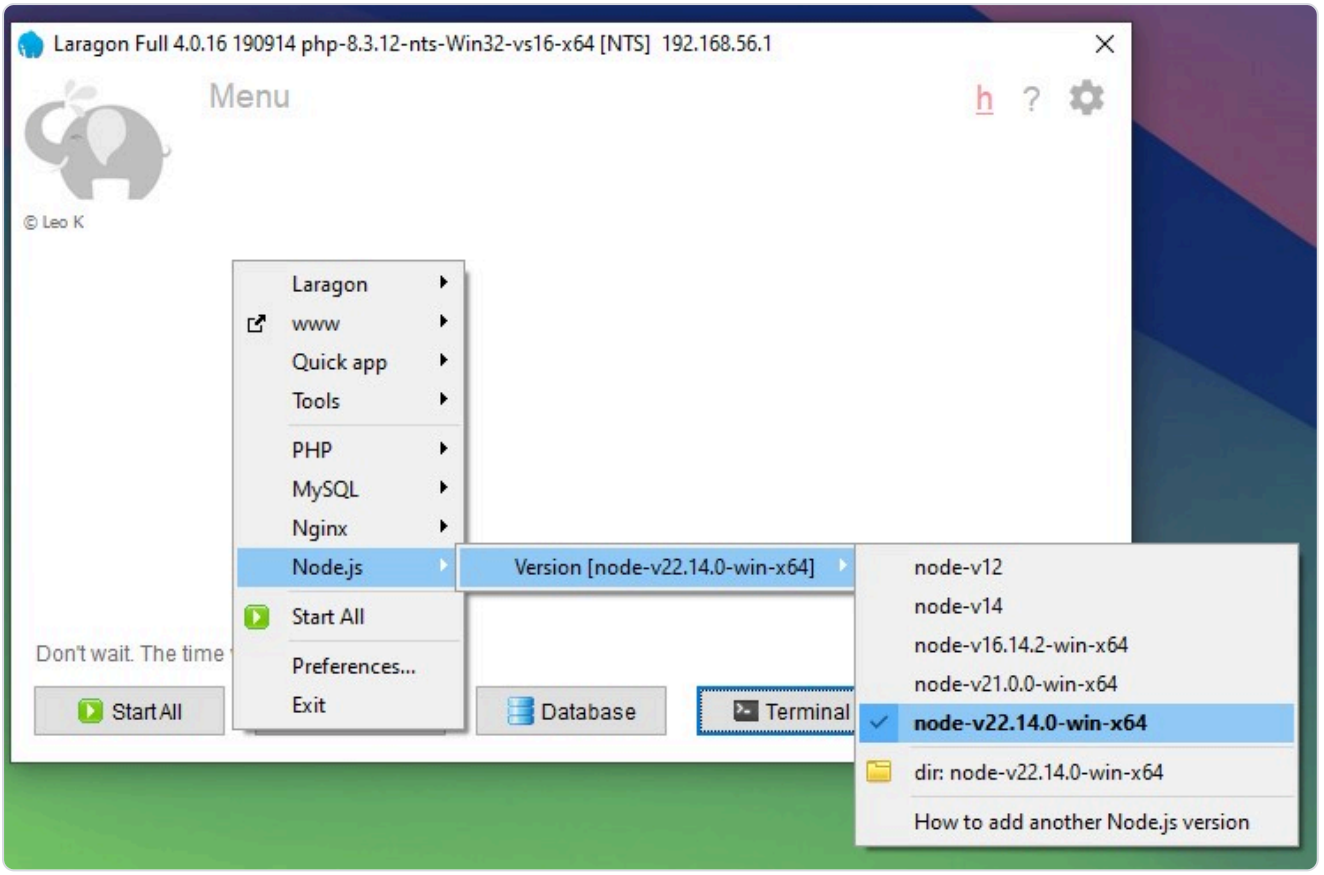
Déplace ce dossier dans `C:\laragon\bin\nodejs\`, à côté des versions déjà présentes.

```
C:\laragon\
└── bin\
    └── nodejs\
        ├── node-v22.11.0-win-x64\   (ancienne version)
        └── node-v24.21.0-win-x64\   (celle que tu viens d'ajouter)
            ├── node.exe
            ├── npm.cmd
            └── ...
```



Étape 4 — Sélectionner la version dans Laragon

1. Ouvre Laragon.
2. Clic droit sur l'icône, ou bouton **Menu**.
3. **Node.js → Version**.
4. Choisis le dossier que tu viens d'ajouter.
5. Laragon redémarre tout seul.



Étape 5 — Vérifier

Ouvre un **nouveau** terminal Laragon (les anciens gardent l'ancienne version en mémoire) :

```bash
C:\Users\prenom
λ node --version
v24.21.0
```

Installer Node.js sur macOS

Sur Mac, pas de Laragon : on passe par l'installeur officiel.

1. Va sur nodejs.org.
2. Télécharge la version **LTS** pour macOS : c'est un fichier `.pkg`.
3. Double-clique dessus et suis l'assistant (suivant, suivant, ton mot de passe de session, terminé).
4. **Ferme complètement ton terminal et rouvre-le.** C'est l'étape que tout le monde saute, et c'est celle qui fait croire que l'installation a raté.
5. Vérifie :

```bash
~ % node --version
v24.21.0
```

ASTUCE

Le `.pkg` installe aussi npm. Si un jour tu veux jongler entre plusieurs versions de Node sur Mac, on te parlera de `nvm` — mais pas cette année, tu n'en as pas besoin. 🙂

Où lancer les commandes : la règle à ne pas oublier

ATTENTION

Les projets JavaScript de ce cours (Vue, Vite) vivent dans **`~/repos`**, un dossier par projet. Jamais dans `C:\laragon\www` ni dans `htdocs` : ces dossiers-là sont pour les projets PHP.

Donc avant de taper `npm create vue@latest`, tu te places d'abord dans `~/repos` :

```bash
cd ~/repos
```

Un projet Vue n'a pas besoin d'Apache ni de PHP : il a son propre serveur, lancé par Vite. Le mettre dans `www` ne sert à rien et crée des problèmes de chemins.

Les 5 commandes npm à connaître

1. `npm install`

Quand tu récupères un projet existant (le tien sur une autre machine, celui d'un collègue, un dépôt Git).

```bash
npm install
```

Elle lit le fichier `package.json` et télécharge tous les paquets listés dedans.

2. `npm install <paquet>`

Pour ajouter un outil à ton projet.

```bash
npm install tailwindcss
npm install @tailwindcss/vite
```

3. `npm run dev`

Pour travailler sur ton projet : elle lance le serveur de développement, avec rechargement automatique à chaque sauvegarde.

```bash
npm run dev
```

Le site devient accessible sur `http://localhost:5173`.

4. `npm run build`

Pour préparer la mise en ligne : elle génère une version optimisée dans un dossier `dist/`.

```bash
npm run build
```

5. `npm create <modèle>`

Pour démarrer un nouveau projet à partir d'un modèle.

```bash
npm create vue@latest
```

Un assistant te pose quelques questions et génère la structure de base. C'est l'objet du chapitre suivant.

Le fichier `package.json`

À la racine de chaque projet npm, tu trouves un `package.json`. C'est la carte d'identité du projet : son nom, ses commandes, ses dépendances.

```json
{
  "name": "mon-portfolio",
  "version": "0.0.0",
  "private": true,
  "type": "module",
  "scripts": {
    "dev": "vite",
    "build": "vite build",
    "preview": "vite preview"
  },
  "dependencies": {
    "vue": "^3.5.42"
  },
  "devDependencies": {
    "vite": "^8.2.2"
  }
}
```

Ce qu'il faut savoir lire :

- `scripts` : les commandes disponibles. Une ligne `"dev": "vite"` signifie que `npm run dev` lance Vite. Si tu ne sais plus quelles commandes existent dans un projet, c'est ici que tu regardes.
- `dependencies` : les paquets nécessaires au site une fois en ligne (Vue, par exemple).
- `devDependencies` : les paquets utiles seulement pendant le développement (Vite, ESLint, Prettier). Ils ne partent pas chez le visiteur.

Tu n'édites presque jamais ce fichier à la main : npm le met à jour à chaque `npm install`.

Le dossier `node_modules/`

Après un `npm install`, un gros dossier `node_modules/` apparaît. Il contient tous les paquets téléchargés, et il pèse facilement plusieurs centaines de mégaoctets.

Ce que tu en fais : **rien**.

- Tu ne l'édites pas.
- Tu ne l'envoies pas sur Git (le `.gitignore` généré l'exclut déjà).
- Tu ne le mets pas dans un zip pour partager ton projet.

Pourquoi ? Parce qu'il est **reconstructible**. Avec le `package.json`, n'importe qui retrouve exactement les mêmes paquets en tapant `npm install`. On partage la recette, pas les courses. 🛒

Récap

Élément	Ce que c'est	Ce que tu en fais
Node.js	JavaScript hors du navigateur	Tu l'installes en version LTS, puis tu l'oublies
npm	Le gestionnaire de paquets	Tu t'en sers tous les jours
`package.json`	Carte d'identité du projet	Tu le lis, npm l'écrit
`node_modules/`	Les paquets téléchargés	Tu l'ignores, jamais sur Git
`npm install`	Installe les dépendances	Premier réflexe sur un projet récupéré
`npm run dev`	Lance le serveur de dev	Premier réflexe quand tu bosses
`~/repos`	Le dossier des projets JS	Jamais dans `www`

Node et npm sont prêts. Au chapitre suivant, on crée enfin le projet Vue.

Pour aller plus loin

- nodejs.org — télécharger Node.js
- docs.npmjs.com — la documentation npm
- npmjs.com — chercher un paquet

3.3 Installation

Slides

Deck 5 — Installation du premier projet Vue (PDF) PDF

Objectif : créer ton premier projet Vue avec `npm create vue@latest`, le lancer dans le navigateur, et l'ouvrir dans VS Code.

Prérequis

Avant de commencer, trois choses :

- **Node.js en version LTS** installé. Si `node --version` ne répond pas un numéro, reviens au chapitre Node.js et npm.
- **Ton terminal ouvert dans `~/repos`**. C'est là que vivent les projets JavaScript.
- **VS Code installé** sur la machine.

```bash
cd ~/repos
node --version
```

ATTENTION

Sous Windows, le terminal Laragon s'ouvre par défaut dans `C:\laragon\www`. **Ne crée pas ton projet Vue là.** Tape `cd ~/repos` avant tout le reste, et vérifie avec `pwd` où tu te trouves.

Créer le projet

Une seule commande :

```bash
npm create vue@latest
```

npm télécharge **create-vue**, l'assistant officiel de Vue, et te pose une série de questions.

Les questions, une par une

Voici ce que l'assistant affiche, dans l'ordre :

```
Nom du projet : › vue-project

Utiliser TypeScript ? › Oui / Non

Sélectionnez les fonctionnalités à inclure dans votre projet :
(↑/↓ pour naviguer, espace pour sélectionner, a pour tout sélectionner, entrée pour
confirmer)
  ◻ Support de JSX
  ◻ Router (développement SPA)
  ◻ Pinia (gestion de l'état)
  ◻ Vitest (tests unitaires)
  ◻ Tests de bout en bout
  ◻ Linter (prévention des erreurs)
  ◻ Prettier (formatage du code)

Sélectionnez les fonctionnalités expérimentales à inclure :
  ◻ Remplacer Prettier par Oxfmt
  ◻ Vue 3.6 (version candidate)

Ignorer tout le code d'exemple et commencer avec un projet Vue vierge ? › Oui / Non
```

Les réponses pour ce cours

Nom du projet : `mon-portfolio`.

En **minuscules**, **sans espace**, **sans accent**. Les mots se séparent par un tiret. C'est à la fois le nom du dossier et le nom du paquet npm : un espace ou une majuscule et l'assistant refuse ou te repose la question.

Utiliser TypeScript ? → Non.

Fais attention : la réponse présélectionnée est **Oui**. Appuie sur la flèche pour choisir `Non` avant de valider. TypeScript est excellent, mais c'est un autre langage à apprendre par-dessus Vue ; ce n'est pas le moment.

Sélectionnez les fonctionnalités → coche uniquement `Linter (prévention des erreurs)` et `Prettier (formatage du code)`. Rien d'autre.

Tu te déplaces avec les flèches, tu coches avec la **barre d'espace**, tu valides avec `[ENTER]`. Le Linter (ESLint, accompagné d'Oxlint) signale tes erreurs pendant que tu codes, Prettier remet ton code en forme tout seul. Les deux te font gagner du temps dès la première heure.

Et **Router** ? On l'ajoutera plus tard, quand ton portfolio aura plusieurs pages. Un chapitre entier lui est consacré. Pinia, Vitest, les tests de bout en bout et JSX ne servent pas à ce cours.

Fonctionnalités expérimentales → rien du tout. Valide avec `[ENTER]` sans rien cocher.

Ignorer tout le code d'exemple ? → Non.

On garde le code d'exemple : il sert de visite guidée du projet, et on le supprimera nous-mêmes au moment de construire le portfolio.

> **NOTE**
>
> L'assistant est traduit : si ton système est en anglais, tu verras « Project name », « Use TypeScript? », « Select features to include in your project ». Les questions et l'ordre sont exactement les mêmes. 🌍

Installer les dépendances et lancer le serveur

L'assistant termine en te donnant les commandes suivantes. Tape-les dans l'ordre :

```bash
cd mon-portfolio
npm install
npm run dev
```

- `cd mon-portfolio` entre dans le dossier fraîchement créé.
- `npm install` télécharge Vue, Vite et le reste dans `node_modules/`. Compte une poignée de secondes la première fois.
- `npm run dev` démarre le serveur de développement.

Le terminal affiche quelque chose comme :

```bash
VITE v8.3.0  ready in 849 ms

➜  Local:   http://localhost:5173/
➜  Network: use --host to expose
```

Ouvre **http://localhost:5173** dans ton navigateur. Tu vois la page d'accueil générée par create-vue : le logo Vue en haut à gauche, le titre « You did it! », et une colonne de liens vers la documentation, l'écosystème et la communauté. C'est le code d'exemple qu'on a gardé.

Laisse cette page ouverte. À chaque fois que tu sauvegarderas un fichier du projet, elle se mettra à jour toute seule. C'est le **rechargement à chaud**, et c'est la raison d'être de Vite.

Arrêter et relancer le serveur

Tant que le serveur tourne, **ton terminal est occupé** : il ne te rendra pas la main. C'est normal.

- Pour **arrêter** le serveur : clique dans le terminal et fais `Ctrl + C` (oui, `Ctrl` aussi sur Mac, pas `Cmd`).
- Pour le **relancer** : `npm run dev`, depuis le dossier du projet.

Tu peux aussi ouvrir un **deuxième terminal** pour taper d'autres commandes pendant que le serveur tourne. C'est ce que font la plupart des développeurs.

Ouvrir le projet dans VS Code

Depuis le dossier du projet :

```bash
code .
```

Le point signifie « le dossier courant ». VS Code s'ouvre sur ton projet.

> **ASTUCE**
>
> Sur Mac, si `code` n'est pas reconnu, ouvre VS Code, fais `Cmd + Shift + P`, tape « Shell Command: Install 'code' command in PATH » et valide. Une seule fois, et c'est réglé pour toujours. 🍏

Il reste à installer les extensions qui rendent VS Code réellement utile pour Vue (Vue - Official, Tailwind CSS IntelliSense, ESLint, Prettier) : c'est l'objet du chapitre **Extensions VS Code**.

Ça ne marche pas ?

> **ATTENTION**
>
> `node : commande introuvable` / `n'est pas reconnu` Node n'est pas installé, ou ton terminal date d'avant l'installation. Ferme-le, rouvre-en un neuf, retente. Si ça persiste, reprends le chapitre Node.js et npm.

Le port 5173 est occupé. Vite te le dit et bascule tout seul sur 5174, 5175… Utilise le port qu'il affiche. Si tu veux récupérer 5173, c'est qu'un ancien serveur tourne encore : retrouve son terminal et fais `Ctrl + C`.

Le projet a atterri dans `www`. Tu as lancé la commande sans avoir fait `cd ~/repos`. Ferme le serveur, déplace le dossier du projet dans `~/repos` (glisser-déposer dans l'explorateur suffit), puis relance `npm run dev` depuis son nouvel emplacement.

Le nom du projet contient un espace ou une majuscule. L'assistant te repose la question sous la forme « Nom du package : » avec une proposition corrigée. Le plus simple : supprime le dossier et recommence avec un nom propre, tout en minuscules avec des tirets.

`npm run dev` renvoie une erreur de module introuvable. Tu as sauté `npm install`. Tape-le, puis relance.

`npm run dev` dit « Missing script: dev ». Tu n'es pas dans le bon dossier. Vérifie avec `pwd` (ou `ls` : tu dois voir `package.json`), puis `cd` dans le dossier du projet.

Récap : les 4 commandes

```bash
# on se place d'abord dans le bon dossier
cd ~/repos

npm create vue@latest
cd mon-portfolio
npm install
npm run dev
```

À retenir :

- Toujours depuis `~/repos`, jamais depuis `www`.
- Dans l'assistant : **TypeScript** → **Non**, et on coche **uniquement Linter et Prettier**.
- **Ctrl + C** arrête le serveur, `npm run dev` le relance.
- http://localhost:5173 est l'adresse de ton projet pendant le développement.

Au chapitre suivant, **Extensions VS Code**, on équipe l'éditeur ; on ouvrira les dossiers générés juste après, au chapitre **Structure d'un projet**.

3.4 Extensions VS Code

Objectif : configurer VS Code pour travailler avec Vue et Tailwind, en 5 minutes chrono.

Tu viens d'ouvrir ton projet avec `code .`. Avant d'écrire la moindre ligne, on installe quatre extensions. Sans elles, VS Code ne comprend pas les fichiers `.vue` et ne t'aide pas du tout sur Tailwind. Avec elles, l'éditeur devient un vrai copilote : il complète, il corrige, il prévient.

Installer une extension

Toujours le même geste :

1. Ouvre l'onglet **Extensions** : `Ctrl+Shift+X` (Windows) ou `Cmd+Shift+X` (Mac).
2. Tape l'**identifiant** de l'extension dans la barre de recherche (c'est plus fiable que le nom : il n'y a qu'un seul résultat).
3. Clique sur **Install**.

> **ASTUCE**
>
> Un identifiant s'écrit `éditeur.nom-du-paquet`, par exemple `Vue.volar`. Copie-le tel quel dans la recherche.

1. Vue - Official — obligatoire

Identifiant : `Vue.volar` · Voir sur le Marketplace

C'est l'extension officielle de l'équipe Vue. Elle apprend à VS Code à lire les fichiers `.vue` : coloration du `<template>`, du `<script>` et du `<style>` dans un même fichier, autocomplétion de tes composants et de leurs props, erreurs soulignées pendant que tu tapes.

Sans elle, un fichier `.vue` s'affiche en texte gris, sans aide ni détection d'erreur. Elle est donc obligatoire pour ce cours.

> **ATTENTION**
>
> Si tu as déjà installé **Vetur** (l'ancienne extension Vue, aujourd'hui abandonnée), **désinstalle-la** : les deux se marchent dessus et tu récoltes des erreurs fantômes. Cherche `octref.vetur` dans l'onglet Extensions, puis **Uninstall**.

2. Tailwind CSS IntelliSense — obligatoire

Identifiant : `bradlc.vscode-tailwindcss` · Voir sur le Marketplace

C'est l'extension qui rend Tailwind apprenable. Tailwind, ce sont des centaines de petites classes ; personne ne les retient par cœur, et personne n'est censé les retenir. L'extension te les propose au fur et à mesure.

Concrètement, dans un `class="..."` :

- **Autocomplétion** : tu tapes `bg-` et la liste complète des couleurs s'affiche, avec un carré de couleur à côté de chaque nom. Tu apprends Tailwind en le lisant.
- **Aperçu des couleurs** : `text-slate-700` affiche sa pastille directement dans le code.
- **Survol** : passe la souris sur `p-4` et tu vois le CSS généré (`padding: 1rem;`). C'est le moyen le plus rapide de comprendre ce que fait une classe.
- **Classes inconnues** : si tu écris `bg-blue-1000` (qui n'existe pas), l'extension ne te propose rien — un signal immédiat que tu t'es trompé.

> **NOTE**
>
> L'extension ne s'active que dans un projet où Tailwind est **installé**. Tant que tu n'as pas fait l'installation (voir le chapitre consacré à Tailwind), l'autocomplétion reste muette : c'est normal, ce n'est pas un bug.

3. ESLint — recommandé

Identifiant : `dbaeumer.vscode-eslint` · Voir sur le Marketplace

ESLint relit ton code et souligne les erreurs de logique et les maladresses : variable déclarée jamais utilisée, composant importé mais absent du template, `=` écrit à la place de `==`. Le projet généré par `npm create vue@latest` contient déjà sa configuration ; l'extension ne fait que l'afficher dans l'éditeur.

4. Prettier — recommandé

Identifiant : `esbenp.prettier-vscode` · Voir sur le Marketplace

Prettier remet ton code en forme : indentation, guillemets, retours à la ligne. Tu ne discutes plus jamais de mise en forme, tu enregistres et c'est propre.

Le déclencher à chaque sauvegarde

Tout se fait dans l'interface des réglages, sans toucher à un fichier de configuration.

1. Ouvre les réglages : `Ctrl+,` (Windows) ou `Cmd+,` (Mac). Par les menus : *Fichier* → *Préférences* → *Paramètres* (Windows), *Code* → *Réglages* → *Paramètres* (Mac).
2. Dans la barre de recherche en haut, tape `format on save`.
3. Coche la case **Editor: Format On Save**.

Puis, pour désigner Prettier comme metteur en forme :

1. Toujours dans les réglages, cherche `default formatter`.
2. Dans la liste déroulante **Editor: Default Formatter**, choisis **Prettier - Code formatter**.

C'est tout : ce réglage est global, il vaut pour tous tes projets et pour la plupart des fichiers.

Si un fichier refuse de se mettre en forme, c'est que VS Code hésite entre plusieurs metteurs en forme pour ce langage. On tranche fichier par langage :

1. Ouvre un fichier `.vue`.
2. Palette de commandes : `Ctrl+Shift+P` (Windows) ou `Cmd+Shift+P` (Mac).
3. Tape **Format Document With...** et valide.
4. En bas de la liste, choisis **Configure Default Formatter...**, puis **Prettier - Code formatter**.

Recommence la même manipulation depuis un fichier `.js`. Chaque langage garde ensuite son réglage.

Vérification : ouvre `App.vue`, désindente une ligne au hasard, sauvegarde avec `Ctrl+S` / `Cmd+S`. La ligne se remet en place toute seule. Si rien ne bouge, reprends les étapes ci-dessus.

Les mêmes réglages, en texte. Palette de commandes → `Preferences: Open User Settings (JSON)`, puis :

```json
{
  "editor.formatOnSave": true,
  "editor.defaultFormatter": "esbenp.prettier-vscode"
}
```

C'est strictement équivalent aux clics décrits plus haut : l'interface écrit ce fichier pour toi.

ASTUCE

Un projet Vue tout neuf embarque un fichier `.vscode/extensions.json` : à l'ouverture du dossier, VS Code te propose lui-même ces extensions dans une petite notification. Tu peux tout accepter d'un clic.

Word Wrap (retour à la ligne automatique)

Avec Tailwind, une balise finit vite avec quinze classes dans son `class="..."`. Sans retour à la ligne automatique, la ligne file vers la droite et sort de l'écran : tu passes ton temps à faire défiler horizontalement pour lire ton propre code.

Le Word Wrap coupe visuellement les longues lignes à la largeur de la fenêtre. Il ne modifie pas le fichier, c'est un confort d'affichage.

- **Ponctuellement** : menu *Affichage → Retour automatique à la ligne* (*View → Word Wrap*), ou le raccourci `Alt+Z` (Windows) / `Option+Z` (Mac). Le même raccourci le désactive.
- **Définitivement** : dans les réglages (`Ctrl+,` / `Cmd+,`), cherche `word wrap` et mets **Editor: Word Wrap** sur **on**.

NOTE

Rien à installer côté navigateur pour inspecter tes composants : le projet généré par `npm create vue@latest` embarque déjà `vite-plugin-vue-devtools`. Quand tu lances `npm run dev`, un petit bouton Vue flotte en bas de la page : il ouvre le panneau qui montre l'arbre de tes composants et la valeur de chaque donnée réactive, en direct.

Récapitulatif

Extension	Identifiant	Statut
Vue - Official	`Vue.volar`	Obligatoire
Tailwind CSS IntelliSense	`bradlc.vscode-tailwindcss`	Obligatoire
ESLint	`dbaeumer.vscode-eslint`	Recommandé
Prettier	`esbenp.prettier-vscode`	Recommandé

Et deux réglages de l'éditeur :

Réglage	Où	Effet
Editor: Format On Save	Réglages, recherche `format on save`	Met le fichier en forme à chaque sauvegarde
Editor: Word Wrap	Réglages, recherche `word wrap` (ou `Alt+Z` / `Option+Z`)	Coupe les longues lignes de classes Tailwind

Installation en 5 minutes

1. `Ctrl+Shift+X` / `Cmd+Shift+X`, puis installe `Vue.volar`, `bradlc.vscode-tailwindcss`, `dbaeumer.vscode-eslint`, `esbenp.prettier-vscode`.
2. Désinstalle Vetur si tu l'avais (`octref.vetur`).
3. Réglages (`Ctrl+,` / `Cmd+,`) → `format on save` → coche **Editor: Format On Save**.
4. Toujours dans les réglages → `default formatter` → choisis **Prettier - Code formatter**.
5. Réglages → `word wrap` → mets **Editor: Word Wrap** sur **on**.
6. Relance VS Code pour être tranquille.

En résumé

Quatre extensions et trois réglages. Volar pour comprendre les fichiers `.vue`, Tailwind IntelliSense pour ne plus jamais chercher un nom de classe, ESLint et Prettier pour un code propre sans y penser, Format On Save pour que ça se fasse tout seul et Word Wrap pour garder tes longues lignes de classes à l'écran. Côté navigateur, rien à installer : les devtools Vue sont déjà dans le projet. Ton éditeur est prêt : on peut ouvrir le projet et regarder ce qu'il y a dedans.

3.5 Structure d'un projet Vue

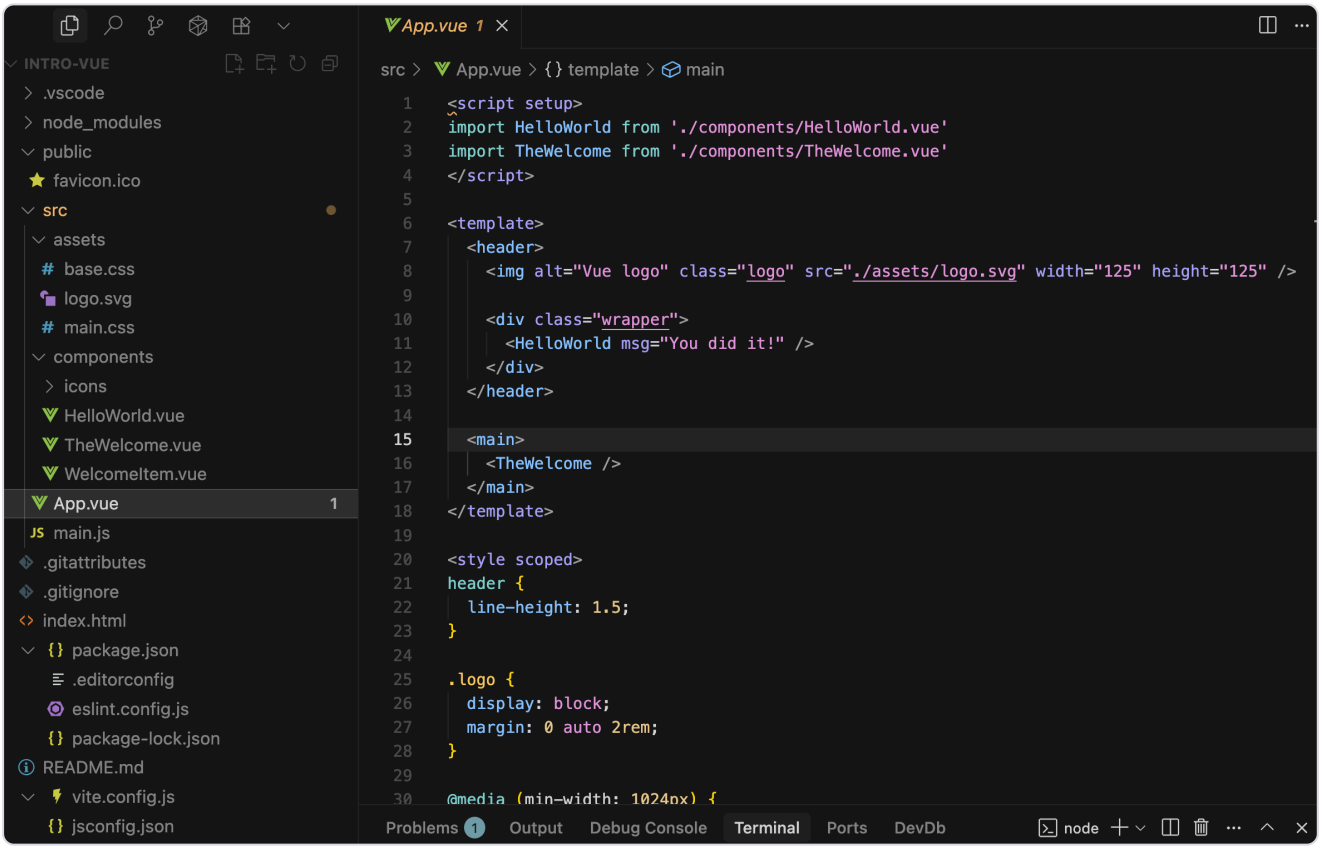
Slides

Deck 6 — Structure d'un projet Vue (PDF) PDF

Objectif : faire le tour du propriétaire d'un projet fraîchement généré, savoir à quoi sert chaque fichier, et comprendre le chemin que suit le navigateur pour afficher ta page.

Tu viens de lancer `npm create vue@latest`, puis `npm install`, et VS Code t'affiche une bonne vingtaine de fichiers. Rassure-toi : tu vas en toucher trois, et connaître les autres de loin suffit largement.

L'arborescence



Voici ce que génère `npm create vue@latest` en 2026, avec les options **Linter** (ESLint + Oxlint) et **Prettier** cochées (le dossier `node_modules/`, créé par `npm install`, n'est pas montré : il contient des milliers de fichiers que tu ne liras jamais) :

```
mon-portfolio/
├── .vscode/
│   ├── extensions.json
│   └── settings.json
├── public/
│   └── favicon.ico
├── src/
│   ├── assets/
│   │   ├── base.css
│   │   ├── logo.svg
│   │   └── main.css
│   ├── components/
│   │   ├── icons/
│   │   │   ├── IconCommunity.vue
│   │   │   ├── IconDocumentation.vue
│   │   │   ├── IconEcosystem.vue
│   │   │   ├── IconSupport.vue
│   │   │   └── IconTooling.vue
│   │   ├── HelloWorld.vue
│   │   ├── TheWelcome.vue
│   │   └── WelcomeItem.vue
│   ├── App.vue
│   └── main.js
├── .editorconfig
├── .gitattributes
├── .gitignore
├── .oxlintrc.json
├── .prettierrc.json
├── eslint.config.js
├── index.html
├── jsconfig.json
├── package.json
├── README.md
└── vite.config.js
```

NOTE

Les versions bougent vite dans cet écosystème. Si ton projet contient un fichier de plus ou de moins que la liste ci-dessus, ce n'est pas grave : les fichiers importants, eux, sont toujours là.

Les quatre fichiers que tu vas vraiment ouvrir

`index.html`, `src/main.js`, `src/App.vue` et `package.json`. On les regarde un par un.

`index.html` : la page unique

```xml
<!DOCTYPE html>
<html lang="">
  <head>
    <meta charset="UTF-8">
    <link rel="icon" href="/favicon.ico">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Vite App</title>
  </head>
  <body>
    <div id="app"></div>
    <script type="module" src="/src/main.js"></script>
  </body>
</html>
```

Deux lignes comptent :

- `<div id="app"></div>` est la **cible**. Vue va remplacer le contenu de ce div vide par toute ton application.
- `<script type="module" src="/src/main.js"></script>` est le **déclencheur** : il charge ton JavaScript, qui démarre Vue.

Le reste est du HTML classique. Tu ne reviendras dans ce fichier que pour changer le `<title>`, la langue (`lang="fr"`) et le favicon.

`src/main.js` : le démarrage

```javascript
import './assets/main.css'

import { createApp } from 'vue'
import App from './App.vue'

createApp(App).mount('#app')
```

Quatre lignes, quatre rôles :

- `import './assets/main.css'` charge les styles globaux de l'application.
- `import { createApp } from 'vue'` récupère la fonction qui fabrique une application Vue.
- `import App from './App.vue'` récupère ton composant racine.
- `createApp(App).mount('#app')` crée l'application à partir de `App.vue` et l'installe dans le `<div id="app">` de `index.html`.

En une phrase : « prends `App.vue`, transforme-le en application, et colle-le dans `#app` ».

`src/App.vue` : le composant racine

C'est le sommet de ton application : tout le reste est appelé depuis lui. Voici ce que génère Vue par défaut, en version raccourcie :

```XML
<script setup>
  import HelloWorld from './components/HelloWorld.vue'
  import TheWelcome from './components/TheWelcome.vue'
</script>

<template>
  <header>
    <img alt="Vue logo" class="logo" src="./assets/logo.svg" width="125" height="125"
/>

    <div class="wrapper">
      <HelloWorld msg="You did it!" />
    </div>
  </header>

  <main>
    <TheWelcome />
  </main>
</template>

<style scoped>
  header {
    line-height: 1.5;
  }

  .logo {
    display: block;
    margin: 0 auto 2rem;
  }
</style>
```

Un fichier `.vue` se lit en trois blocs :

- `<script setup>` : la logique du composant, et les imports des composants enfants.
- `<template>` : le HTML affiché. Les balises à majuscule (`<HelloWorld />` , `<TheWelcome />`) sont des composants, pas des balises HTML.
- `<style scoped>` : le CSS. Le mot `scoped` limite ce style à ce composant uniquement — il ne fuit pas dans le reste du site.

Ce contenu est une démonstration. Tu vas le vider dès l'exercice à la fin de ce chapitre.

`package.json` : la carte d'identité du projet

```json
{
  "name": "mon-portfolio",
  "version": "0.0.0",
  "private": true,
  "type": "module",
  "scripts": {
    "dev": "vite",
    "build": "vite build",
    "preview": "vite preview",
    "lint": "run-s \"lint:*\"",
    "lint:oxlint": "oxlint . --fix",
    "lint:eslint": "eslint . --fix --cache",
    "format": "prettier --write --experimental-cli src/"
  },
  "dependencies": {
    "vue": "^3.5.42"
  },
  "devDependencies": {
    "@eslint/js": "^10.0.1",
    "@vitejs/plugin-vue": "^6.0.8",
    "eslint": "^10.10.0",
    "eslint-config-prettier": "^10.1.8",
    "eslint-plugin-oxlint": "~1.82.0",
    "eslint-plugin-vue": "~10.11.0",
    "globals": "^17.12.0",
    "npm-run-all2": "^9.0.3",
    "oxlint": "~1.82.0",
    "prettier": "3.9.6",
    "vite": "^8.2.2",
    "vite-plugin-vue-devtools": "^8.2.1",
    "vue-eslint-parser": "^10.4.1"
  },
  "engines": {
    "node": "^22.18.0 || ⩾24.12.0"
  }
}
```

Les métadonnées, d'abord :

- `name` : le nom du projet, repris du dossier que tu as créé.
- `private: true` : empêche une publication accidentelle sur npm.
- `type: "module"` : le projet utilise les modules modernes (`import` / `export`).
- `engines` : les versions de Node.js compatibles. Si ton Node est plus ancien, npm t'avertira.

Les **scripts**, ensuite. Tu les lances avec `npm run <nom>` :

Commande	Ce qu'elle fait
`npm run dev`	Démarre le serveur de développement sur `http://localhost:5173`. C'est celle que tu tapes tous les jours.
`npm run build`	Fabrique la version optimisée du site dans un dossier `dist/`. C'est ce dossier qui part en ligne.
`npm run preview`	Sert le contenu de `dist/` en local, pour vérifier le résultat final avant mise en ligne.
`npm run lint`	Relit tout le code et corrige les problèmes qu'il sait corriger seul.
`npm run format`	Remet le code de `src/` en forme avec Prettier.

Enfin, les dépendances :

- **`dependencies`** : ce dont le site a besoin pour fonctionner une fois en ligne. Ici, une seule ligne : Vue.
- **`devDependencies`** : les outils qui ne servent que pendant que tu développes — Vite, ESLint, Prettier, le plugin Vue. Ils ne finissent pas dans le site publié.

> **ASTUCE**
>
> Le `^` devant un numéro de version signifie « cette version ou une plus récente, tant que le premier chiffre ne change pas ». `^3.5.42` accepte donc `3.6.0` mais jamais `4.0.0`.

Les fichiers de configuration

Tu ne les ouvriras presque jamais, mais autant savoir qui est qui.

`vite.config.js`

```javascript
import { fileURLToPath, URL } from 'node:url'

import { defineConfig } from 'vite'
import vue from '@vitejs/plugin-vue'
import vueDevTools from 'vite-plugin-vue-devtools'

// https://vite.dev/config/
export default defineConfig({
  plugins: [
    vue(),
    vueDevTools(),
  ],
  resolve: {
    alias: {
      '@': fileURLToPath(new URL('./src', import.meta.url)),
    },
  },
})
```

Vite est l'outil qui sert ton site pendant le développement et qui le compile pour la mise en ligne. Deux choses à retenir ici :

- `plugins` : le plugin `vue()` apprend à Vite à lire les fichiers `.vue` ; `vueDevTools()` ajoute un petit panneau de débogage flottant dans la page.
- `alias` : le raccourci `@` désigne le dossier `src/`. Tu peux donc écrire `import Bouton from '@/components/Bouton.vue'` depuis n'importe où, au lieu de compter les `../../`.

C'est le fichier `jsconfig.json` qui fait connaître ce même raccourci `@` à VS Code, pour que l'autocomplétion suive.

> **NOTE**
>
> C'est dans ce fichier que tu ajouteras le plugin Tailwind au chapitre suivant. Tu ajouteras deux lignes, tu ne supprimeras rien.

`eslint.config.js` et `.oxlintrc.json`

La configuration du linter : les règles qui soulignent les erreurs dans ton éditeur. Tu n'as rien à y faire pour l'instant.

`.prettierrc.json`

Les préférences de mise en forme du code :

```json
{
  "$schema": "https://json.schemastore.org/prettierrc",
  "semi": false,
  "singleQuote": true,
  "printWidth": 100
}
```

Pas de point-virgule en fin de ligne, des guillemets simples, des lignes de 100 caractères maximum. Prettier applique ça à chaque sauvegarde.

`.vscode/extensions.json`

```json
{
  "recommendations": [
    "Vue.volar",
    "dbaeumer.vscode-eslint",
    "EditorConfig.EditorConfig",
    "oxc.oxc-vscode",
    "esbenp.prettier-vscode"
  ]
}
```

Ce fichier est une liste d'extensions recommandées **pour ce projet**. Quand tu ouvres le dossier, VS Code affiche une notification « Ce dossier recommande des extensions » et te propose de les installer d'un clic. Pratique quand on travaille à plusieurs sur un même projet : tout le monde finit avec le même outillage. Le fichier voisin `.vscode/settings.json` fait la même chose pour les réglages (Format On Save y est déjà activé).

`public/` ou `src/assets/` : où mettre mes images ?

La question revient toujours. La règle est simple.

`src/assets/` : les fichiers que tu **importes** depuis un composant (une image de ta galerie, une police, un CSS). Vite les prend en charge : il les optimise, les compresse et renomme le fichier final. C'est le choix par défaut pour presque tout.

`public/` : les fichiers copiés **tels quels**, sans transformation, accessibles à la racine du site. Utilise ce dossier quand l'URL doit rester exactement celle que tu écris : `favicon.ico`, `robots.txt`, un PDF de CV à télécharger.

Un fichier `public/cv.pdf` sera servi à l'adresse `/cv.pdf`. Ce dossier est public au sens propre : n'y range rien de confidentiel.

Le flux d'exécution

Tu tapes `npm run dev`, et voici l'enchaînement :

1. Vite démarre un serveur local sur `http://localhost:5173`.
2. Le navigateur demande la page et reçoit `index.html`.
3. `index.html` contient un `<div id="app">` vide et charge `/src/main.js`.
4. `main.js` importe le CSS global, crée l'application avec `App.vue`, et la monte dans `#app`.
5. `App.vue` affiche son `<template>` et appelle ses composants enfants (`HelloWorld`, `TheWelcome`...).
6. Chaque composant produit son HTML, et Vue assemble le tout dans la page.

En une ligne : **`index.html` → `main.js` → `App.vue` → composants enfants → ta page**.

Et tant que le serveur tourne, chaque fois que tu sauvegardes un fichier, Vite recharge instantanément la partie modifiée dans le navigateur. Tu n'as jamais besoin de rafraîchir à la main.

Exercice : fais le ménage

La page de démonstration de Vue est jolie, mais ce n'est pas ton portfolio. On repart d'une page blanche. Cet état de départ sera le point de départ des chapitres sur Tailwind : fais-le sérieusement.

À faire :

1. Supprime les composants de démo : `src/components/HelloWorld.vue`, `TheWelcome.vue`, `WelcomeItem.vue` et tout le dossier `src/components/icons/`.
2. Supprime aussi `src/assets/logo.svg`.
3. Vide `src/App.vue` et remplace son contenu par un simple `<h1>Mon portfolio</h1>`.
4. Vide `src/assets/main.css` et `src/assets/base.css` (garde les fichiers, mais laisse-les vides ou avec deux ou trois règles de base).
5. Dans `index.html`, passe `lang="fr"` et change le `<title>` en « Mon portfolio ».
6. Lance `npm run dev` : la page doit afficher un `h1` « Mon portfolio », rien d'autre, et la console du navigateur doit être vide d'erreurs.

src/App.vue :

```xml
<script setup>
  // Rien pour l'instant : on ajoutera nos composants ici.
</script>

<template>
  <h1>Mon portfolio</h1>
</template>

<style scoped>
  h1 {
    text-align: center;
  }
</style>
```

src/assets/base.css — quelques règles minimales suffisent :

```css
*,
*::before,
*::after {
  box-sizing: border-box;
  margin: 0;
  padding: 0;
}

body {
  font-family: system-ui, sans-serif;
  line-height: 1.6;
}
```

src/assets/main.css — on garde uniquement l'import de base.css :

```css
@import './base.css';
```

index.html :

```xml
<!DOCTYPE html>
<html lang="fr">
  <head>
    <meta charset="UTF-8">
    <link rel="icon" href="/favicon.ico">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Mon portfolio</title>
  </head>
  <body>
    <div id="app"></div>
    <script type="module" src="/src/main.js"></script>
  </body>
</html>
```

`src/main.js` ne change pas : il importe toujours `./assets/main.css`, qui existe toujours.

En cas d'erreur : si la page affiche « Failed to resolve import ./components/HelloWorld.vue », c'est qu'un import vers un composant supprimé traîne encore dans `App.vue`. Relis ton `<script setup>`.

En résumé

- `index.html` : la page unique, avec `<div id="app">` comme cible.
- `src/main.js` : crée l'application Vue et la monte dans `#app`.
- `src/App.vue` : le composant racine, sommet de l'arbre.
- `src/components/` : tes composants réutilisables.
- `src/assets/` : ce qui est importé et optimisé ; `public/` : ce qui est copié tel quel.
- `package.json` : les scripts (`dev`, `build`, `preview`, `lint`, `format`) et les dépendances.
- Flux : `index.html` → `main.js` → `App.vue` → composants.

Le projet est vide et propre. On peut commencer à le remplir.

3.6 Tailwind CSS — les bases

Slides

Deck 7 — Tailwind CSS (PDF) PDF

Objectif : comprendre ce qu'est une classe utilitaire, installer Tailwind CSS v4 dans ton projet Vue, et savoir styler un bouton, un champ, une mise en page et un texte sans écrire une seule ligne de CSS.

1. C'est quoi, « utility-first » ?

Tailwind CSS est un framework CSS **utility-first** : au lieu d'écrire des règles CSS et d'inventer des noms de classes, tu assembles des petites classes qui font **une chose chacune**. `px-4`, c'est du padding horizontal. `rounded-lg`, c'est des coins arrondis. Tu composes.

En CSS classique, tu écris d'abord la règle :

```css
.bouton-primaire {
  background-color: #1c1a17;
  color: #f4efe6;
  padding: 10px 20px;
  border-radius: 6px;
}
```

Puis tu l'utilises :

```xml
<button class="bouton-primaire">Clique-moi</button>
```

Avec Tailwind, tout tient dans le balisage :

```xml
<button class="bg-zinc-900 text-stone-50 px-5 py-2.5 rounded-md">
  Clique-moi
</button>
```

Pas de fichier CSS à maintenir, pas de nom de classe à inventer, pas de « c'est quoi déjà, `.btn-primary-2-final` ? ». Le style est là où il s'applique.

> **NOTE**
>
> Ça paraît verbeux au début. C'est normal. Après deux séances, tu liras une ligne de classes plus vite que tu ne retrouverais la règle CSS correspondante dans un fichier de 800 lignes.

Ce que tu vas apprendre dans ce chapitre :

- installer Tailwind v4 dans ton projet Vue ;
- styler les briques de base : boutons, champs, mises en page, typographie, couleurs ;

- rendre tout ça responsive et accessible.

2. Installation dans le projet Vue

On installe Tailwind **v4**. C'est important : la v4 n'a plus de fichier `tailwind.config.js` et beaucoup de tutos que tu trouveras en ligne parlent encore de la v3. Si un tuto te demande de créer un `tailwind.config.js` ou de toucher à PostCSS, c'est de la v3, passe ton chemin.

Tu pars de ton projet Vue déjà créé et nettoyé (`App.vue` ne contient plus qu'un `<h1>`).

Étape 1 — Installer les paquets

Dans le terminal, à la racine de ton projet :

```bash
npm install tailwindcss @tailwindcss/vite
```

Étape 2 — Brancher le plugin dans Vite

Ouvre `vite.config.js` . Ajoute l'import en haut et `tailwindcss()` dans le tableau `plugins` — `vue()` et `vueDevTools()` y sont déjà.

```javascript
import { fileURLToPath, URL } from 'node:url'

import { defineConfig } from 'vite'
import vue from '@vitejs/plugin-vue'
import vueDevTools from 'vite-plugin-vue-devtools'
import tailwindcss from '@tailwindcss/vite'

export default defineConfig({
  plugins: [
    vue(),
    vueDevTools(),
    tailwindcss(),
  ],
  resolve: {
    alias: {
      '@': fileURLToPath(new URL('./src', import.meta.url)),
    },
  },
})
```

Tu n'ajoutes que **deux lignes** : l'import de `tailwindcss` et l'appel `tailwindcss()` dans `plugins` . Ne supprime rien d'autre : `vueDevTools()` et le raccourci `@` du bloc `resolve` étaient déjà là.

Étape 3 — Importer Tailwind dans le CSS

Ouvre `src/assets/main.css` et remplace tout son contenu par cette ligne :

```css
@import "tailwindcss";
```

C'est tout. Une ligne. Vérifie juste que `src/main.js` importe bien ce fichier (`import './assets/main.css'`) — c'est le cas par défaut.

Étape 4 — Relancer le serveur

Le plugin Vite vient d'être ajouté : il faut **redémarrer** le serveur pour qu'il soit pris en compte. Dans le terminal, `Ctrl + C` pour arrêter (oui, `Ctrl` aussi sur Mac, pas `Cmd`), puis :

```bash
npm run dev
```

Étape 5 — Tester que ça marche

Ouvre `src/App.vue` et remplace son contenu par ceci :

```xml
<script setup>
  // Rien ici pour l'instant.
</script>

<template>
  <main class="min-h-screen bg-stone-100 flex items-center justify-center p-8">
    <div class="max-w-md border-2 border-zinc-900 bg-white p-8">
      <p class="text-xs uppercase tracking-widest text-zinc-500">Test</p>
      <h1 class="mt-2 text-4xl font-bold text-zinc-900">
        Tailwind est branché.
      </h1>
      <p class="mt-4 text-zinc-600">
        Fond crème, cadre noir, texte net. Si tu vois ça, on peut travailler.
      </p>
    </div>
  </main>
</template>
```



Sauvegarde, retourne dans le navigateur. Tu dois voir un cadre noir centré sur un fond crème. Si tu vois du texte brut non stylé, c'est que Tailwind n'est pas branché : relis les étapes 2 et 3, et regarde les erreurs dans le terminal.

> **ASTUCE**
>
> **VS Code** : vérifie que l'extension *Tailwind CSS IntelliSense* s'active. Place ton curseur dans un attribut `class`, tape `bg-` et regarde : une liste de classes doit apparaître, avec un carré de couleur en face de chacune. Si rien ne s'affiche, ferme et rouvre le dossier du projet dans VS Code. Cette autocomplétion, c'est ton meilleur outil d'apprentissage : elle te montre ce qui existe.

3. Les bases

3.1 Boutons

Un bouton, c'est une couleur de fond, du padding, un texte lisible et une réaction au survol. Trois directions très différentes, avec les mêmes briques :

```xml
<button class="bg-zinc-900 text-stone-50 px-5 py-2.5 rounded-md hover:bg-zinc-700 transition">
  Sobre
</button>
```

```xml
<button class="border-4 border-zinc-900 bg-amber-300 text-zinc-900 px-6 py-3 uppercase font-extrabold tracking-wide shadow-[4px_4px_0_0_#18181b] hover:-translate-x-0.5 hover:-translate-y-0.5 hover:shadow-[6px_6px_0_0_#18181b] transition">
  Brutaliste
</button>
```

```xml
<button class="border border-zinc-300 bg-white text-zinc-800 px-6 py-3 font-serif
text-lg hover:border-zinc-900 transition">
  Éditorial
</button>
```



NOTE

L'ombre décalée du bouton brutaliste n'existe pas comme classe toute faite. On l'écrit en **valeur arbitraire**, entre crochets : `shadow-[4px_4px_0_0_#18181b]`. Les espaces de la valeur CSS deviennent des `_`. C'est la porte de sortie de Tailwind quand une classe manque — à utiliser avec parcimonie.

Classe	Utilité	Exemple
`bg-{couleur}`	Couleur de fond	`bg-zinc-900`
`text-{couleur}`	Couleur du texte	`text-stone-50`
`px-{n}`, `py-{n}`	Padding horizontal / vertical	`px-4 py-2`
`rounded`	Coins arrondis	`rounded-md`, `rounded-full`, `rounded-none`
`border`	Bordure	`border-2 border-zinc-900`
`hover:`	Style au survol	`hover:bg-zinc-700`
`shadow`	Ombre portée	`shadow-sm`, `shadow-lg`
`transition`	Animer le changement	`transition`

ATTENTION

En Tailwind v4, l'échelle des ombres a bougé : l'ancien `shadow-sm` de la v3 s'appelle maintenant `shadow-xs`, et l'ancien `shadow` tout court s'appelle `shadow-sm`. Si un exemple trouvé en ligne te donne une ombre plus forte que prévu, c'est souvent ça.

3.2 Champs de formulaire

```xml
<input
  class="border border-zinc-300 px-3 py-2 rounded-md focus-visible:outline-2 focus-visible:outline-offset-2 focus-visible:outline-zinc-900"
  placeholder="Nom complet"
>
```

```xml
<input
  class="border-4 border-zinc-900 bg-white px-4 py-3 text-lg font-bold placeholder:text-zinc-400 focus-visible:bg-amber-100 focus-visible:outline-2 focus-visible:outline-offset-2 focus-visible:outline-zinc-900"
  placeholder="TON NOM"
>
```



Classe	Utilité	Exemple
`border`	Bordure	`border border-zinc-300`
`rounded`	Coins arrondis	`rounded-md`
`px-{n}`, `py-{n}`	Padding intérieur	`px-4 py-2`
`placeholder:`	Style du texte d'exemple	`placeholder:text-zinc-400`
`focus-visible:`	Apparence au focus clavier	`focus-visible:outline-2`
`w-full`	Largeur totale du parent	`w-full`

3.3 Mise en page : Flexbox et Grid

Centrer un bloc avec Flexbox :

```xml
<div class="flex items-center justify-center h-48 bg-stone-100">
  <div class="bg-zinc-900 text-stone-50 p-4">Je suis centré.</div>
</div>
```

Une grille de trois colonnes égales :

```xml
<div class="grid grid-cols-3 gap-4">
  <div class="bg-emerald-200 p-4">Bloc 1</div>
  <div class="bg-emerald-300 p-4">Bloc 2</div>
  <div class="bg-emerald-400 p-4">Bloc 3</div>
</div>
```



Classe	Utilité	Exemple
`flex`	Conteneur Flexbox	`flex`
`items-center`	Alignement vertical (en flex-row)	`items-center`
`justify-between`	Répartition horizontale	`justify-center`, `justify-between`
`flex-col`	Empiler verticalement	`flex-col`
`grid`, `grid-cols-{n}`	Créer une grille	`grid grid-cols-3`
`gap-{n}`	Espace entre les éléments	`gap-4` (1 rem)

3.4 Typographie

```xml
<h1 class="text-4xl font-bold tracking-tight text-zinc-900">Titre principal</h1>
<p class="mt-3 text-zinc-600 leading-relaxed">Un paragraphe confortable à lire.</p>
```

```xml
<p class="font-mono text-sm uppercase tracking-widest text-zinc-500">Étiquette</p>
```



Classe	Utilité	Exemple
`text-{taille}`	Taille du texte	`text-sm`, `text-xl`, `text-4xl`
`font-{poids}`	Graisse	`font-medium`, `font-bold`, `font-black`
`font-{famille}`	Famille	`font-sans`, `font-serif`, `font-mono`
`leading-{n}`	Hauteur de ligne	`leading-relaxed`, `leading-tight`
`tracking-{n}`	Interlettrage	`tracking-tight`, `tracking-widest`
`uppercase`	Tout en majuscules	`uppercase`

3.5 Couleurs et arrière-plans

```xml
<div class="bg-red-600 text-white p-4">Fond rouge, texte blanc</div>
```

```xml
<div class="bg-zinc-900 text-amber-300 p-4">Contraste franc</div>
```

```xml
<div class="bg-linear-to-b from-stone-100 to-stone-300 text-zinc-900 p-4">Dégradé discret</div>
```



Classe	Utilité	Exemple
`bg-{couleur}`	Couleur de fond	`bg-amber-300`

Classe	Utilité	Exemple
`text-{couleur}`	Couleur du texte	`text-zinc-900`
`bg-linear-to-{dir}`	Dégradé linéaire	`bg-linear-to-r`, `bg-linear-to-b`
`from-`, `to-`	Bornes du dégradé	`from-stone-100 to-stone-300`

ATTENTION

En Tailwind v4, le dégradé s'écrit `bg-linear-to-r`, pas `bg-gradient-to-r` (v3). Si tu copies un exemple qui ne produit aucun dégradé, vérifie d'abord ça.

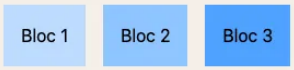
Et rappelle-toi le chapitre sur le slop : un dégradé violet-rose sur fond sombre, ce n'est pas un style, c'est le réglage par défaut de tout le monde. Un aplat franc dit plus de choses.

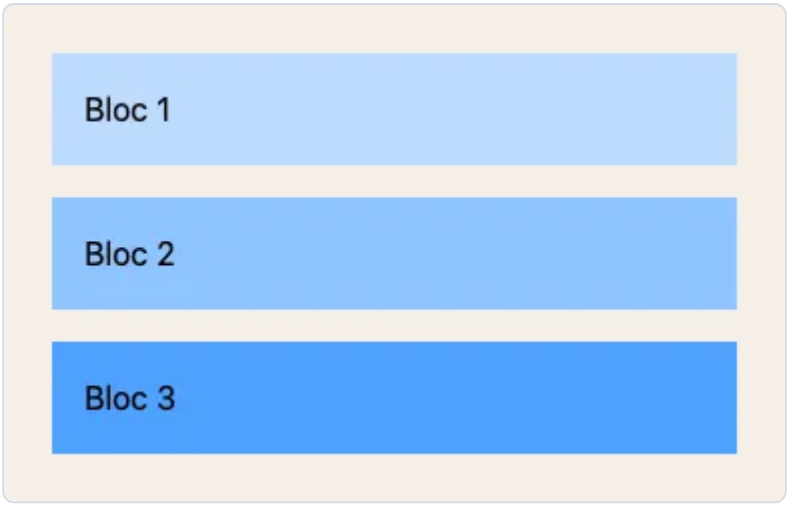
4. Responsive et variantes d'état

4.1 Mobile first

Une classe sans préfixe s'applique **partout**. Un préfixe comme `md:` s'applique **à partir de** cette largeur. Donc : tu écris d'abord le style mobile, puis tu corriges pour les grands écrans.

```xml
<div class="flex flex-col md:flex-row gap-4">
  <div class="bg-blue-200 p-4">Bloc 1</div>
  <div class="bg-blue-300 p-4">Bloc 2</div>
  <div class="bg-blue-400 p-4">Bloc 3</div>
</div>
```





Empilés sur mobile grâce à `flex-col`, côte à côte dès 768 px grâce à `md:flex-row`.

Préfixe	Écrans	Largeur minimale
aucun	mobile	0 px
`sm:`	grands mobiles	640 px
`md:`	tablettes	768 px
`lg:`	ordinateurs	1024 px
`xl:`	grands écrans	1280 px
`2xl:`	très grands écrans	1536 px

```XML
<p class="text-base md:text-lg lg:text-xl">
  Texte qui grossit avec l'écran.
</p>
```

4.2 Variantes d'état

Même principe : un préfixe devant la classe.

```XML
<button class="bg-emerald-600 text-white px-5 py-2.5 rounded-md hover:bg-emerald-700
active:scale-95 focus-visible:outline-2 focus-visible:outline-offset-2 focus-
visible:outline-emerald-700 transition">
  Envoyer
</button>
```

```xml
<button class="bg-zinc-200 text-zinc-500 px-5 py-2.5 rounded-md disabled:cursor-not-allowed disabled:opacity-60" disabled>
  Indisponible
</button>
```



Variante	Utilité	Exemple
`hover:`	Au survol de la souris	`hover:bg-zinc-700`
`focus:`	Au focus (clic ou clavier)	`focus:border-zinc-900`
`focus-visible:`	Au focus **clavier** uniquement	`focus-visible:outline-2`
`active:`	Pendant le clic	`active:scale-95`
`disabled:`	Élément désactivé	`disabled:opacity-60`

ATTENTION

Accessibilité : ne supprime jamais le focus sans le remplacer.

Beaucoup d'exemples trouvés en ligne mettent `outline-none` (ou `outline-hidden` en v4) sur les boutons et les champs « parce que le contour bleu du navigateur est moche ». Résultat : une personne qui navigue au clavier ne sait plus où elle est dans la page. C'est éliminatoire.

Deux règles, valables pour tous tes exercices :

1. **Focus visible obligatoire.** Si tu enlèves le contour par défaut, tu en remets un : `focus-visible:ring-2 focus-visible:ring-zinc-900` ou `focus-visible:outline-2 focus-visible:outline-offset-2 focus-visible:outline-zinc-900`. Teste avec la touche `[TAB]` : tu dois voir où tu es, à chaque étape.
2. **Contraste suffisant.** Un texte doit ressortir nettement sur son fond (ratio 4,5:1 minimum pour du texte courant). `text-zinc-400` sur `bg-white`, c'est joli sur ton écran neuf et illisible partout ailleurs. Dans le doute, descends d'un ou deux crans : `text-zinc-600`, `text-zinc-700`.

Attention aussi : en v4, `ring` tout seul fait **1 px** (contre 3 px en v3). Écris `ring-2` si tu veux un anneau franc.

4.3 Combiner les deux

Ça se cumule, dans l'ordre responsive puis état :

```xml
<button class="bg-red-600 text-white px-4 py-2 md:hover:bg-red-800 transition">
  Survol renforcé sur grand écran
</button>
```

Récap

- Tailwind = des classes qui font une chose chacune, assemblées dans le `class`.
- Installation v4 : `npm install tailwindcss @tailwindcss/vite`, plugin dans `vite.config.js`, `@import "tailwindcss";` dans `main.css`, puis on relance `npm run dev`. Pas de `tailwind.config.js`.
- Les briques : `bg-`, `text-`, `p-` / `m-`, `border`, `rounded`, `flex` / `grid`, `gap-`.
- Mobile first : sans préfixe d'abord, `md:` / `lg:` ensuite.
- États : `hover:`, `focus-visible:`, `active:`, `disabled:`. Le focus visible n'est pas négociable.
- Pièges v4 : `bg-linear-to-r` (pas `bg-gradient-to-r`), échelle des ombres décalée, `ring` à 1 px.

Sources

- Documentation officielle Tailwind CSS v4 — la barre de recherche en haut est le chemin le plus court.
- Guide d'installation avec Vite

3.7 Premiers pas avec Tailwind

Objectif : mettre les mains dedans. Cinq exercices qui partent du code tout fait et te lâchent progressivement la main, jusqu'à ce que tu conçoives tes propres composants. Deux d'entre eux finiront dans ton portfolio.

Comment lire ce chapitre

Chaque exercice est découpé en **paliers**. Tu ne fais pas tous les paliers de tous les exercices au hasard : tu suis l'ordre.

Palier	Ce que tu fais	Ce que tu y gagnes
1 — Copie et observe	Tu colles le code donné, puis tu modifies une ou deux classes imposées et tu regardes le résultat	Tu vois ce que fait chaque classe, sans réfléchir à la mise en page
2 — Complète les trous	Le code est là, avec des `???` à remplacer. Chaque trou est précédé d'un commentaire qui dit ce qu'on attend et propose trois candidates	Tu choisis, donc tu décides. Plusieurs réponses sont justes
3 — À toi	Une consigne, des critères, une banque de classes. Pas de code	Tu conçois

Au palier 2, **plusieurs réponses sont correctes**. Les trois candidates proposées fonctionnent toutes techniquement : la question est de savoir laquelle sert ton intention. C'est exactement le travail du reste de l'année.

> **ASTUCE**
>
> Garde l'autocomplétion de *Tailwind CSS IntelliSense* ouverte pendant que tu remplis les trous : tape les premières lettres d'une candidate, VS Code te montre la couleur et la valeur CSS derrière. C'est plus rapide que la doc.

Setup

Ouvre `src/App.vue` et remplace tout son contenu par ceci. Toutes les sections des exercices viendront se poser à l'intérieur du `<main>`, les unes sous les autres.

```xml
<script setup>
  // Rien ici : on ne fait que du template et des classes.
</script>

<template>
  <main class="min-h-screen bg-stone-100 p-8">
    <h1 class="text-4xl font-bold tracking-tight text-zinc-900 mb-10">
      Mes exercices Tailwind
    </h1>

    <!-- Tes sections d'exercices viennent ici -->

  </main>
</template>
```

Lance le serveur si ce n'est pas déjà fait :

```bash
npm run dev
```

Et ouvre `http://localhost:5173`.

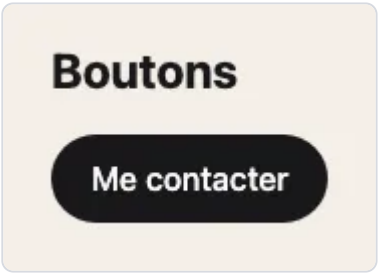
Exercice 1 — Boutons

Durée indicative : 45 minutes.

Palier 1 — Copie et observe

Colle cette section dans ton `<main>` :

```xml
<section class="mb-12">
  <h2 class="text-2xl font-bold text-zinc-900 mb-4">Boutons</h2>
  <div class="flex flex-wrap gap-4">
    <button class="bg-zinc-900 text-stone-50 px-5 py-2.5 rounded-full font-medium
hover:bg-zinc-700 focus-visible:outline-2 focus-visible:outline-offset-2 focus-
visible:outline-zinc-900 transition">
      Me contacter
    </button>
  </div>
</section>
```



Maintenant, trois manipulations. Fais-les **une par une**, regarde le navigateur entre chaque, puis remets la classe d'origine avant de passer à la suivante.

1. Remplace `bg-zinc-900` par `bg-amber-300`. Le bouton devient jaune… et le texte `text-stone-50` devient presque illisible. C'est un problème de **contraste**. Corrige en mettant aussi `text-zinc-900`.
2. Supprime `rounded-full`. Le bouton devient un rectangle à angles droits. Essaie ensuite `rounded-md`, puis `rounded-none`. Trois personnalités très différentes pour une seule classe.
3. Supprime `transition`. Survole le bouton : le changement de couleur devient brutal, instantané. Remets-la.

> **NOTE**
>
> Prends l'habitude de cette boucle : je change **une** classe, je regarde, je comprends, je passe à la suivante. Changer cinq classes d'un coup et se demander laquelle a fait quoi, c'est comme ça qu'on perd une heure.

Palier 2 — Complète les trous

Colle ce deuxième bouton dans le même `<div class="flex flex-wrap gap-4">`. Remplace chaque `???` par une classe. Le commentaire juste au-dessus te dit ce qu'on attend et te propose trois candidates : choisis-en une.

```xml
<button class="
  ??? ??? ??? ??? ??? ??? ??? ???
">
  Voir mes projets
</button>
```

Voici les huit trous, dans l'ordre :

```xml
<!-- ??? couleur de fond : bg-amber-300 | bg-emerald-400 | bg-orange-500 -->
<!-- ??? couleur du texte, qui doit contraster avec le fond choisi : text-zinc-900 | text-white | text-stone-50 -->
<!-- ??? padding intérieur, généreux : px-6 py-3 | px-8 py-4 | px-4 py-2 -->
<!-- ??? bordure épaisse et noire : border-2 border-zinc-900 | border-4 border-zinc-900 | border-8 border-zinc-900 -->
<!-- ??? arrondi : rounded-none | rounded-sm | rounded-md -->
<!-- ??? ombre dure et décalée : shadow-[4px_4px_0_0_#18181b] | shadow-[6px_6px_0_0_#18181b] | shadow-lg -->
<!-- ??? effet au survol : hover:-translate-y-0.5 | hover:bg-zinc-900 | hover:rotate-1 -->
<!-- ??? focus clavier visible : focus-visible:outline-2 focus-visible:outline-offset-2 focus-visible:outline-zinc-900 | focus-visible:ring-2 focus-visible:ring-zinc-900 | focus-visible:ring-4 focus-visible:ring-amber-300 -->
```

Objectif : un bouton **brutaliste** — couleur franche, bordure noire épaisse, angles droits ou presque, ombre dure. Pas de flou, pas de dégradé.

Teste avec la touche `[TAB]` : ton bouton doit montrer clairement qu'il a le focus.

Une combinaison parmi d'autres :

```xml
<button class="bg-amber-300 text-zinc-900 px-6 py-3 border-4 border-zinc-900
rounded-none shadow-[4px_4px_0_0_#18181b] font-extrabold uppercase tracking-wide
hover:-translate-y-0.5 focus-visible:outline-2 focus-visible:outline-offset-2
focus-visible:outline-zinc-900 transition">
  Voir mes projets
</button>
```



Une autre, tout aussi valable, plus sourde :

```xml
<button class="bg-emerald-400 text-zinc-900 px-8 py-4 border-2 border-zinc-900
rounded-sm shadow-[6px_6px_0_0_#18181b] font-bold hover:bg-zinc-900 hover:text-
emerald-400 focus-visible:ring-2 focus-visible:ring-zinc-900 transition">
  Voir mes projets
</button>
```

Palier 3 — À toi

Consigne : ajoute **trois boutons** de plus, chacun dans une direction esthétique différente, choisie dans cette liste :

- **éditorial** : serif, fin, beaucoup d'air, peu de couleur ;
- **riso / imprimé** : une couleur franche, une bordure épaisse, aucun effet de profondeur ;
- **minimal japonais** : blanc cassé, trait fin, une seule respiration ;
- **brutaliste** : contrastes violents, majuscules, ombre dure ;
- **ticket de caisse** : monospace, majuscules, bordure pointillée.

Trois directions **différentes** : pas trois variantes de la même. Le but est que le prof puisse deviner ta direction sans que tu la nommes.

> **ATTENTION**
>
> **Interdits, et ce n'est pas négociable** : néon / cyberpunk, glassmorphism (`backdrop-blur` + fond blanc translucide), dégradés violet-rose. C'est le slop du chapitre d'intro : le réglage par défaut de tous les générateurs, donc l'absence de décision. Tu peux le faire pour rire chez toi, pas ici.

Banque de classes — pioche, combine, cherche le reste dans la doc.

Catégorie	Classes
Fond	`bg-white`, `bg-stone-100`, `bg-zinc-900`, `bg-amber-300`, `bg-red-600`, `bg-emerald-500`, `bg-blue-700`, `bg-transparent`
Texte	`text-zinc-900`, `text-zinc-600`, `text-white`, `text-stone-50`, `font-serif`, `font-mono`, `font-light`, `font-bold`, `font-extrabold`, `uppercase`, `tracking-tight`, `tracking-widest`, `text-sm`, `text-lg`
Espacement	`px-3 py-1.5`, `px-5 py-2.5`, `px-8 py-4`, `gap-4`
Bordure	`border`, `border-2`, `border-4`, `border-zinc-900`, `border-zinc-300`, `border-dashed`, `border-dotted`, `rounded-none`, `rounded-md`, `rounded-full`
États	`hover:bg-zinc-700`, `hover:text-white`, `hover:-translate-y-0.5`, `active:scale-95`, `focus-visible:outline-2 focus-visible:outline-offset-2`, `focus-visible:ring-2`, `transition`
Ombre	`shadow-xs`, `shadow-sm`, `shadow-lg`, `shadow-none`, `shadow-[4px_4px_0_0_#18181b]`

Critères observables

- [ ] Trois boutons, trois directions esthétiques nettement distinctes
- [ ] Chacun a un `hover:` visible
- [ ] Chacun a un focus clavier visible, testé au `[TAB]`
- [ ] Le texte contraste franchement avec son fond
- [ ] Aucun néon, aucun glassmorphism, aucun dégradé violet-rose
- [ ] Pas de classes inutiles empilées « au cas où »

Montre au prof cette checklist cochée et tes cinq boutons à l'écran.

Exercice 2 — Carte de profil

Durée indicative : 45 minutes.

C'est la future carte « à propos » de ton portfolio. Autant la faire te ressembler tout de suite.

Palier 2 — Complète les trous

Colle cette section. La structure est donnée, les décisions de style sont à toi.

```xml
<section class="mb-12">
  <h2 class="text-2xl font-bold text-zinc-900 mb-4">Carte de profil</h2>

  <!-- ??? ombre de la carte au repos : shadow-xs | shadow-sm | shadow-lg -->
  <!-- ??? effet au survol de toute la carte : hover:shadow-xl | hover:-translate-y-1 | hover:border-zinc-900 -->
  <article class="max-w-sm bg-white border border-zinc-200 overflow-hidden ??? ??? transition">

    <!-- ??? hauteur de l'image : h-40 | h-48 | h-64 -->
    <!-- ??? cadrage de l'image, pour qu'elle ne soit pas déformée : object-cover | object-contain | object-fill -->
    <img
      src="https://i.pravatar.cc/600"
      alt="Portrait"
      class="w-full ??? ???"
    >

    <!-- ??? padding intérieur du bloc texte : p-4 | p-6 | p-8 -->
    <div class="???">

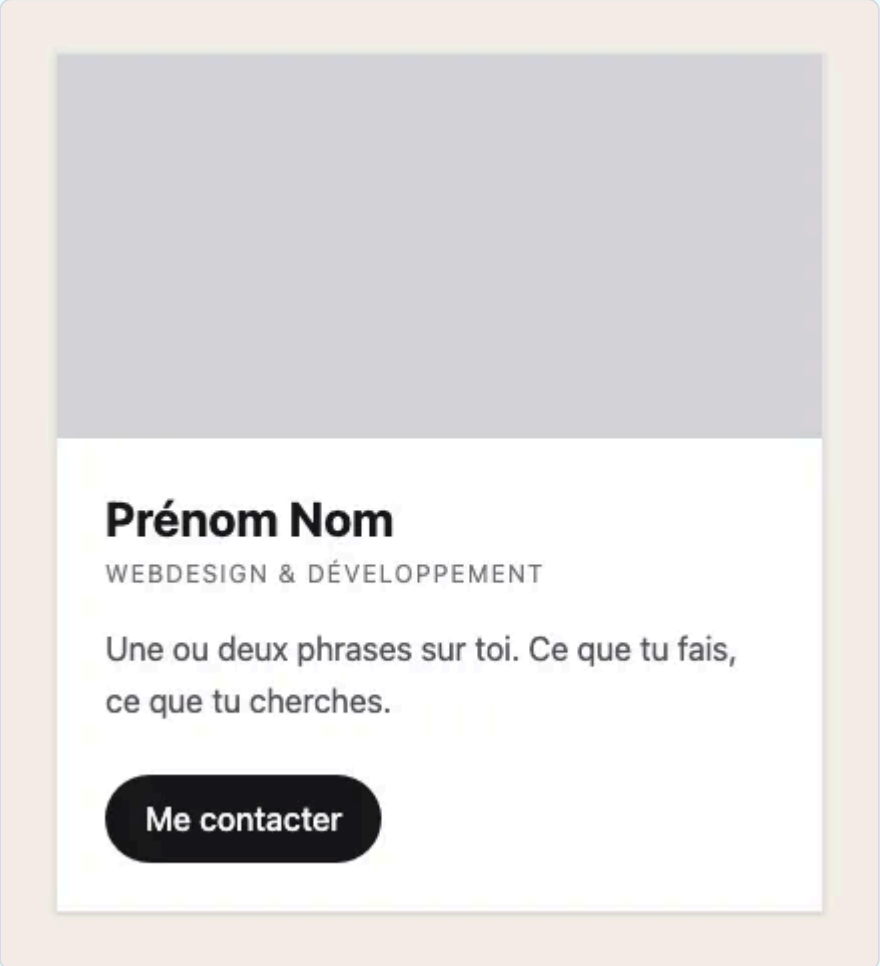
      <!-- ??? taille et graisse du nom : text-xl font-bold | text-2xl font-bold | text-2xl font-serif -->
      <h3 class="text-zinc-900 ???">Prénom Nom</h3>

      <!-- ??? style du sous-titre, plus discret que le nom : text-sm text-zinc-500 | text-base text-zinc-600 | text-xs uppercase tracking-widest text-zinc-500 -->
      <p class="mt-1 ???">Webdesign &amp;amp; développement</p>

      <!-- ??? espace au-dessus de la bio : mt-2 | mt-4 | mt-6 -->
      <p class="??? text-zinc-600 leading-relaxed">
        Une ou deux phrases sur toi. Ce que tu fais, ce que tu cherches.
      </p>

      <!-- ??? bouton : reprends les classes d'un de tes boutons de l'exercice 1 -->
      <button class="mt-6 ???">
        Me contacter
      </button>
    </div>
  </article>
</section>
```

```xml
<article class="max-w-sm bg-white border border-zinc-200 overflow-hidden shadow-sm hover:shadow-xl transition">
  <img
    src="https://i.pravatar.cc/600"
    alt="Portrait"
    class="w-full h-48 object-cover"
  >
  <div class="p-6">
    <h3 class="text-zinc-900 text-2xl font-bold">Prénom Nom</h3>
    <p class="mt-1 text-xs uppercase tracking-widest text-zinc-500">Webdesign &amp;amp; développement</p>
    <p class="mt-4 text-zinc-600 leading-relaxed">
      Une ou deux phrases sur toi. Ce que tu fais, ce que tu cherches.
    </p>
    <button class="mt-6 bg-zinc-900 text-stone-50 px-5 py-2.5 rounded-full font-medium hover:bg-zinc-700 focus-visible:outline-2 focus-visible:outline-offset-2 focus-visible:outline-zinc-900 transition">
      Me contacter
    </button>
  </div>
</article>
```



`object-cover` recadre l'image pour remplir la zone sans la déformer ; `object-contain` laisse des bandes vides ; `object-fill` l'écrase. Pour un portrait, c'est `object-cover`.

Palier 3 — À toi

Consigne : refais cette carte dans **ta** direction — celle que tu as choisie à l'exercice 1. Même contenu, autre parti pris. Tu peux déplacer l'image, la supprimer, passer en deux colonnes, mettre le nom en énorme, ne garder qu'un trait.

Remplace le texte d'exemple par **le tien** : ton vrai prénom, ta vraie phrase. C'est la carte qui ira sur ton portfolio.

Banque de classes

Catégorie	Classes
Fond	`bg-white`, `bg-stone-100`, `bg-amber-100`, `bg-zinc-900`
Texte	`font-serif`, `font-mono`, `text-sm`, `text-2xl`, `text-4xl`, `font-light`, `font-black`, `leading-relaxed`, `tracking-tight`
Espacement	`p-4`, `p-8`, `mt-2`, `mt-6`, `space-y-3`, `max-w-sm`, `max-w-md`
Bordure	`border`, `border-2`, `border-zinc-900`, `border-b`, `rounded-none`, `rounded-lg`, `overflow-hidden`
Image	`w-full`, `h-40`, `h-64`, `object-cover`, `aspect-square`, `rounded-full`, `grayscale`
États	`hover:shadow-xl`, `hover:-translate-y-1`, `transition`, `focus-visible:ring-2`
Responsive	`md:flex`, `md:w-1/2`, `md:text-3xl`, `md:p-10`

Critères observables

- [ ] Le contenu est le mien, pas le texte d'exemple
- [ ] La hiérarchie est claire : on lit le nom en premier, la bio en dernier
- [ ] L'image n'est pas déformée
- [ ] La carte réagit au survol
- [ ] Le bouton a un focus clavier visible
- [ ] La direction esthétique est la même qu'à l'exercice 1

Montre au prof la carte et la checklist.

Exercice 3 — Navbar responsive

Durée indicative : 30 minutes.

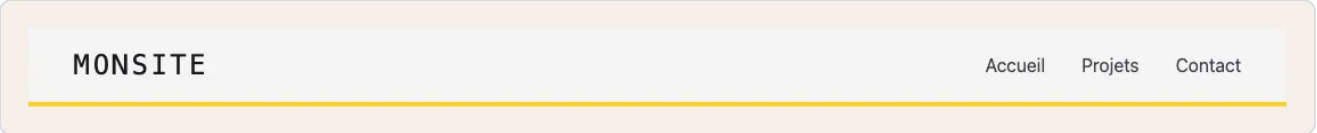
Palier 1 — Copie et observe

Cette navbar se place **avant** le `<h1>`, tout en haut du `<main>` — ou mieux, juste au-dessus, collée en haut de page.

```xml
<nav class="bg-white border-b-2 border-zinc-900 px-6 py-4 mb-10">
  <div class="mx-auto max-w-5xl flex items-center justify-between">
    <span class="text-xl font-bold tracking-tight text-zinc-900">MonSite</span>

    <div class="hidden md:flex gap-8">
      <a href="#" class="text-zinc-700 hover:text-zinc-900 transition">Accueil</a>
      <a href="#" class="text-zinc-700 hover:text-zinc-900 transition">Projets</a>
      <a href="#" class="text-zinc-700 hover:text-zinc-900 transition">Contact</a>
    </div>

    <button class="md:hidden text-2xl text-zinc-900" aria-label="Ouvrir le menu">
      &#9776;
    </button>
  </div>
</nav>
```



Trois manipulations :

1. Ouvre les DevTools (`F12` sur Windows, `Cmd + Option + I` sur Mac), active le **mode responsive** (l'icône téléphone/tablette, ou `Ctrl + Shift + M` / `Cmd + Option + M`). Fais varier la largeur autour de 768 px : les liens disparaissent, le bouton hamburger apparaît. C'est `hidden md:flex` d'un côté et `md:hidden` de l'autre.
2. Remplace `hidden md:flex` par `flex`. Les liens restent visibles même en mobile, et ça déborde. Remets la version d'origine.
3. Remplace `md:` par `lg:` partout (trois endroits). Le basculement se fait maintenant à 1024 px au lieu de 768.

> **NOTE**
>
> Le bouton hamburger ne fait rien pour l'instant : il n'y a pas de JavaScript. C'est normal, on le branchera quand on verra la réactivité de Vue.

Palier 2 — Complète les trous

Repars de la navbar ci-dessus et remplace ces classes par tes choix.

XML

```xml
<!-- ??? fond de la navbar : bg-white | bg-stone-100 | bg-zinc-900 -->
<!-- ??? bordure basse : border-b border-zinc-200 | border-b-2 border-zinc-900 |
border-b-4 border-amber-300 -->
<nav class="??? ??? px-6 py-4 mb-10">
  <div class="mx-auto max-w-5xl flex items-center justify-between">

    <!-- ??? taille du logo : text-lg | text-xl | text-2xl -->
    <!-- ??? caractère du logo : font-bold tracking-tight | font-serif | font-mono
uppercase tracking-widest -->
    <span class="??? ??? text-zinc-900">MonSite</span>

    <!-- ??? espacement entre les liens : gap-4 | gap-8 | gap-12 -->
    <div class="hidden md:flex ???">

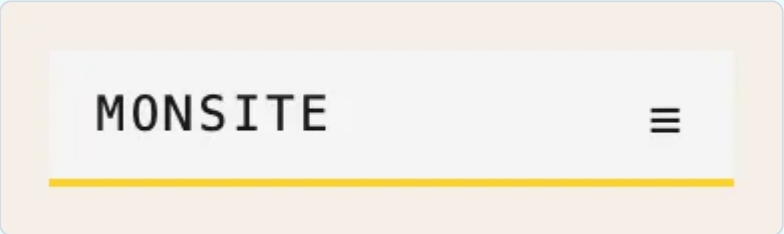
      <!-- ??? survol des liens : hover:text-zinc-900 | hover:underline
hover:underline-offset-4 | hover:bg-amber-300 -->
      <a href="#" class="text-zinc-700 ??? transition">Accueil</a>
      <a href="#" class="text-zinc-700 ??? transition">Projets</a>
      <a href="#" class="text-zinc-700 ??? transition">Contact</a>
    </div>

    <button class="md:hidden text-2xl text-zinc-900" aria-label="Ouvrir le menu">
      &#9776;
    </button>
  </div>
</nav>
```

```xml
<nav class="bg-stone-100 border-b-4 border-amber-300 px-6 py-4 mb-10">
  <div class="mx-auto max-w-5xl flex items-center justify-between">
    <span class="text-2xl font-mono uppercase tracking-widest text-zinc-
900">MonSite</span>

    <div class="hidden md:flex gap-8">
      <a href="#" class="text-zinc-700 hover:underline hover:underline-offset-4
transition">Accueil</a>
      <a href="#" class="text-zinc-700 hover:underline hover:underline-offset-4
transition">Projets</a>
      <a href="#" class="text-zinc-700 hover:underline hover:underline-offset-4
transition">Contact</a>
    </div>

    <button class="md:hidden text-2xl text-zinc-900" aria-label="Ouvrir le menu">
      &#9776;
    </button>
  </div>
</nav>
```



Si tu choisis `bg-zinc-900`, pense à repasser le logo et les liens en clair : `text-stone-50`, `text-zinc-300`. Sinon, contraste raté.

Critères observables

- [ ] En dessous de 768 px : liens cachés, hamburger visible
- [ ] Au-dessus : liens visibles, hamburger caché
- [ ] Les liens réagissent au survol
- [ ] Logo et liens lisibles sur le fond choisi
- [ ] La navbar ne déborde pas horizontalement en mobile

Montre au prof le basculement en direct dans les DevTools.

Exercice 4 — Formulaire de contact

Durée indicative : 40 minutes.

Palier 2 — Complète les trous

La structure HTML est complète, y compris les `for` / `id` qui associent chaque label à son champ. Tu ne touches qu'aux classes.

XML

```xml
<section class="mb-12">
  <h2 class="text-2xl font-bold text-zinc-900 mb-4">Me contacter</h2>

  <!-- ??? fond et cadre du bloc : bg-white border border-zinc-200 | bg-white border-2 border-zinc-900 | bg-stone-50 border-none -->
  <!-- ??? padding du bloc : p-6 | p-8 | p-10 -->
  <form class="max-w-md ??? ??? space-y-5">

    <div>
      <!-- ??? style des labels : text-sm font-medium text-zinc-700 | text-xs uppercase tracking-widest text-zinc-500 | text-base font-bold text-zinc-900 -->
      <label for="nom" class="block mb-1.5 ???">Nom complet</label>

      <!-- ??? bordure du champ au repos : border border-zinc-300 | border-2 border-zinc-900 | border-b-2 border-zinc-400 -->
      <!-- ??? focus clavier visible : focus-visible:outline-2 focus-visible:outline-offset-2 focus-visible:outline-zinc-900 | focus-visible:ring-2 focus-visible:ring-zinc-900 | focus-visible:border-zinc-900 focus-visible:bg-amber-50 -->
      <input
        id="nom"
        type="text"
        placeholder="Prénom Nom"
        class="w-full px-4 py-2.5 rounded-md placeholder:text-zinc-400 ??? ???"
      >
    </div>

    <div>
      <label for="email" class="block mb-1.5 ???">Email</label>
      <input
        id="email"
        type="email"
        placeholder="prenom@exemple.be"
        class="w-full px-4 py-2.5 rounded-md placeholder:text-zinc-400 ??? ???"
      >
    </div>

    <div>
      <label for="message" class="block mb-1.5 ???">Message</label>

      <!-- ??? mêmes bordure et focus que les champs ci-dessus, pour rester cohérent -->
      <textarea
        id="message"
        rows="4"
        placeholder="Ton message"
        class="w-full px-4 py-2.5 rounded-md placeholder:text-zinc-400 ??? ???"
      ></textarea>
    </div>

    <!-- ??? bouton d'envoi, pleine largeur : w-full bg-zinc-900 text-stone-50 | w-full bg-red-600 text-white | w-full bg-amber-300 text-zinc-900 border-2 border-zinc-900 -->
```

```
    <!-- ??? survol du bouton : hover:bg-zinc-700 | hover:opacity-90 | hover:-translate-y-0.5 -->
    <button
      type="submit"
      class="??? px-6 py-3 rounded-md font-medium ??? focus-visible:outline-2 focus-visible:outline-offset-2 focus-visible:outline-zinc-900 transition"
    >
      Envoyer
    </button>
  </form>
</section>
```

XML

```xml
<form class="max-w-md bg-white border-2 border-zinc-900 p-8 space-y-5">
  <div>
    <label for="nom" class="block mb-1.5 text-xs uppercase tracking-widest text-zinc-500">Nom complet</label>
    <input
      id="nom"
      type="text"
      placeholder="Prénom Nom"
      class="w-full px-4 py-2.5 rounded-md placeholder:text-zinc-400 border border-zinc-300 focus-visible:ring-2 focus-visible:ring-zinc-900"
    >
  </div>
  <!-- email et message sur le même modèle -->
  <button
    type="submit"
    class="w-full bg-amber-300 text-zinc-900 border-2 border-zinc-900 px-6 py-3 rounded-md font-medium hover:-translate-y-0.5 focus-visible:outline-2 focus-visible:outline-offset-2 focus-visible:outline-zinc-900 transition"
  >
    Envoyer
  </button>
</form>
```



Les trois champs doivent recevoir **les mêmes** classes de bordure et de focus : un formulaire où chaque champ réagit différemment donne l'impression que la page est cassée.

Checklist accessibilité

Celle-ci, on la vérifie ensemble, champ par champ.

- [ ] Chaque champ a un `<label>` avec un `for` qui correspond exactement à son `id`
- [ ] Je clique sur le texte du label : le curseur saute dans le champ (c'est le test qui prouve que l'association marche)
- [ ] Je parcours tout le formulaire au `[TAB]` : à chaque étape, je vois où je suis
- [ ] Le focus est visible sur les trois champs **et** sur le bouton
- [ ] Le texte des labels contraste franchement avec le fond (pas de `text-zinc-400` sur blanc)
- [ ] Le `placeholder` ne remplace pas le label : il donne un exemple, il ne dit pas ce qu'on demande
- [ ] Les trois champs ont le même traitement visuel

Montre au prof le parcours complet au clavier, sans toucher la souris.

Exercice 5 — Grille de projets

Durée indicative : 45 minutes.

C'est la future grille de projets de ton portfolio.

Palier 1 — Copie et observe

Voici **une** carte. Tu la colles dans la grille, puis tu la dupliques **cinq fois de plus** (copier-coller) pour en avoir six, en changeant le `?random=1` en `2`, `3`… et le titre de chaque projet.

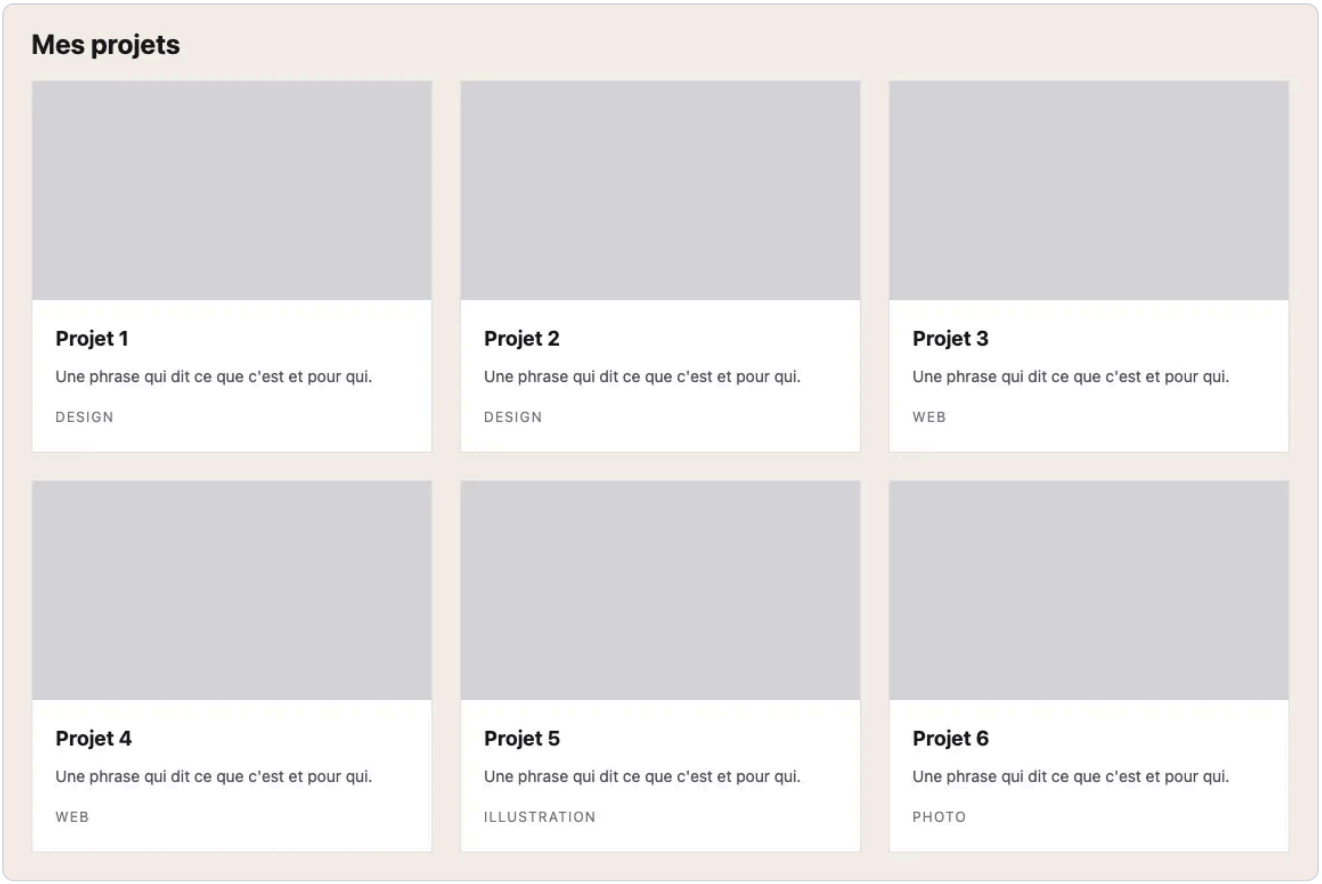
```xml
<section class="mb-12">
  <h2 class="text-2xl font-bold text-zinc-900 mb-4">Mes projets</h2>

  <div class="grid grid-cols-1 md:grid-cols-2 lg:grid-cols-3 gap-6">

    <article class="bg-white border border-zinc-200 overflow-hidden hover:shadow-lg
transition">
      <img
        src="https://picsum.photos/600/400?random=1"
        alt="Aperçu du projet 1"
        class="w-full h-48 object-cover"
      >
      <div class="p-5">
        <h3 class="text-lg font-bold text-zinc-900">Projet 1</h3>
        <p class="mt-2 text-sm text-zinc-600 leading-relaxed">
          Une phrase qui dit ce que c'est et pour qui.
        </p>
        <span class="mt-4 inline-block text-xs uppercase tracking-widest text-zinc-
500">
          Design
        </span>
      </div>
    </article>

    <!— Duplique cet <article> cinq fois de plus —>

  </div>
</section>
```



Deux manipulations :

1. Ouvre les DevTools en mode responsive et fais varier la largeur : 1 colonne en mobile, 2 dès 768 px (`md:grid-cols-2`), 3 dès 1024 px (`lg:grid-cols-3`). Change `lg:grid-cols-3` en `lg:grid-cols-4` et observe.
2. Supprime `object-cover` sur une seule carte et compare-la à ses voisines : l'image est écrasée. C'est à ça que sert cette classe.

Palier 3 — À toi

Consigne : redessine cette carte dans **la direction que tu as choisie à l'exercice 1**. Une seule carte à concevoir, puis tu répercutes sur les six (copier-coller). Tu peux tout changer : supprimer l'image, la mettre en noir et blanc, passer le titre en énorme, mettre le tag en tampon coloré, retirer les bordures, inverser les couleurs.

Remplace les faux titres par **tes vrais projets** — même s'ils ne sont pas encore faits : les projets de l'année, un travail de l'an dernier, une envie. Tu réécriras plus tard, mais du texte réel donne un vrai design.

Banque de classes

Catégorie	Classes
Fond	`bg-white` , `bg-stone-100` , `bg-zinc-900` , `bg-amber-100` , `bg-blue-700`

Catégorie	Classes
Texte	`text-lg`, `text-2xl`, `font-serif`, `font-mono`, `font-black`, `uppercase`, `tracking-widest`, `text-zinc-600`, `text-stone-50`, `line-clamp-2`
Espacement	`p-4`, `p-6`, `gap-2`, `gap-6`, `gap-10`, `mt-2`, `mt-4`
Bordure	`border`, `border-2`, `border-zinc-900`, `border-t-4`, `rounded-none`, `rounded-lg`, `overflow-hidden`
Image	`h-40`, `h-56`, `object-cover`, `aspect-video`, `aspect-square`, `grayscale`, `hover:grayscale-0`
États	`hover:shadow-lg`, `hover:-translate-y-1`, `hover:bg-amber-300`, `transition`, `focus-visible:ring-2`
Responsive	`grid-cols-1`, `sm:grid-cols-2`, `md:grid-cols-2`, `lg:grid-cols-3`, `md:gap-8`

Critères observables

- [ ] Six cartes, toutes traitées exactement pareil
- [ ] La grille passe de 1 à 2 à 3 colonnes selon la largeur
- [ ] Les images ne sont pas déformées
- [ ] Les titres sont mes vrais projets, pas « Projet 1 »
- [ ] La carte réagit au survol
- [ ] La direction esthétique est la même que celle des exercices 1 et 2
- [ ] Aucun néon, aucun glassmorphism, aucun dégradé violet-rose

Montre au prof la grille en mode responsive et la checklist.

Chercher par soi-même

Dans la doc Tailwind

1. Va sur tailwindcss.com/docs.
2. Utilise la barre de recherche (`Ctrl + K` sur Windows, `Cmd + K` sur Mac), en anglais et par **propriété CSS**, pas par intention : cherche `border-radius` plutôt que « coins arrondis », `object-fit` plutôt que « image pas déformée ».
3. Chaque page liste toutes les classes de la famille avec leur équivalent CSS. Quand tu as le CSS en tête, tu as la classe.

> **ATTENTION**
>
> Une recherche Google te renverra souvent vers des articles écrits pour Tailwind **v3**. Les signes qui trahissent : un `tailwind.config.js`, `bg-gradient-to-r` (v4 : `bg-linear-to-r`), `flex-shrink-0` (v4 : `shrink-0`). En cas de doute, la doc officielle fait foi.

Demander à une IA sans se faire refiler du slop

Une IA à qui tu ne donnes pas de direction te rendra la moyenne d'Internet : dégradé violet-rose, coins arrondis, ombre floue, glassmorphism. Ce n'est pas la faute du modèle, c'est la tienne : tu n'as rien décidé, donc elle décide à ta place, et elle décide comme tout le monde.

Mauvais prompt

> *Fais-moi un bouton stylé en Tailwind.*

Bon prompt

> *Écris le template d'un bouton en Tailwind CSS v4, classes uniquement, pas de CSS custom. Direction : imprimé riso — un aplat de couleur franche, une bordure noire de 2 px, angles droits, aucune ombre floue. Palette : fond crème `stone-100`, encre `zinc-900`, accent `amber-300`. Contraintes : focus clavier visible avec `focus-visible:`, contraste texte/fond d'au moins 4,5:1, maximum 10 classes. Interdits : dégradés, `backdrop-blur`, néon, `shadow-xl`. Donne-moi trois variantes qui diffèrent vraiment, pas trois nuances de la même.*

La différence tient en quatre choses : une **direction** nommée, une **palette** imposée, des **contraintes** vérifiables, des **interdits** explicites. Et surtout : tu dois pouvoir dire *pourquoi* tu gardes le résultat. Si tu ne peux pas, rebriefe.

> **ASTUCE**
>
> Demande toujours de **courtes** réponses et relis chaque classe. Une IA invente des classes qui n'existent pas (c'est arrivé à ton prof avec `shadow-brutalist`). L'autocomplétion de VS Code est ton filtre : une classe qui n'apparaît pas dans la liste n'existe pas.

Récap

- Palier 1 : on copie, on modifie une classe à la fois, on regarde.
- Palier 2 : on choisit parmi des candidates — plusieurs réponses sont justes, c'est l'intention qui tranche.
- Palier 3 : on conçoit, avec des critères observables et une banque de classes.
- Une direction esthétique tenue sur les exercices 1, 2 et 5 : c'est déjà l'identité de ton portfolio.

- Focus visible et contraste sur chaque exercice. Ce n'est pas du bonus.
- Cherche par propriété CSS dans la doc v4 ; brief une IA avec direction, palette, contraintes et interdits.

3.8 Les Composants Vue : Props & Réutilisabilité

Tu sais ce qui différencie un dev organisé d'un dev chaotique ? **Les composants.** C'est la base de tout en Vue, et si tu comprends pas ça, autant retourner faire du HTML statique en 2005. (▀̿_▀̿)

Objectifs du chapitre

À la fin de ce chapitre, tu seras capable de :

- ✅ Créer un composant Vue réutilisable
- ✅ Passer des données à un composant via les **props**
- ✅ Valider et typer tes props correctement
- ✅ Réutiliser un même composant avec des configurations différentes

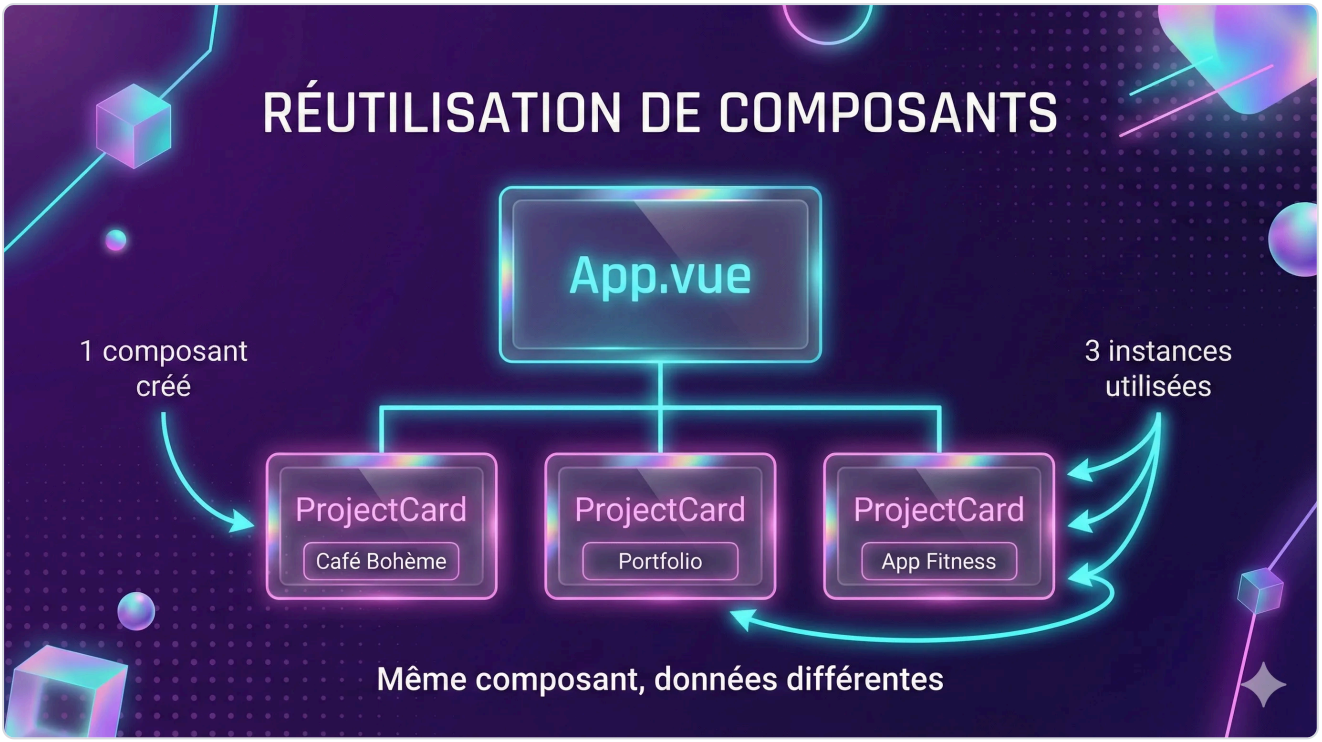
Pourquoi créer des composants ?

Imagine que tu designs un site portfolio. Tu as des **cartes de projet** partout : sur la page d'accueil, sur la page projets, peut-être même sur la page contact.

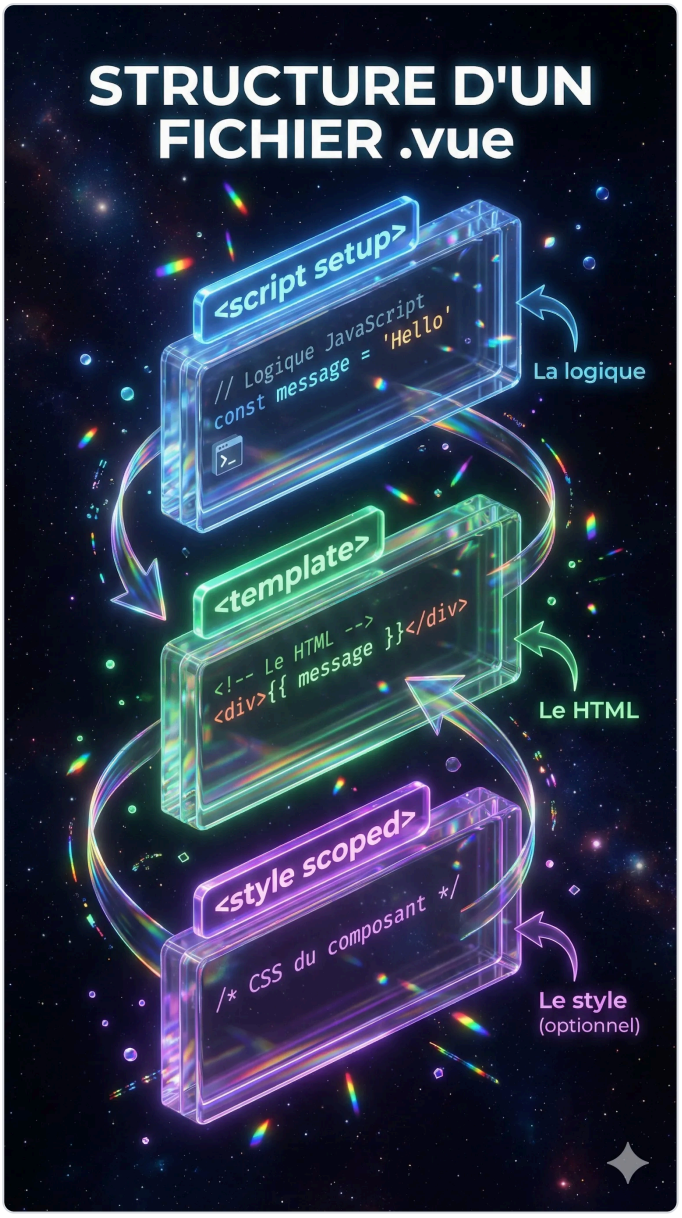
Sans composants : Tu copies-colles le même bloc HTML 15 fois. Un jour, le client veut changer le border-radius. Tu dois modifier... 15 endroits. 💀

Avec composants : Tu crées UN composant `ProjectCard.vue`. Tu l'utilises 15 fois. Tu changes le border-radius UNE fois. Terminé.

C'est exactement comme les **symboles dans Figma**. Tu modifies le symbole maître, toutes les instances se mettent à jour. Même logique.


RÉUTILISATION DE COMPOSANTS
App.vue
1 composant
créé
3 instances
utilisées
ProjectCard
Café Bohème
ProjectCard
Portfolio
ProjectCard
App Fitness
Même composant, données différentes

Anatomie d'un composant Vue



Un composant Vue, c'est un fichier `.vue` avec trois sections :

```XML
<script setup>
// La logique JavaScript
</script>

<template>
  <!-- Le HTML de ton composant -->
  <div class="ma-classe">
    <p>Hello !</p>
  </div>
</template>

<style scoped>
/* Le CSS (optionnel, privilégie Tailwind) */
</style>
```

Les règles d'or :

Règle	Explication
Un seul élément racine	Ton `<template>` doit contenir UN élément parent qui englobe tout
`<script setup>`	La syntaxe moderne, plus simple que l'ancienne
`scoped` sur le style	Le CSS ne s'applique qu'à CE composant

Créer et utiliser un composant

Étape 1 : Crée le fichier dans `/src/components/`

```XML
<!-- src/components/MonBouton.vue -->
<script setup>
// Rien pour l'instant
</script>

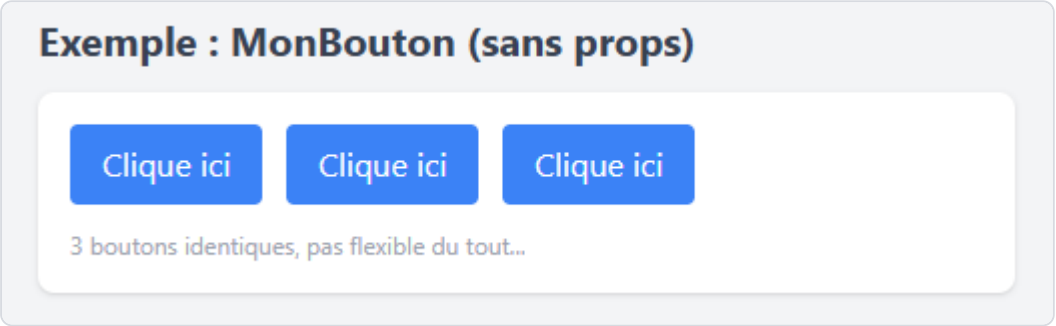
<template>
  <button class="bg-blue-500 text-white px-4 py-2 rounded">
    Clique ici
  </button>
</template>
```

Étape 2 : Importe-le et utilise-le dans `App.vue`

```xml
<!-- src/App.vue -->
<script setup>
import MonBouton from './components/MonBouton.vue'
</script>

<template>
  <div>
    <h1>Ma page</h1>
    <MonBouton />
    <MonBouton />
    <MonBouton />
  </div>
</template>
```

⚠️ **Important** : Tu dois toujours importer tes composants avant de les utiliser !



Convention de nommage

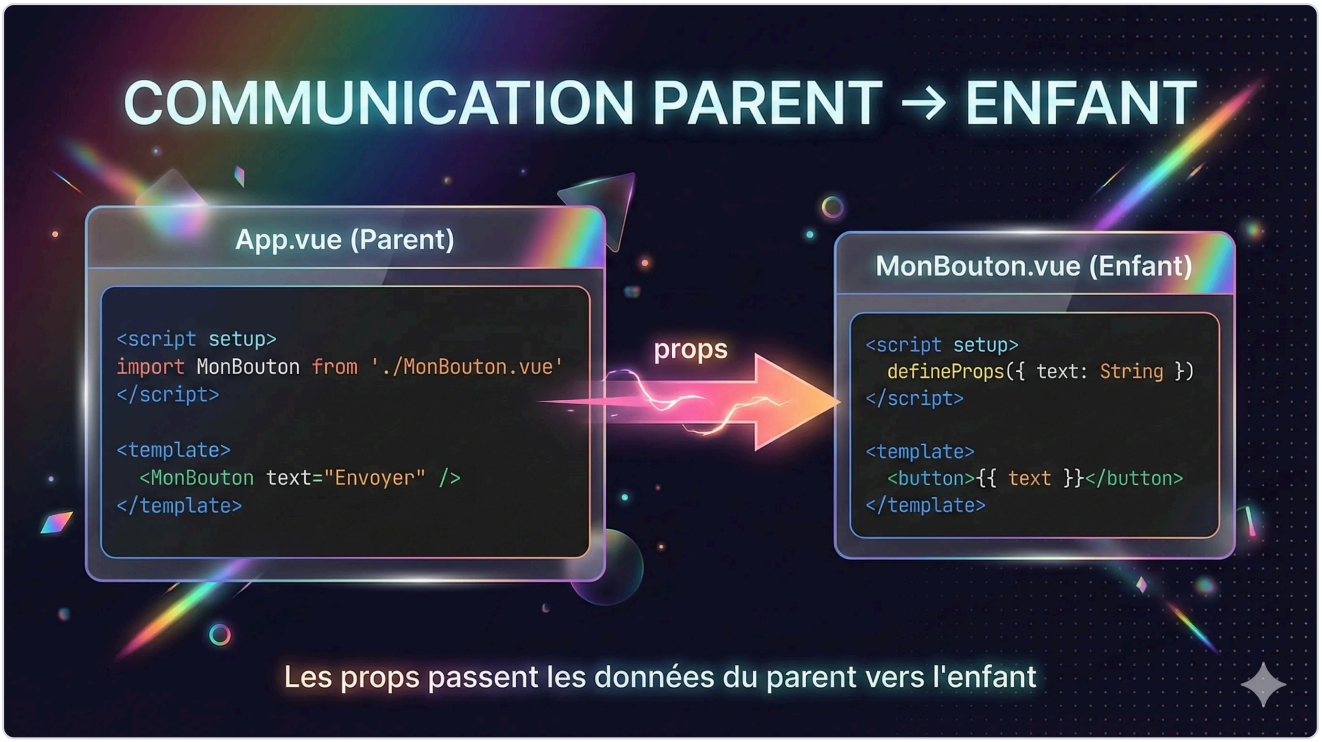
Format	Exemple	Utilisation
PascalCase (fichier)	`MonBouton.vue`	Nom du fichier
PascalCase (template)	`<MonBouton />`	Dans le HTML

💡 **Conseil** : Évite les noms génériques comme `Button.vue` ou `Card.vue`. Préfixe avec `Base`, `App` ou le nom de ton projet : `BaseButton.vue`, `AppCard.vue`.

Les Props : rendre un composant configurable

Ok, ton bouton fonctionne. Mais il affiche toujours "Clique ici". Pas très flexible. 😕

Les **props** permettent de passer des données du parent vers l'enfant. C'est comme les paramètres d'une fonction.



Déclarer une prop

XML

```xml
<!-- src/components/MonBouton.vue -->
<script setup>
defineProps({
  text: String
})
</script>

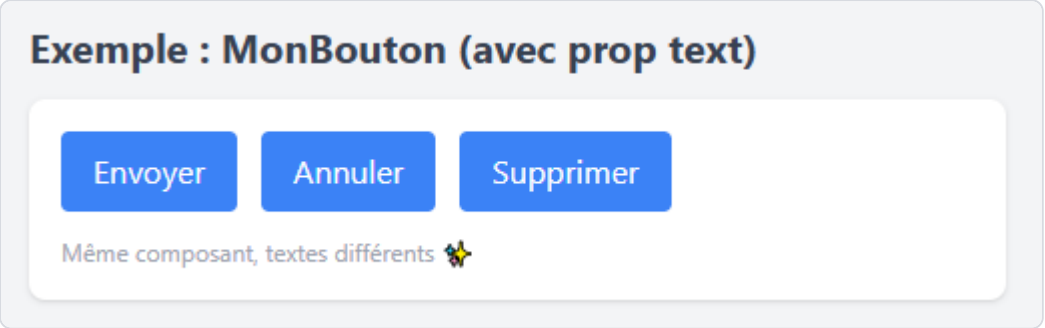
<template>
  <button class="bg-blue-500 text-white px-4 py-2 rounded">
    {{ text }}
  </button>
</template>
```

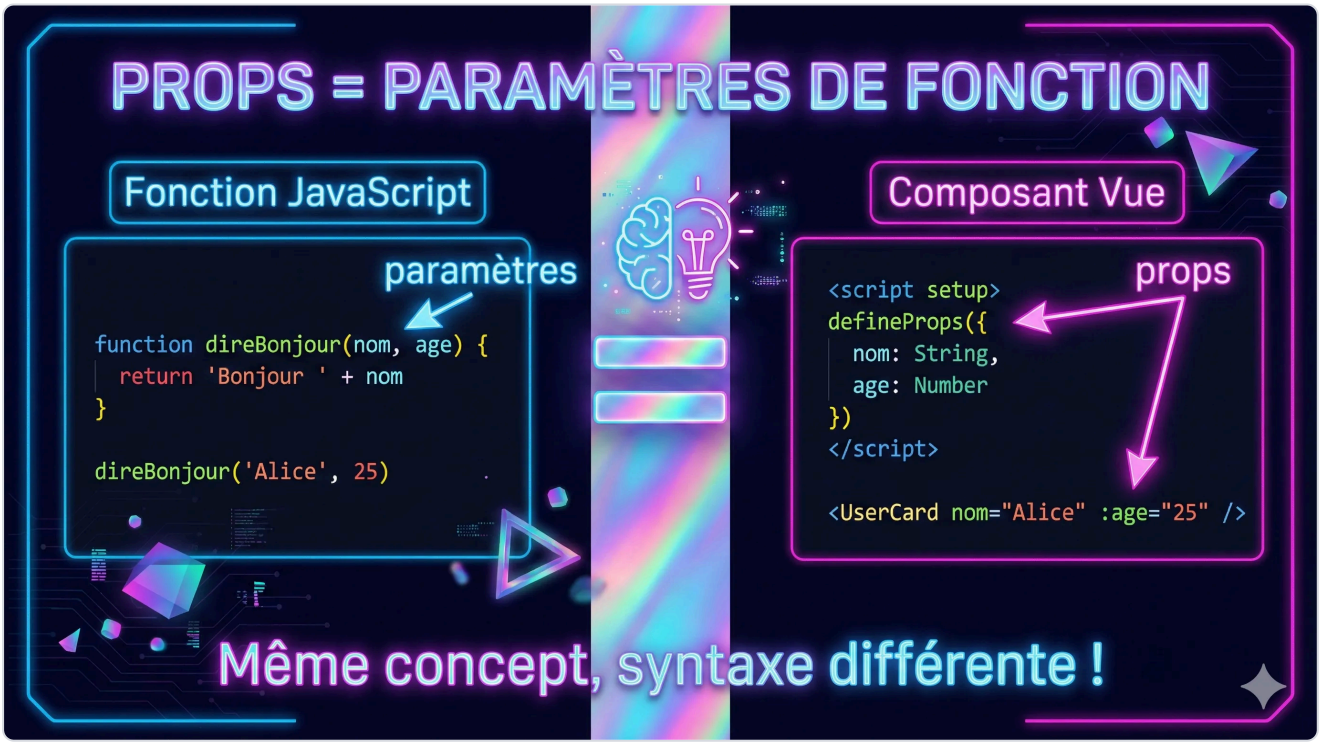
Utiliser le composant avec sa prop

```xml
<script setup>
import MonBouton from './components/MonBouton.vue'
</script>

<template>
  <div class="flex gap-4">
    <MonBouton text="Envoyer" />
    <MonBouton text="Annuler" />
    <MonBouton text="Supprimer" />
  </div>
</template>
```

Résultat : 3 boutons avec des textes différents, même composant. 🎉





Syntaxe : avec ou sans `:`

C'est LA source d'erreur numéro 1 des débutants.

Syntaxe	Ce qui est passé	Exemple
`prop="valeur"`	Une **string** littérale	`text="Hello"` → reçoit `"Hello"`
`:prop="valeur"`	Une **expression JavaScript**	`:count="42"` → reçoit le nombre `42`

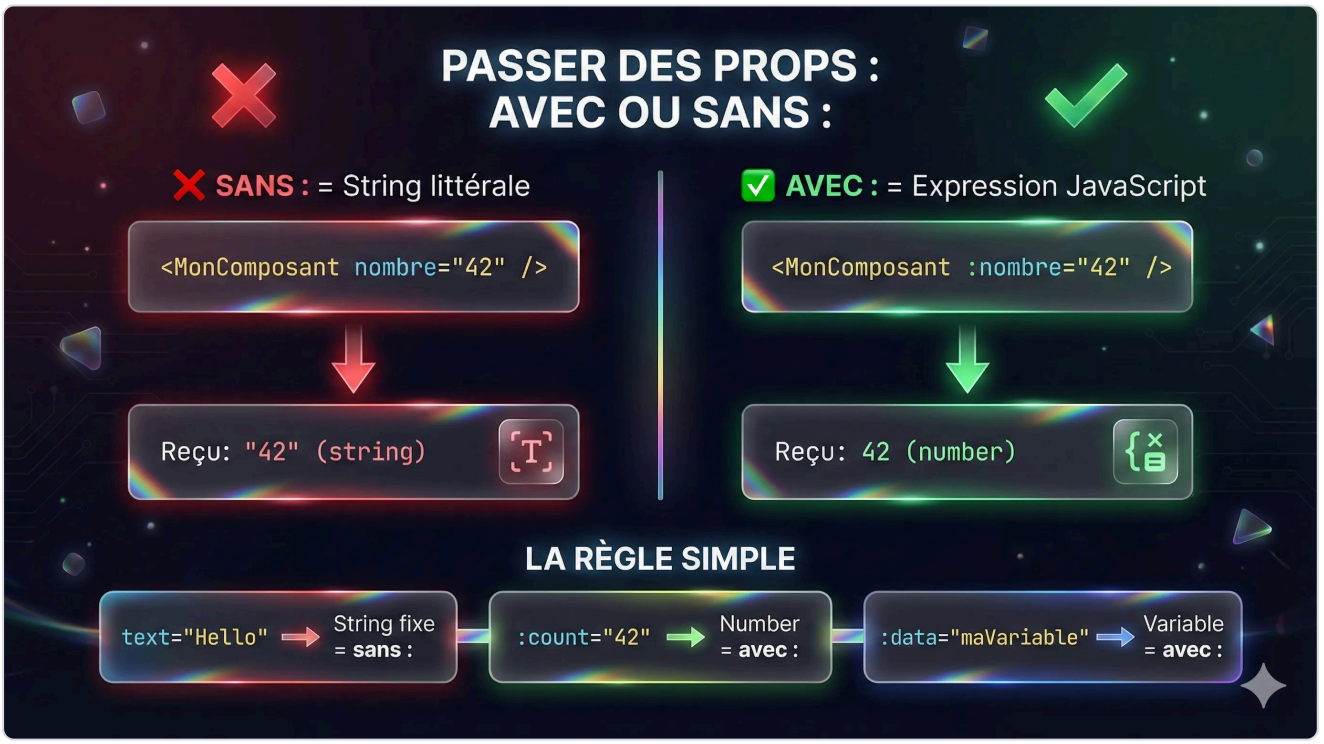
```xml
<!-- ❌ FAUX : "42" est une string -->
<MonComposant nombre="42" />

<!-- ✅ CORRECT : 42 est un number -->
<MonComposant :nombre="42" />

<!-- ✅ CORRECT : on passe une variable -->
<MonComposant :nombre="maVariable" />

<!-- ✅ CORRECT : les strings marchent sans ":" -->
<MonComposant text="Hello" />
```

La règle simple : Si c'est une string fixe → sans `:`. Sinon → avec `:`.



Valider ses props

Tu peux (et tu devrais) définir des règles pour tes props.

Syntaxe complète

```xml
<script setup>
defineProps({
  // Juste le type
  titre: String,

  // Type + obligatoire
  id: {
    type: Number,
    required: true
  },

  // Type + valeur par défaut
  variant: {
    type: String,
    default: 'primary'
  },

  // Plusieurs types possibles
  valeur: [String, Number],

  // Validation personnalisée
  taille: {
    type: String,
    validator: (value) ⇒ ['sm', 'md', 'lg'].includes(value)
  }
})
</script>
```

Types disponibles

Type	Exemple de valeur
`String`	`"Hello"`
`Number`	`42`
`Boolean`	`true` / `false`
`Array`	`['a', 'b', 'c']`
`Object`	`{ nom: 'Alice', age: 25 }`

Exemple concret : Bouton avec variants

```XML
<!-- src/components/BaseButton.vue -->
<script setup>
import { computed } from 'vue'

const props = defineProps({
  text: {
    type: String,
    required: true
  },
  variant: {
    type: String,
    default: 'primary',
    validator: (v) => ['primary', 'secondary', 'danger'].includes(v)
  }
})

const classes = computed(() => {
  const variants = {
    primary: 'bg-blue-600 text-white hover:bg-blue-700',
    secondary: 'bg-gray-200 text-gray-800 hover:bg-gray-300',
    danger: 'bg-red-600 text-white hover:bg-red-700'
  }
  return variants[props.variant]
})
</script>

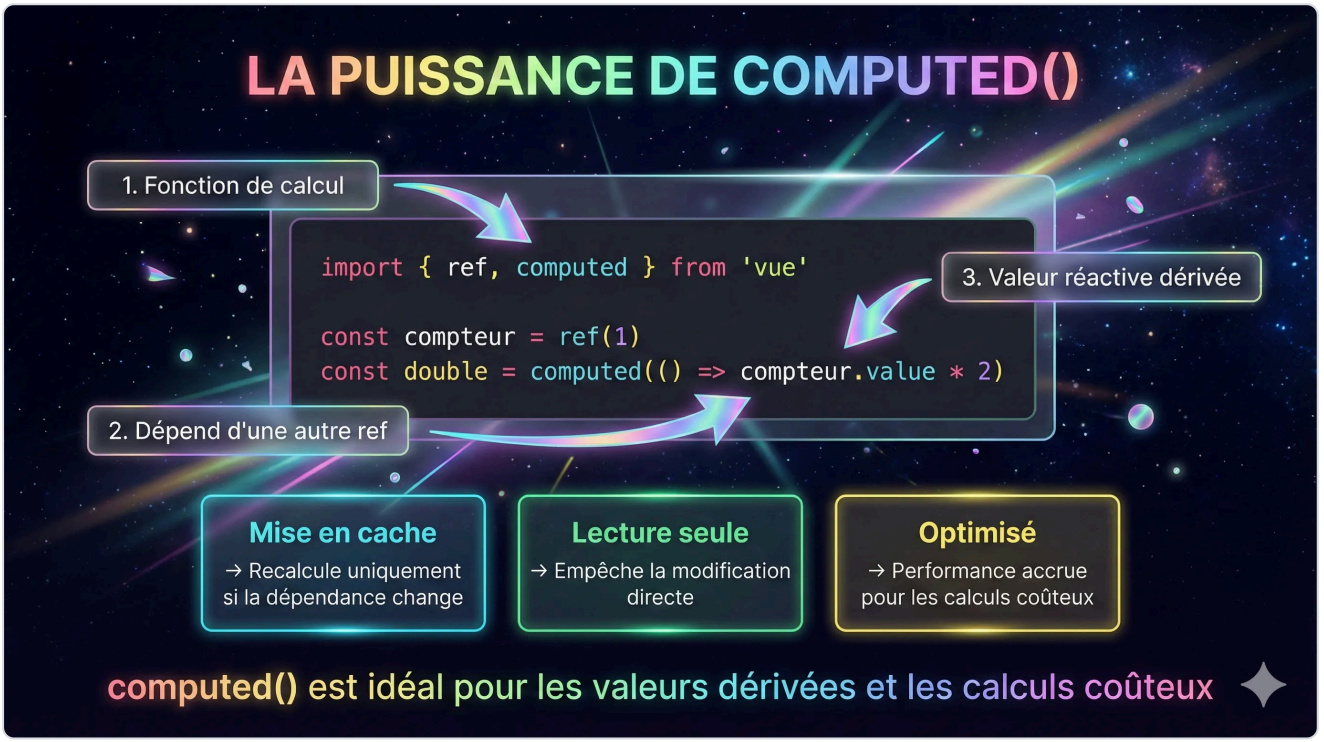
<template>
  <button :class="classes" class="px-4 py-2 rounded font-medium">
    {{ text }}
  </button>
</template>
```

Utilisation :

```XML
<BaseButton text="Enregistrer" variant="primary" />
<BaseButton text="Annuler" variant="secondary" />
<BaseButton text="Supprimer" variant="danger" />
```

⚠️ **Note** : On utilise `computed()` ici pour calculer les classes dynamiquement. On verra ça en détail dans un prochain chapitre, pour l'instant fais-moi confiance !

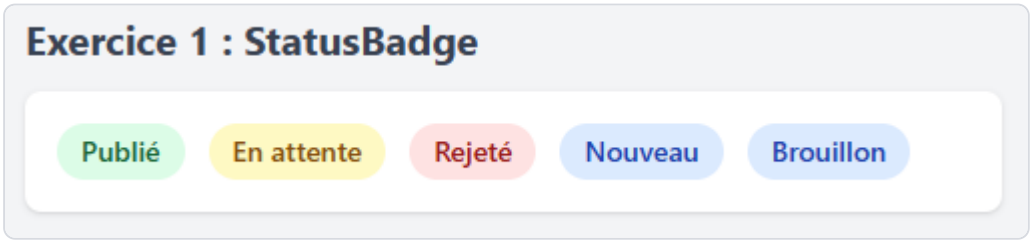




Exercices pratiques

Exercice 1 : Badge de statut ⭐

Objectif : Créer ton premier composant avec props



Ta mission : Créer un composant `StatusBadge.vue` qui affiche un petit badge coloré.

Props à implémenter :

Prop	Type	Requis	Description
`label`	String	✅ Oui	Le texte du badge
`variant`	String	Non (défaut: `"info"`)	La couleur : `"success"`, `"warning"`, `"error"`, `"info"`

Couleurs attendues (Tailwind) :

Variant	Classes
`success`	`bg-green-100 text-green-800`
`warning`	`bg-yellow-100 text-yellow-800`
`error`	`bg-red-100 text-red-800`
`info`	`bg-blue-100 text-blue-800`

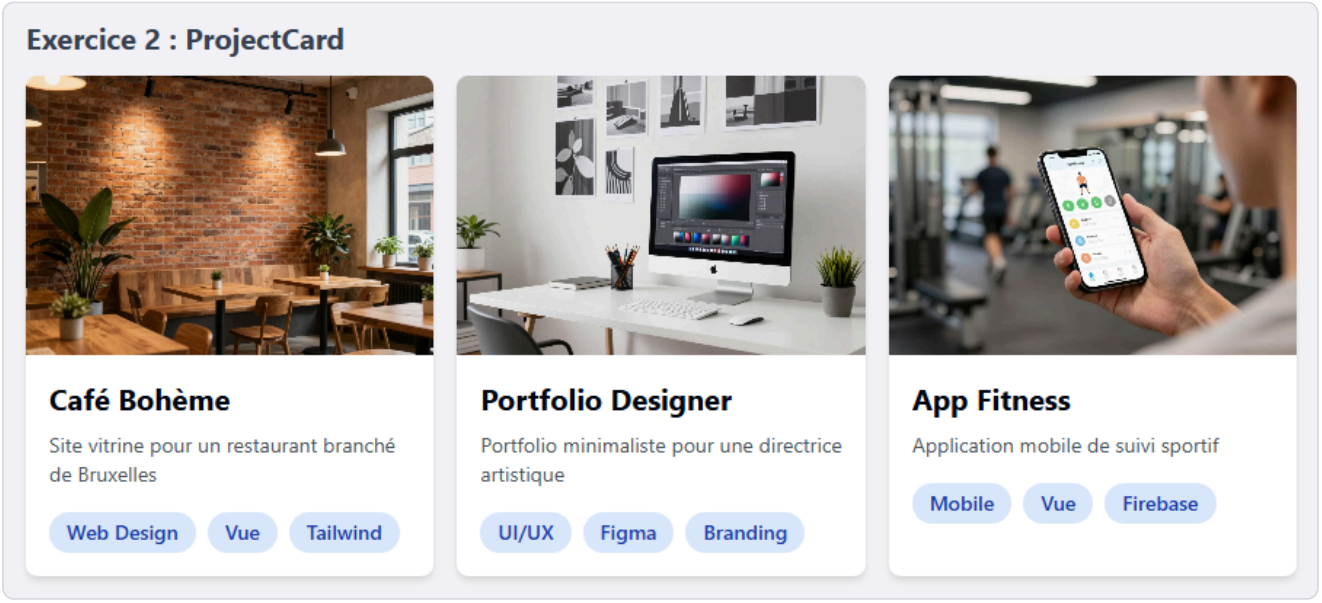
Résultat attendu :

```XML
<StatusBadge label="Publié" variant="success" />
<StatusBadge label="En attente" variant="warning" />
<StatusBadge label="Rejeté" variant="error" />
<StatusBadge label="Nouveau" />  <!— Utilise "info" par défaut —>
```

Le badge doit avoir un style pill : `rounded-full px-3 py-1 text-sm font-medium`.

Exercice 2 : Carte de projet ⭐⭐

Objectif : Composant avec plusieurs props et réutilisation d'un autre composant



Ta mission : Créer un composant `ProjectCard.vue` pour afficher un projet de portfolio.

Props à implémenter :

Prop	Type	Requis	Description
`title`	String	✅ Oui	Titre du projet
`description`	String	✅ Oui	Description courte
`image`	String	✅ Oui	URL de l'image
`tags`	Array	Non (défaut: `[]`)	Liste des tags

Structure HTML suggérée :

```css
┌─────────────────────────────────┐
│  [Image du projet]              │
├─────────────────────────────────┤
│  Titre du projet                │
│  Description courte...          │
│                                 │
│  [Tag 1] [Tag 2] [Tag 3]        │
└─────────────────────────────────┘
```

Contraintes :

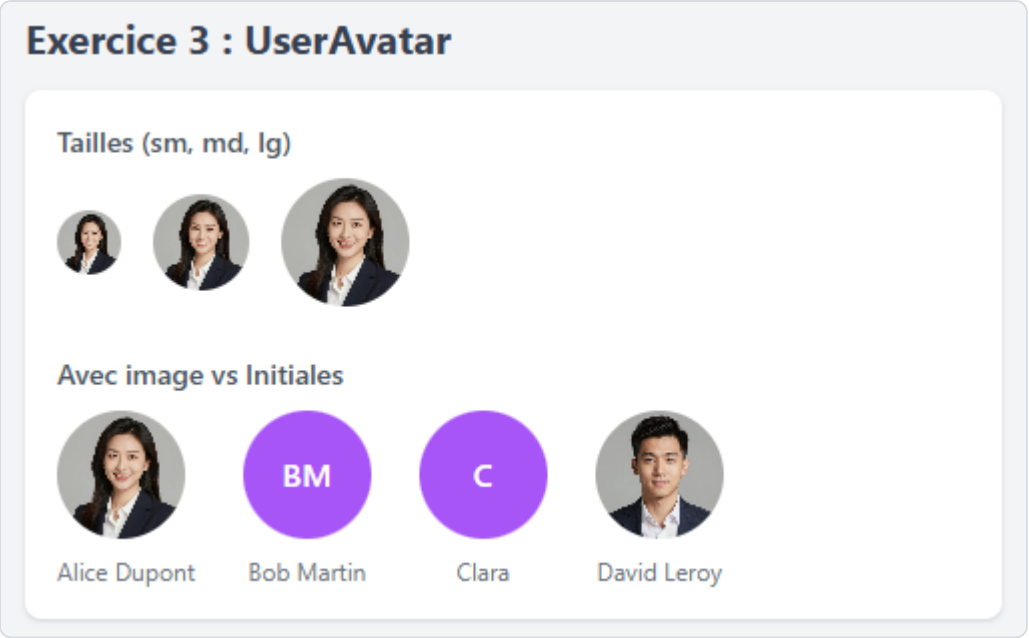
- Utilise ton composant `StatusBadge` pour afficher les tags (avec `v-for`)
- L'image doit utiliser `object-cover` pour garder ses proportions
- N'oublie pas d'importer `StatusBadge` dans ton composant !

Exemple d'utilisation :

```xml
<ProjectCard
  title="Café Bohème"
  description="Site vitrine pour un restaurant branché de Bruxelles"
  image="/images/cafe-boheme.jpg"
  :tags="['Web Design', 'Vue', 'Tailwind']"
/>
```

Exercice 3 : Avatar utilisateur ⭐⭐

Objectif : Gérer les cas conditionnels (image présente ou non)



Ta mission : Créer un composant `UserAvatar.vue` qui affiche soit une photo, soit les initiales de l'utilisateur.

Props à implémenter :

Prop	Type	Requis	Description
`name`	String	✅ Oui	Nom complet de l'utilisateur
`src`	String	Non	URL de la photo (optionnel)
`size`	String	Non (défaut: `"md"`)	Taille : `"sm"`, `"md"`, `"lg"`

Comportement :

- Si `src` est fourni → afficher l'image
- Si `src` est absent → afficher les initiales sur un fond coloré (utilise `v-if` / `v-else`)

Tailles :

Size	Dimensions
`sm`	`w-8 h-8 text-xs`
`md`	`w-12 h-12 text-sm`
`lg`	`w-16 h-16 text-base`

Exemple d'utilisation :

```xml
<!-- Avec image -->
<UserAvatar name="Alice Dupont" src="/photos/alice.jpg" size="lg" />

<!-- Sans image (affiche "AD") -->
<UserAvatar name="Alice Dupont" size="lg" />

<!-- Nom simple (affiche "B") -->
<UserAvatar name="Bob" size="sm" />
```

Aide pour extraire les initiales :

```javascript
function getInitials(name) {
  return name
    .split(' ')
    .map(word => word[0])
    .join('')
    .toUpperCase()
    .slice(0, 2)
}
```

La suite...

Tu sais maintenant créer des composants configurables avec des props. Mais ils sont encore **statiques** — aucune interactivité.

Dans le prochain chapitre, on verra `ref()` pour rendre tes composants **vivants** : boutons qui comptent les clics, menus qui s'ouvrent et se ferment, toggles qui switchent... Bref, le fun commence. (ノ'-')ノ

3.9 Objets et JSON

Objectif : ranger les informations d'une chose dans un seul objet, le lire et le modifier, comprendre ce qu'est un fichier JSON, puis afficher une liste d'objets dans Vue.

1. Pourquoi les objets ?

Imagine un petit bestiaire : une galerie de créatures inventées. Voici la première, avec ce que tu connais déjà :

```javascript
const nom = 'Grolem'
const niveau = 12
const pv = 180
const vivant = true
```

Quatre variables pour **une seule** créature. À la huitième, tu jongles avec `nom8`, `niveau8`, `pv8`... 😵

Un **objet** regroupe ces informations sous un seul nom :

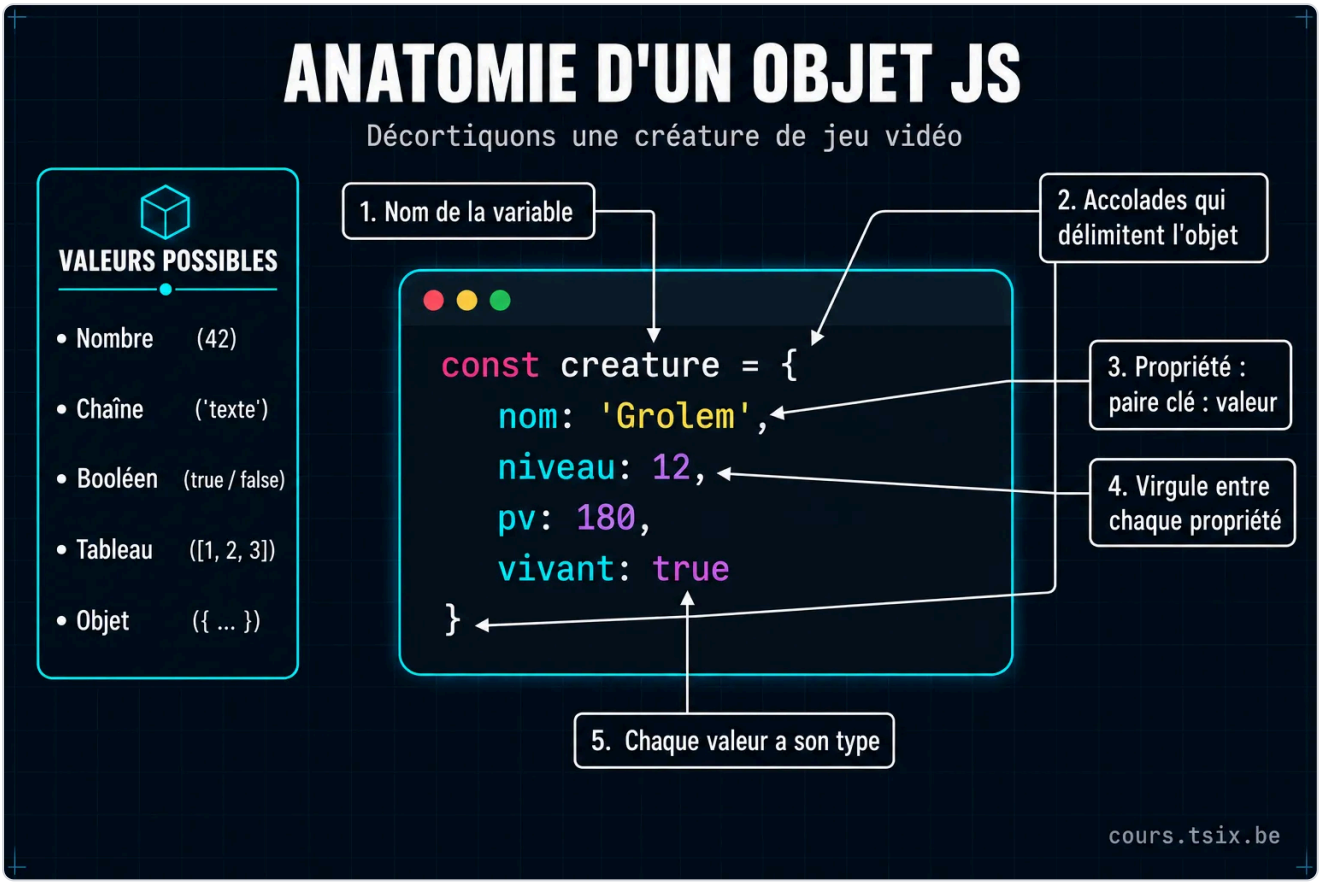
```javascript
const creature = {
  nom: 'Grolem',
  niveau: 12,
  pv: 180,
  vivant: true,
}
```

Une variable, quatre propriétés. Sur ton portfolio, ce sera pareil : un projet a un titre, une image, une description, des tags.

2. Créer un objet

Un objet, c'est un sac à variables. Des accolades `{ }` l'ouvrent et le ferment.

Chaque ligne est une **propriété** : une **clé** (son nom, sans guillemets), deux-points, puis sa **valeur**. Une virgule sépare deux propriétés ; celle après la dernière est facultative, Prettier l'ajoute pour toi.



La valeur peut être de n'importe quel type :

Type	Exemple
Chaîne de caractères	`nom: 'Grolem'`
Nombre	`niveau: 12`
Booléen	`vivant: true`
Tableau	`faiblesses: ['eau', 'mousse']`
Objet	`stats: { pv: 180, attaque: 25 }`

Un objet peut donc en contenir un autre : on y vient à la section 5. Il peut aussi ranger des fonctions, et tu en utilises déjà : `tableau.push()` ou `texte.toUpperCase()`.

NOTE

Les clés de ce chapitre n'ont **ni accent ni espace** : `capacites`, `degats`, `defense`. C'est une convention : une clé sans accent ne s'écrit que d'une seule façon. Les valeurs, elles, peuvent en contenir : `'Éboulement'`.

3. Lire une propriété

Pour lire une propriété : le nom de l'objet, un point, la clé. C'est la **notation point**, celle que tu utiliseras presque toujours.

```javascript
creature.nom    // 'Grolem'
creature.niveau // 12
```

Le commentaire montre ce que répond la console. Pour essayer, ouvre les DevTools (`F12` , ou `Cmd + Option + I` sur Mac), onglet **Console**, et colles-y l'objet. Au premier collage, le navigateur peut te demander de taper `allow pasting` : c'est une protection, tape-le.

Les crochets font la même chose, avec la clé entre apostrophes. Ils servent quand la clé est **dans une variable** :

```javascript
creature['niveau'] // 12

const champ = 'pv'
creature[champ] // 180, comme creature.pv
```

Écrire `creature.champ` chercherait une clé qui s'appelle littéralement `champ` , et répondrait `undefined` .

La règle : le point par défaut, les crochets quand la clé est dans une variable.

4. Modifier, ajouter, supprimer

Pour changer une valeur, un `=` . Si la clé n'existe pas, JavaScript la crée. `delete` la retire.

```javascript
creature.pv = 150
creature.surnom = 'Le Têtu'
delete creature.surnom
```

Mais `creature` est déclarée avec `const` : comment peut-on la modifier ? `const` verrouille l'étiquette du sac, pas son contenu.

Remplacer l'objet entier (`creature = {}`) provoque l'erreur `TypeError: Assignment to constant variable.` Changer ce qu'il contient ne pose aucun problème.

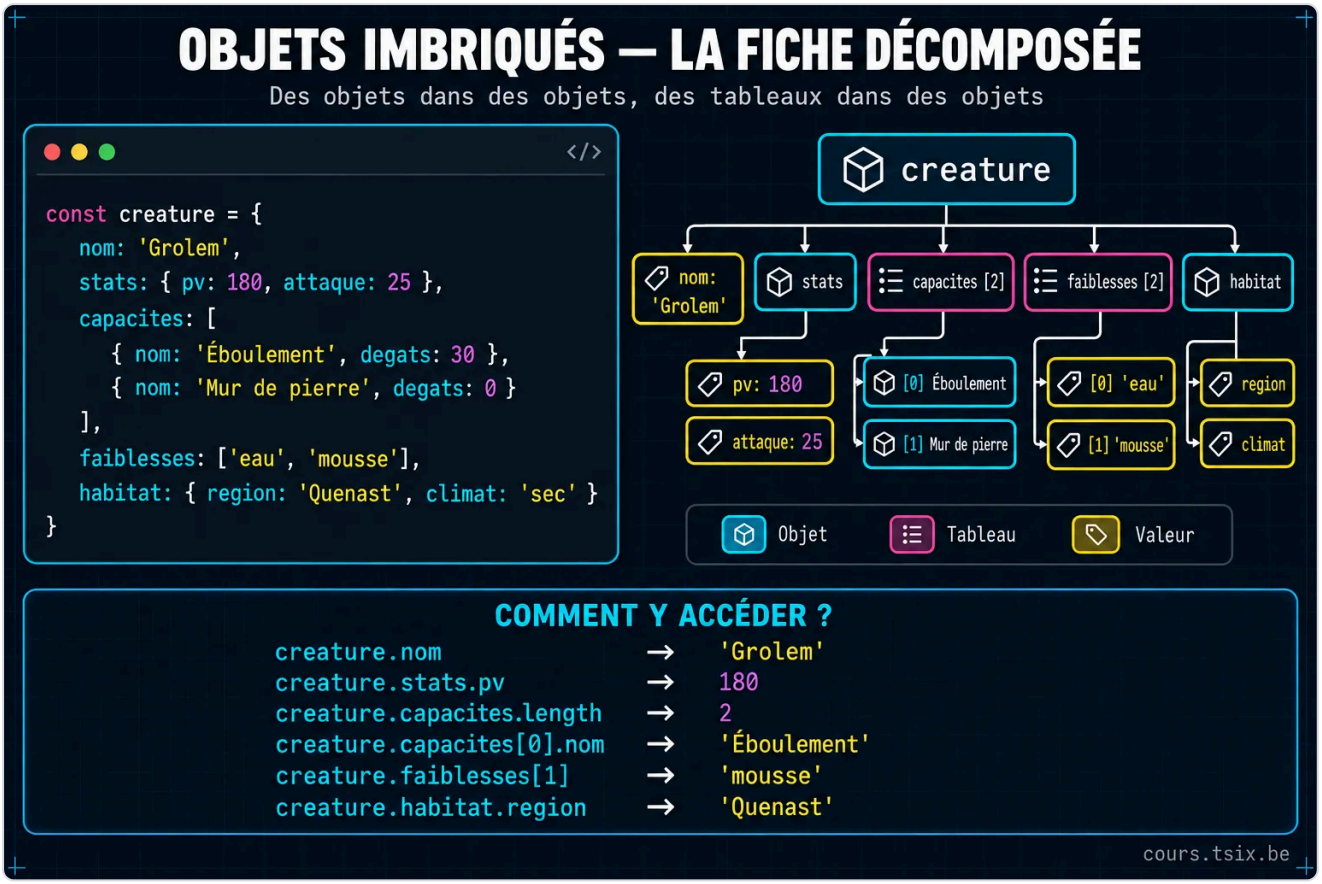
5. Objets dans des objets, tableaux d'objets

Une vraie fiche contient plus que quatre informations. Voici Grolem au complet, l'objet de tout le reste du chapitre :

```javascript
const creature = {
  nom: 'Grolem',
  espece: 'Golem de galets',
  type: 'pierre',
  niveau: 12,
  rarete: 'rare',
  description: "Un tas de galets qui s'est mis debout un jour de pluie et n'a jamais voulu se recoucher.",
  image: '/creatures/grolem.png',
  stats: { pv: 180, attaque: 25, defense: 40, vitesse: 2 },
  capacites: [
    { nom: 'Éboulement', type: 'physique', degats: 30 },
    { nom: 'Mur de pierre', type: 'soutien', degats: 0 },
  ],
  faiblesses: ['eau', 'mousse'],
  habitat: { region: 'Carrières de Quenast', climat: 'sec' },
}
```

`stats` et `habitat` sont des **objets dans l'objet**. `faiblesses` est un **tableau**, `capacites` un **tableau d'objets**.

La description est entre guillemets doubles parce qu'elle contient une apostrophe. Prettier fait ce choix tout seul.



Pour descendre dans l'objet, tu **enchaînes** les accès :

```javascript
creature.nom              // 'Grolem'
creature.stats.pv         // 180
creature.capacites.length // 2
creature.capacites[0].nom // 'Éboulement'
creature.faiblesses[1]    // 'mousse'
creature.habitat.region   // 'Carrières de Quenast'
creature.boss             // undefined (pas d'erreur)
creature.boss.nom         // ❌ TypeError: Cannot read properties of undefined
(reading 'nom')
```

Lis `creature.capacites[0].nom` de gauche à droite, comme une adresse : dans `creature`, prends `capacites`, puis l'élément `0`, puis son `nom`.

Un tableau se compte à partir de zéro : `[0]` est le premier élément. `.length` donne le nombre d'éléments.

Les deux dernières lignes : `creature.boss` n'existe pas, JavaScript répond `undefined`, sans erreur. Mais `creature.boss.nom` demande le nom de… rien. Là, ça plante.

ATTENTION

Dans un projet Vue, `Cannot read properties of undefined` vient presque toujours d'une faute de frappe dans la chaîne : `creature.stat.pv` au lieu de `creature.stats.pv`. Relis-la lettre par lettre, en partant de la gauche.

6. Exercices de lecture

Vérifie que tu sais lire la fiche de Grolem. Réponds dans ta tête (ou dans la console), puis retourne la carte.

JAVASCRIPT

```
const creature = {
  nom: 'Grolem',
  espece: 'Golem de galets',
  type: 'pierre',
  niveau: 12,
  rarete: 'rare',
  description: "Un tas de galets qui s'est mis debout un jour de pluie et n'a jamais
voulu se recoucher.",
  image: '/creatures/grolem.png',
  stats: { pv: 180, attaque: 25, defense: 40, vitesse: 2 },
  capacites: [
    { nom: 'Éboulement', type: 'physique', degats: 30 },
    { nom: 'Mur de pierre', type: 'soutien', degats: 0 },
  ],
  faiblesses: ['eau', 'mousse'],
  habitat: { region: 'Carrières de Quenast', climat: 'sec' },
}
```

Comment lis-tu le nom de la créature ?

`creature.nom`, qui vaut `'Grolem'`.

Et ses points de vie ?

`creature.stats.pv`, qui vaut `180`. Les PV sont rangés dans le sous-objet `stats`.

Comment obtiens-tu le nom de sa première capacité ?

`creature.capacites[0].nom`, qui vaut `'Éboulement'`. Le premier élément d'un tableau porte le numéro `0`.

Combien a-t-elle de faiblesses, et comment le savoir sans compter ?

`creature.faiblesses.length`, qui vaut `2`.

Comment lis-tu sa dernière faiblesse sans connaître la longueur du tableau à l'avance ?

`creature.faiblesses[creature.faiblesses.length - 1]`, ou plus court : `creature.faiblesses.at(-1)`. Les deux valent `'mousse'`.

Que vaut `creature.habitat.climat` ?

`'sec'`.

Que vaut `creature.boss` ? Et `creature.boss.nom` ?

`creature.boss` vaut `undefined` : la clé n'existe pas, sans erreur. `creature.boss.nom` plante : `TypeError: Cannot read properties of undefined (reading 'nom')`.

Comment rends-tu la créature légendaire ?

`creature.rarete = 'légendaire'`. Ça marche même si `creature` est déclarée avec `const` : tu modifies le contenu, pas la variable.

Comment lui ajoutes-tu une propriété `cri` qui vaut `'Grrrolem !'` ?

`creature.cri = 'Grrrolem !'`. La clé n'existait pas : JavaScript la crée.

Combien de dégâts fait sa deuxième capacité ?

`creature.capacites[1].degats`, qui vaut `0`.

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/projet-web-uiux/vuejs/objets-et-json

Un deuxième paquet, sur le JSON et Vue, t'attend à la fin de la section 9.

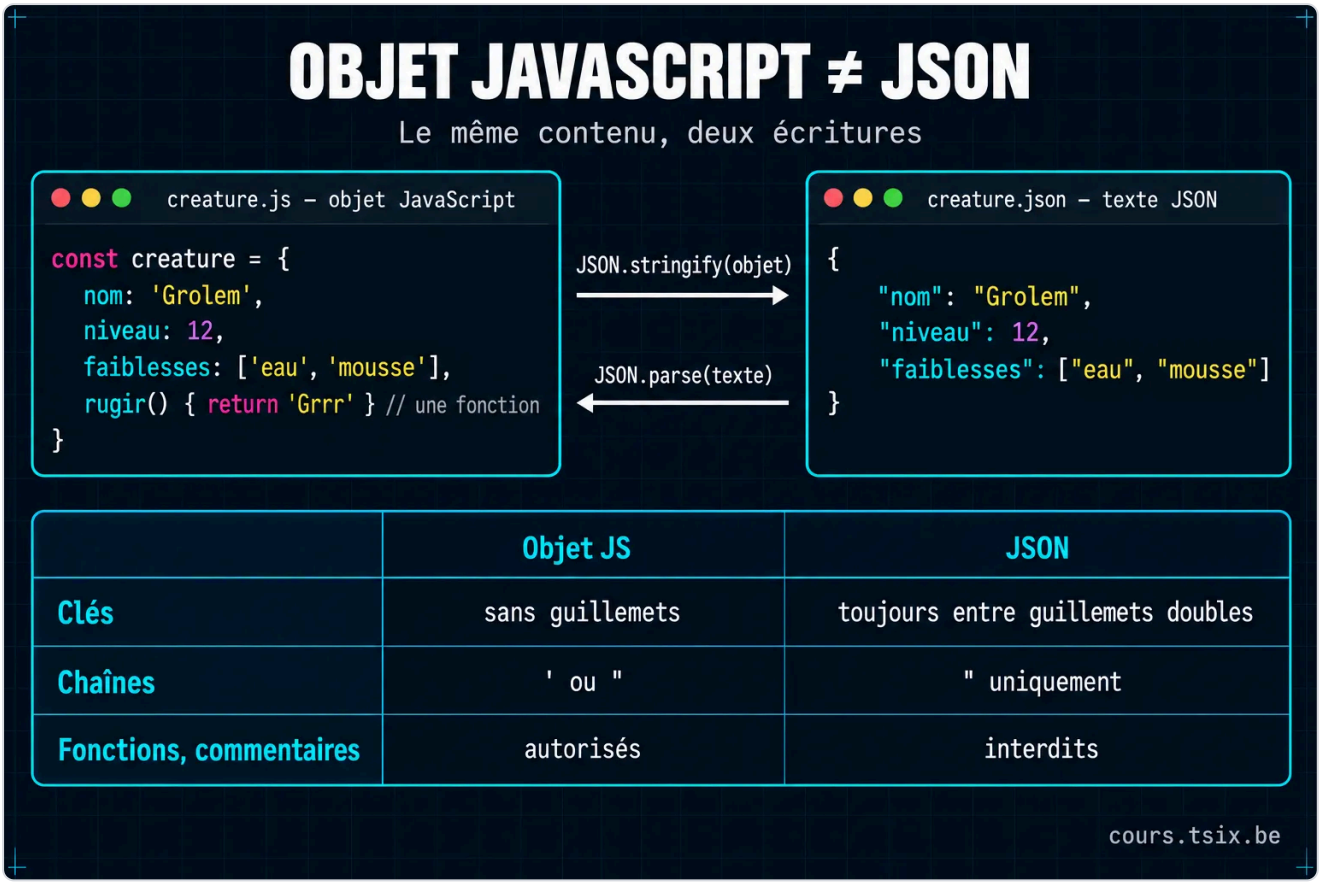
7. JSON : du texte qui ressemble à un objet

JSON (*JavaScript Object Notation*) est un **format de texte** pour écrire des données. La créature de la section 1, en JSON :

```json
{
  "nom": "Grolem",
  "niveau": 12,
  "pv": 180,
  "vivant": true
}
```

On dirait un objet JavaScript. Ce n'en est pas un : c'est du texte, rangé dans un fichier ou envoyé par un serveur, avec des règles plus strictes.

	Objet JavaScript	JSON
Ce que c'est	Une valeur, dans ton code	Du texte, dans un fichier ou sur le réseau
Clés	Sans guillemets : `nom:`	Toujours entre guillemets doubles : `"nom":`
Chaînes	Apostrophes ou guillemets	Guillemets doubles uniquement
Virgule après le dernier élément	Acceptée	Interdite
Fonctions, commentaires, `undefined`	Acceptés	Interdits



C'est le format d'échange universel : fichiers de configuration, réponses des serveurs, contenus d'un site. Tu en as déjà ouvert un : `package.json`.

`JSON.parse` transforme un texte JSON en objet :

```javascript
const texte = '{"nom":"Grolem","niveau":12}'
const grolem = JSON.parse(texte)
grolem.niveau // 12
```

`JSON.stringify` fait l'inverse, d'un objet vers du texte :

```javascript
JSON.stringify(grolem) // '{"nom":"Grolem","niveau":12}'
```

Tu les écriras rarement à la main : pour un fichier, Vite s'en charge.

8. Importer un JSON dans un projet Vue

Les données d'un site n'ont rien à faire au milieu du code. On les range dans `src/data/` :

```
mon-projet/
├── public/
│   └── creatures/
│       ├── grolem.png
│       └── …
└── src/
    ├── data/
    │   └── creatures.json
    └── App.vue
```

`creatures.json` contient un **tableau de huit objets** sur le modèle de Grolem, chacun avec un `id` unique en plus. Le fichier complet est à l'exercice 2.

Dans le `<script setup>` d'un composant, une ligne suffit pour l'utiliser :

JAVASCRIPT

```javascript
import creatures from '@/data/creatures.json'

console.log(creatures.length) // 8
console.log(creatures[1].nom) // Grolem
```

`@` désigne `src/`, comme au chapitre Structure d'un projet.

Vite parse pour toi : il lit le texte, fait le `JSON.parse` et te donne directement le tableau d'objets.

Les images suivent la règle de `public/`. Le JSON indique `/creatures/grolem.png` : le fichier va donc dans `public/creatures/grolem.png`.

ATTENTION

Cet `import` marche parce que Vite est là. Sans Vite, le navigateur refuse d'importer un fichier `.json` écrit tel quel.

9. Et dans Vue ?

Afficher un objet

Un objet déclaré dans le `<script setup>` s'utilise dans le template avec la même chaîne d'accès que dans la console :

```XML
<script setup>
  const creature = {
    nom: 'Grolem',
    niveau: 12,
    image: '/creatures/grolem.png',
    stats: { pv: 180, attaque: 25 },
  }
</script>

<template>
  <img :src="creature.image" :alt="creature.nom" class="w-48">
  <h1>{{ creature.nom }}</h1>
  <p>Niveau {{ creature.niveau }} · {{ creature.stats.pv }} PV</p>
</template>
```

Deux endroits, deux écritures :

- entre deux balises, les moustaches : `{{ creature.stats.pv }}` ;
- dans un attribut, les deux-points devant : `:src="creature.image"`. Sans les deux-points, le navigateur chercherait une image qui s'appelle littéralement `creature.image`. C'est la règle des props du chapitre Components : une expression, pas un texte.

Pour l'instant, on affiche. Modifier l'objet et voir la page suivre toute seule, c'est le chapitre suivant, Ref.

Afficher un tableau d'objets avec `v-for`

Huit créatures, ce n'est pas huit `<li>` : tu en écris un, et `v-for` le répète.

```XML
<script setup>
  import creatures from '@/data/creatures.json'
</script>

<template>
  <ul>
    <li v-for="creature in creatures" :key="creature.id">
      {{ creature.nom }} — niveau {{ creature.niveau }}
    </li>
  </ul>
</template>
```

Lis : « pour chaque `creature` dans `creatures` ». `creatures` est le tableau importé, `creature` l'élément du tour en cours.

`:key` donne à chaque élément un identifiant unique, ici son `id`. Vue s'en sert pour suivre chaque ligne quand la liste change. Le chapitre v-for détaillera le reste : `:key`, l'index, `v-for` sur un objet.

Un sous-tableau ? Un deuxième `v-for`, placé dans le premier, liste les capacités de chaque créature :

```xml
<ul>
  <li v-for="capacite in creature.capacites" :key="capacite.nom">
    {{ capacite.nom }} ({{ capacite.degats }} dégâts)
  </li>
</ul>
```

Passer un objet entier à un composant

Au chapitre Components, `ProjectCard` recevait une prop par champ. Tu peux aussi passer **l'objet entier**, avec une prop de type `Object` :

```xml
<!-- src/components/CreatureCard.vue -->
<script setup>
  defineProps({
    creature: {
      type: Object,
      required: true,
    },
  })
</script>

<template>
  <article class="bg-white border border-zinc-200 overflow-hidden">
    <img :src="creature.image" :alt="creature.nom" class="w-full h-48 object-cover">
    <div class="p-4">
      <h3 class="text-xl font-bold text-zinc-900">{{ creature.nom }}</h3>
      <p class="text-sm text-zinc-500">{{ creature.espece }} · niveau {{
creature.niveau }}</p>
    </div>
  </article>
</template>
```

```xml
<!-- src/App.vue -->
<script setup>
  import creatures from '@/data/creatures.json'
  import CreatureCard from '@/components/CreatureCard.vue'
</script>

<template>
  <main class="min-h-screen bg-stone-100 p-8">
    <div class="grid grid-cols-1 md:grid-cols-2 lg:grid-cols-4 gap-6">
      <CreatureCard v-for="creature in creatures" :key="creature.id"
:creature="creature" />
    </div>
  </main>
</template>
```

Dans `:creature="creature"`, à gauche le nom de la prop, à droite la variable du `v-for`.

Une prop par champ (`:title="creature.nom"`) ou l'objet entier (`:creature="creature"`) : la deuxième façon est plus courte quand la carte a besoin de presque tout l'objet.



Exercices de lecture : JSON et Vue

Deuxième paquet, sur ce texte et sur la liste `v-for` vue plus haut :

JAVASCRIPT

```javascript
const texte = '{"nom":"Brumelle","niveau":8,"faiblesses":["vent","soleil"]}'
```

XML

```xml
<script setup>
  import creatures from '@/data/creatures.json'
</script>

<template>
  <ul>
    <li v-for="creature in creatures" :key="creature.id">
      {{ creature.nom }} — niveau {{ creature.niveau }}
    </li>
  </ul>
</template>
```

`texte` est-il un objet JavaScript ?

Non. C'est du **texte** (une chaîne entre apostrophes) écrit au format JSON. Pour t'en servir comme d'un objet, il faut d'abord le convertir.

Comment le convertis-tu en objet ?

```
const brumelle = JSON.parse(texte)
```

Une fois converti, comment lis-tu le niveau ?

`brumelle.niveau` , qui vaut `8` .

Quelle différence entre `{ nom: 'Brumelle' }` et `{ "nom": "Brumelle" }` ?

Le premier est un objet JavaScript : clé sans guillemets, apostrophes acceptées. Le second est aussi du JSON valide : en JSON, les clés et les chaînes sont **toujours entre guillemets doubles**.

Cite deux choses qu'un objet JavaScript peut contenir, et un JSON jamais.

Des fonctions et des commentaires. Aussi : `undefined` , et des chaînes entre apostrophes.

Comment transformes-tu l'objet `brumelle` en texte JSON ?

```
JSON.stringify(brumelle)
```

Dans un projet Vite, comment charges-tu `src/data/creatures.json` ?

`import creatures from '@/data/creatures.json'` . Pas de `JSON.parse` : Vite le fait pour toi.

Le JSON contient 8 créatures. Combien de `<li>` le `v-for` crée-t-il ?

8, un par élément du tableau.

À quoi sert `:key="creature.id"` ?

À donner à Vue un identifiant unique et stable pour chaque élément, pour qu'il retrouve chaque ligne quand la liste change. Détail au chapitre v-for.

Pour afficher le portrait, `src="creature.image"` ou `:src="creature.image"` ?

`:src="creature.image"`. Avec les deux-points, Vue évalue l'expression et lit l'objet ; sans, le navigateur reçoit le texte `creature.image` et cherche une image qui s'appelle comme ça.

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/projet-web-uiux/vuejs/objets-et-json

10. Quand mettre les données où ?

Trois endroits possibles, selon la taille et la vie des données :

Où	Quand	Exemple
Dans le `<script setup>`	C'est petit, ça change rarement, c'est propre au composant	Les liens d'une navbar, les options d'un `<select>`
Dans un fichier `.json`	Une liste qui grandit, des données séparées du code, plusieurs composants qui la lisent	Tes projets, le bestiaire, des témoignages
Sur un serveur ou une API	Plus tard dans l'année : quand les données doivent être partagées ou modifiées par d'autres	Les messages d'un formulaire, un catalogue tenu à jour par un client

Sur un site de designer, tout ce qui se répète avec la même forme a sa place dans un JSON :

- les projets d'un portfolio ;
- une galerie d'images légendées ;
- des témoignages clients ;
- un catalogue de produits ;
- les articles d'un blog.

11. Exercices pratiques

Tout se fait dans ton projet Vue des chapitres précédents, avec Tailwind v4 branché.

1. Télécharge et dézippe bestiaire.zip ZIP : un fichier `creatures.json` et un dossier `creatures/` de huit portraits.
2. Mets `creatures/` dans `public/`, et `creatures.json` dans `src/data/` (crée le dossier).
3. Copie le contenu actuel de `App.vue` dans un fichier à part, par exemple `src/ExercicesTailwind.vue` (ta carte de profil resservira à l'exercice 2). Puis remplace-le par :

```xml
<script setup>
  // Les données de l'exercice viennent ici.
</script>

<template>
  <main class="min-h-screen bg-stone-100 p-8">
    <!-- La fiche ou la grille vient ici -->
  </main>
</template>
```

Lance `npm run dev` si ce n'est pas déjà fait.

Exercice 1 — Fiche de créature ⭐⭐

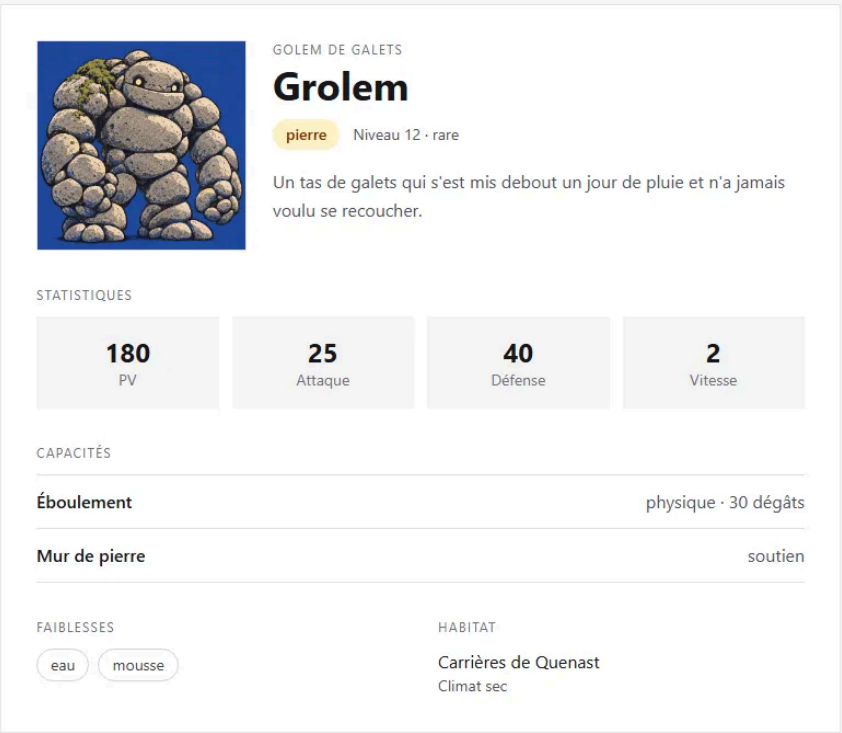
Objectif : afficher **un** objet complexe : sous-objets `stats` et `habitat`, tableaux `capacites` et `faiblesses`.

Ta mission : colle l'objet Grolem de la section 5 dans le `<script setup>`, puis construis sa fiche dans le template.

1. L'image (`:src`, `:alt`), le nom en `<h1>`, l'espèce, le type dans une pastille, le niveau et la rareté.
2. La description.
3. Les quatre stats en quatre tuiles (`grid grid-cols-4`), chacune avec son libellé et sa valeur.
4. Les capacités avec `v-for` : nom, type, dégâts. « Mur de pierre » fait 0 dégât : affiche « soutien » plutôt que « 0 dégâts », avec `v-if="capacite.degats > 0"` sur un `<span>` et `v-else` sur le suivant, comme à l'exercice 3 du chapitre Components.
5. Les faiblesses en pastilles, avec `v-for`.
6. L'habitat : région et climat.

Pour l'image : `/creatures/grolem.png`, ou une image à toi.

Rendu attendu



Critères observables

- [ ] Aucune donnée de Grolem n'est écrite en dur dans le template : tout vient de `creature`
- [ ] Les quatre stats viennent de `creature.stats`
- [ ] Les capacités et les faiblesses sont des `v-for`
- [ ] La page utilise la palette du cours : `stone`, `zinc`, un seul accent
- [ ] L'image a un `alt`

Bonus ⭐ : une barre de vie sous les stats. Un `<div>` de couleur, de hauteur fixe, dont la largeur dépend des PV : `:style="{ width: creature.stats.pv / 3 + '%' }"` (300 PV, le maximum du bestiaire, donnent 100 %). Regarde la valeur de `:style` : un objet, encore un, dont les clés sont des propriétés CSS.



GOLEM DE GALETS

Grolem

pierre Niveau 12 · rare

Un tas de galets qui s'est mis debout un jour de pluie et n'a jamais voulu se recoucher.

STATISTIQUES

180	25	40	2
PV	Attaque	Défense	Vitesse

CAPACITÉS

Éboulement	physique · 30 dégâts
Mur de pierre	soutien

FAIBLESSES

eau mousse

HABITAT

Carrières de Quenast
Climat sec

src/App.vue :

XML

```
<script setup>
  const creature = {
    nom: 'Grolem',
    espece: 'Golem de galets',
    type: 'pierre',
    niveau: 12,
    rarete: 'rare',
    description:
      "Un tas de galets qui s'est mis debout un jour de pluie et n'a jamais voulu
se recoucher.",
    image: '/creatures/grolem.png',
    stats: { pv: 180, attaque: 25, defense: 40, vitesse: 2 },
    capacites: [
      { nom: 'Éboulement', type: 'physique', degats: 30 },
      { nom: 'Mur de pierre', type: 'soutien', degats: 0 },
    ],
    faiblesses: ['eau', 'mousse'],
    habitat: { region: 'Carrières de Quenast', climat: 'sec' },
  }
</script>

<template>
  <main class="min-h-screen bg-stone-100 p-8">
    <article class="mx-auto max-w-3xl bg-white border border-zinc-200 p-8">
      <!-- En-tête : portrait + identité -->
      <div class="flex flex-col gap-6 md:flex-row">
        <img
          :src="creature.image"
          :alt="creature.nom"
          class="w-48 h-48 object-cover border border-zinc-200"
        />
        <div>
          <p class="text-xs uppercase tracking-widest text-zinc-500">{{
creature.espece }}</p>
          <h1 class="mt-1 text-4xl font-bold text-zinc-900">{{ creature.nom }}
</h1>
          <div class="mt-3 flex items-center gap-3">
            <span class="rounded-full bg-amber-100 px-3 py-1 text-sm font-medium
text-amber-900">
              {{ creature.type }}
            </span>
            <span class="text-sm text-zinc-600">
              Niveau {{ creature.niveau }} · {{ creature.rarete }}
            </span>
          </div>
          <p class="mt-4 text-zinc-600 leading-relaxed">{{ creature.description
}}</p>
        </div>
      </div>

      <!-- Les quatre stats -->
      <h2 class="mt-8 text-xs uppercase tracking-widest text-zinc-
```

```
500">Statistiques</h2>
      <div class="mt-3 grid grid-cols-4 gap-3">
        <div class="bg-stone-100 p-4 text-center">
          <p class="text-2xl font-bold text-zinc-900">{{ creature.stats.pv }}</p>
          <p class="text-sm text-zinc-500">PV</p>
        </div>
        <div class="bg-stone-100 p-4 text-center">
          <p class="text-2xl font-bold text-zinc-900">{{ creature.stats.attaque
}}</p>
          <p class="text-sm text-zinc-500">Attaque</p>
        </div>
        <div class="bg-stone-100 p-4 text-center">
          <p class="text-2xl font-bold text-zinc-900">{{ creature.stats.defense
}}</p>
          <p class="text-sm text-zinc-500">Défense</p>
        </div>
        <div class="bg-stone-100 p-4 text-center">
          <p class="text-2xl font-bold text-zinc-900">{{ creature.stats.vitesse
}}</p>
          <p class="text-sm text-zinc-500">Vitesse</p>
        </div>
      </div>

      <!-- Les capacités : un v-for sur le tableau -->
      <h2 class="mt-8 text-xs uppercase tracking-widest text-zinc-
500">Capacités</h2>
      <ul class="mt-3 divide-y divide-zinc-200 border-y border-zinc-200">
        <li
          v-for="capacite in creature.capacites"
          :key="capacite.nom"
          class="flex justify-between py-3"
        >
          <span class="font-medium text-zinc-900">{{ capacite.nom }}</span>
          <span v-if="capacite.degats > 0" class="text-zinc-600">
            {{ capacite.type }} · {{ capacite.degats }} dégâts
          </span>
          <span v-else class="text-zinc-600">soutien</span>
        </li>
      </ul>

      <div class="mt-8 grid gap-8 md:grid-cols-2">
        <div>
          <h2 class="text-xs uppercase tracking-widest text-zinc-
500">Faiblesses</h2>
          <div class="mt-3 flex gap-2">
            <span
              v-for="faiblesse in creature.faiblesses"
              :key="faiblesse"
              class="rounded-full border border-zinc-300 px-3 py-1 text-sm text-
zinc-700"
            >
              {{ faiblesse }}
            </span>
```

```
          </div>
        </div>
        <div>
          <h2 class="text-xs uppercase tracking-widest text-zinc-
500">Habitat</h2>
          <p class="mt-3 text-zinc-900">{{ creature.habitat.region }}</p>
          <p class="text-sm text-zinc-600">Climat {{ creature.habitat.climat }}
</p>
        </div>
      </div>
    </article>
  </main>
</template>
```

src/App.vue :

XML

```
<script setup>
  const creature = {
    nom: 'Grolem',
    espece: 'Golem de galets',
    type: 'pierre',
    niveau: 12,
    rarete: 'rare',
    description:
      "Un tas de galets qui s'est mis debout un jour de pluie et n'a jamais voulu
se recoucher.",
    image: '/creatures/grolem.png',
    stats: { pv: 180, attaque: 25, defense: 40, vitesse: 2 },
    capacites: [
      { nom: 'Éboulement', type: 'physique', degats: 30 },
      { nom: 'Mur de pierre', type: 'soutien', degats: 0 },
    ],
    faiblesses: ['eau', 'mousse'],
    habitat: { region: 'Carrières de Quenast', climat: 'sec' },
  }
</script>

<template>
  <main class="min-h-screen bg-stone-100 p-8">
    <article class="mx-auto max-w-3xl bg-white border border-zinc-200 p-8">
      <!-- En-tête : portrait + identité -->
      <div class="flex flex-col gap-6 md:flex-row">
        <img
          :src="creature.image"
          :alt="creature.nom"
          class="w-48 h-48 object-cover border border-zinc-200"
        />
        <div>
          <p class="text-xs uppercase tracking-widest text-zinc-500">{{
creature.espece }}</p>
          <h1 class="mt-1 text-4xl font-bold text-zinc-900">{{ creature.nom }}
</h1>
          <div class="mt-3 flex items-center gap-3">
            <span class="rounded-full bg-amber-100 px-3 py-1 text-sm font-medium
text-amber-900">
              {{ creature.type }}
            </span>
            <span class="text-sm text-zinc-600">
              Niveau {{ creature.niveau }} · {{ creature.rarete }}
            </span>
          </div>
          <p class="mt-4 text-zinc-600 leading-relaxed">{{ creature.description
}}</p>
        </div>
      </div>

      <!-- Les quatre stats -->
      <h2 class="mt-8 text-xs uppercase tracking-widest text-zinc-
```

```
500">Statistiques</h2>
      <div class="mt-3 grid grid-cols-4 gap-3">
        <div class="bg-stone-100 p-4 text-center">
          <p class="text-2xl font-bold text-zinc-900">{{ creature.stats.pv }}</p>
          <p class="text-sm text-zinc-500">PV</p>
        </div>
        <div class="bg-stone-100 p-4 text-center">
          <p class="text-2xl font-bold text-zinc-900">{{ creature.stats.attaque
}}</p>
          <p class="text-sm text-zinc-500">Attaque</p>
        </div>
        <div class="bg-stone-100 p-4 text-center">
          <p class="text-2xl font-bold text-zinc-900">{{ creature.stats.defense
}}</p>
          <p class="text-sm text-zinc-500">Défense</p>
        </div>
        <div class="bg-stone-100 p-4 text-center">
          <p class="text-2xl font-bold text-zinc-900">{{ creature.stats.vitesse
}}</p>
          <p class="text-sm text-zinc-500">Vitesse</p>
        </div>
      </div>
      <!-- Bonus : la largeur dépend des PV -->
      <div class="mt-3 h-2 bg-stone-200">
        <div class="h-2 bg-amber-500" :style="{ width: creature.stats.pv / 3 +
'%' }"></div>
      </div>

      <!-- Les capacités : un v-for sur le tableau -->
      <h2 class="mt-8 text-xs uppercase tracking-widest text-zinc-
500">Capacités</h2>
      <ul class="mt-3 divide-y divide-zinc-200 border-y border-zinc-200">
        <li
          v-for="capacite in creature.capacites"
          :key="capacite.nom"
          class="flex justify-between py-3"
        >
          <span class="font-medium text-zinc-900">{{ capacite.nom }}</span>
          <span v-if="capacite.degats > 0" class="text-zinc-600">
            {{ capacite.type }} · {{ capacite.degats }} dégâts
          </span>
          <span v-else class="text-zinc-600">soutien</span>
        </li>
      </ul>

      <div class="mt-8 grid gap-8 md:grid-cols-2">
        <div>
          <h2 class="text-xs uppercase tracking-widest text-zinc-
500">Faiblesses</h2>
          <div class="mt-3 flex gap-2">
            <span
              v-for="faiblesse in creature.faiblesses"
              :key="faiblesse"
```

```
              class="rounded-full border border-zinc-300 px-3 py-1 text-sm text-
zinc-700"
            >
              {{ faiblesse }}
            </span>
          </div>
        </div>
        <div>
          <h2 class="text-xs uppercase tracking-widest text-zinc-
500">Habitat</h2>
          <p class="mt-3 text-zinc-900">{{ creature.habitat.region }}</p>
          <p class="text-sm text-zinc-600">Climat {{ creature.habitat.climat }}
</p>
        </div>
      </div>
    </article>
  </main>
</template>
```

Exercice 2 — Le bestiaire ⭐⭐

Objectif : importer un fichier JSON, afficher une liste d'objets en cartes avec `v-for`.

Le contenu de `creatures.json` :

JSON


```
[
  {
    "id": 1,
    "nom": "Mousseron",
    "espece": "Champignon lumineux",
    "type": "plante",
    "niveau": 3,
    "rarete": "commune",
    "description": "Un petit champignon qui brille dans le noir et qui éternue des spores quand on le surprend.",
    "image": "/creatures/mousseron.png",
    "stats": { "pv": 40, "attaque": 8, "defense": 12, "vitesse": 6 },
    "capacites": [
      { "nom": "Nuage de spores", "type": "magique", "degats": 12 },
      { "nom": "Lueur apaisante", "type": "soutien", "degats": 0 }
    ],
    "faiblesses": ["feu", "sécheresse"],
    "habitat": { "region": "Forêt de Soignes", "climat": "humide" }
  },
  {
    "id": 2,
    "nom": "Grolem",
    "espece": "Golem de galets",
    "type": "pierre",
    "niveau": 12,
    "rarete": "rare",
    "description": "Un tas de galets qui s'est mis debout un jour de pluie et n'a jamais voulu se recoucher.",
    "image": "/creatures/grolem.png",
    "stats": { "pv": 180, "attaque": 25, "defense": 40, "vitesse": 2 },
    "capacites": [
      { "nom": "Éboulement", "type": "physique", "degats": 30 },
      { "nom": "Mur de pierre", "type": "soutien", "degats": 0 }
    ],
    "faiblesses": ["eau", "mousse"],
    "habitat": { "region": "Carrières de Quenast", "climat": "sec" }
  },
  {
    "id": 3,
    "nom": "Brumelle",
    "espece": "Esprit de brume",
    "type": "spectre",
    "niveau": 8,
    "rarete": "rare",
    "description": "Elle apparaît au petit matin au-dessus des étangs et disparaît dès qu'on prononce son nom.",
    "image": "/creatures/brumelle.png",
    "stats": { "pv": 60, "attaque": 18, "defense": 5, "vitesse": 20 },
    "capacites": [
      { "nom": "Souffle glacé", "type": "magique", "degats": 20 },
      { "nom": "Évanouissement", "type": "soutien", "degats": 0 }
    ],
```

```
    "faiblesses": ["vent", "soleil"],
    "habitat": { "region": "Étangs de Genval", "climat": "brumeux" }
  },
  {
    "id": 4,
    "nom": "Krakette",
    "espece": "Petit kraken d'étang",
    "type": "eau",
    "niveau": 6,
    "rarete": "commune",
    "description": "Huit bras, une mauvaise humeur, et une passion pour les bouchons
de pêche.",
    "image": "/creatures/krakette.png",
    "stats": { "pv": 70, "attaque": 15, "defense": 10, "vitesse": 9 },
    "capacites": [
      { "nom": "Fouet de tentacule", "type": "physique", "degats": 14 },
      { "nom": "Jet d'encre", "type": "magique", "degats": 10 }
    ],
    "faiblesses": ["électricité", "filet"],
    "habitat": { "region": "Lac de Louvain-la-Neuve", "climat": "tempéré" }
  },
  {
    "id": 5,
    "nom": "Fourmidable",
    "espece": "Fourmi géante",
    "type": "insecte",
    "niveau": 10,
    "rarete": "commune",
    "description": "Elle porte vingt fois son poids, et elle le fait savoir.",
    "image": "/creatures/fourmidable.png",
    "stats": { "pv": 90, "attaque": 22, "defense": 18, "vitesse": 12 },
    "capacites": [
      { "nom": "Morsure", "type": "physique", "degats": 18 },
      { "nom": "Appel de la colonie", "type": "soutien", "degats": 0 }
    ],
    "faiblesses": ["feu", "pluie"],
    "habitat": { "region": "Dunes de La Panne", "climat": "sec" }
  },
  {
    "id": 6,
    "nom": "Pyrouette",
    "espece": "Salamandre de braise",
    "type": "feu",
    "niveau": 9,
    "rarete": "rare",
    "description": "Elle danse sur les cendres chaudes et met le feu aux rideaux par
simple distraction.",
    "image": "/creatures/pyrouette.png",
    "stats": { "pv": 65, "attaque": 24, "defense": 8, "vitesse": 16 },
    "capacites": [
      { "nom": "Boule de feu", "type": "magique", "degats": 26 },
      { "nom": "Pirouette", "type": "physique", "degats": 10 }
    ],
```

```
    "faiblesses": ["eau", "glace"],
    "habitat": { "region": "Terrils de Charleroi", "climat": "chaud" }
  },
  {
    "id": 7,
    "nom": "Vieux-Chêne",
    "espece": "Tréant",
    "type": "plante",
    "niveau": 20,
    "rarete": "légendaire",
    "description": "Il a vu passer trois siècles et quatre autoroutes. Il n'est pas
pressé.",
    "image": "/creatures/vieux-chene.png",
    "stats": { "pv": 300, "attaque": 30, "defense": 35, "vitesse": 1 },
    "capacites": [
      { "nom": "Racines", "type": "physique", "degats": 28 },
      { "nom": "Sève régénérante", "type": "soutien", "degats": 0 },
      { "nom": "Chute de glands", "type": "physique", "degats": 15 }
    ],
    "faiblesses": ["feu", "hache"],
    "habitat": { "region": "Forêt de Meerdael", "climat": "tempéré" }
  },
  {
    "id": 8,
    "nom": "Ronflax",
    "espece": "Blaireau dormeur",
    "type": "bête",
    "niveau": 5,
    "rarete": "commune",
    "description": "Il dort dix-huit heures par jour et il est de très mauvaise humeur
les six autres.",
    "image": "/creatures/ronflax.png",
    "stats": { "pv": 110, "attaque": 12, "defense": 20, "vitesse": 4 },
    "capacites": [
      { "nom": "Ronflement sonique", "type": "magique", "degats": 8 },
      { "nom": "Coup de museau", "type": "physique", "degats": 14 }
    ],
    "faiblesses": ["bruit", "lumière"],
    "habitat": { "region": "Bois de Lauzelle", "climat": "tempéré" }
  }
]
```

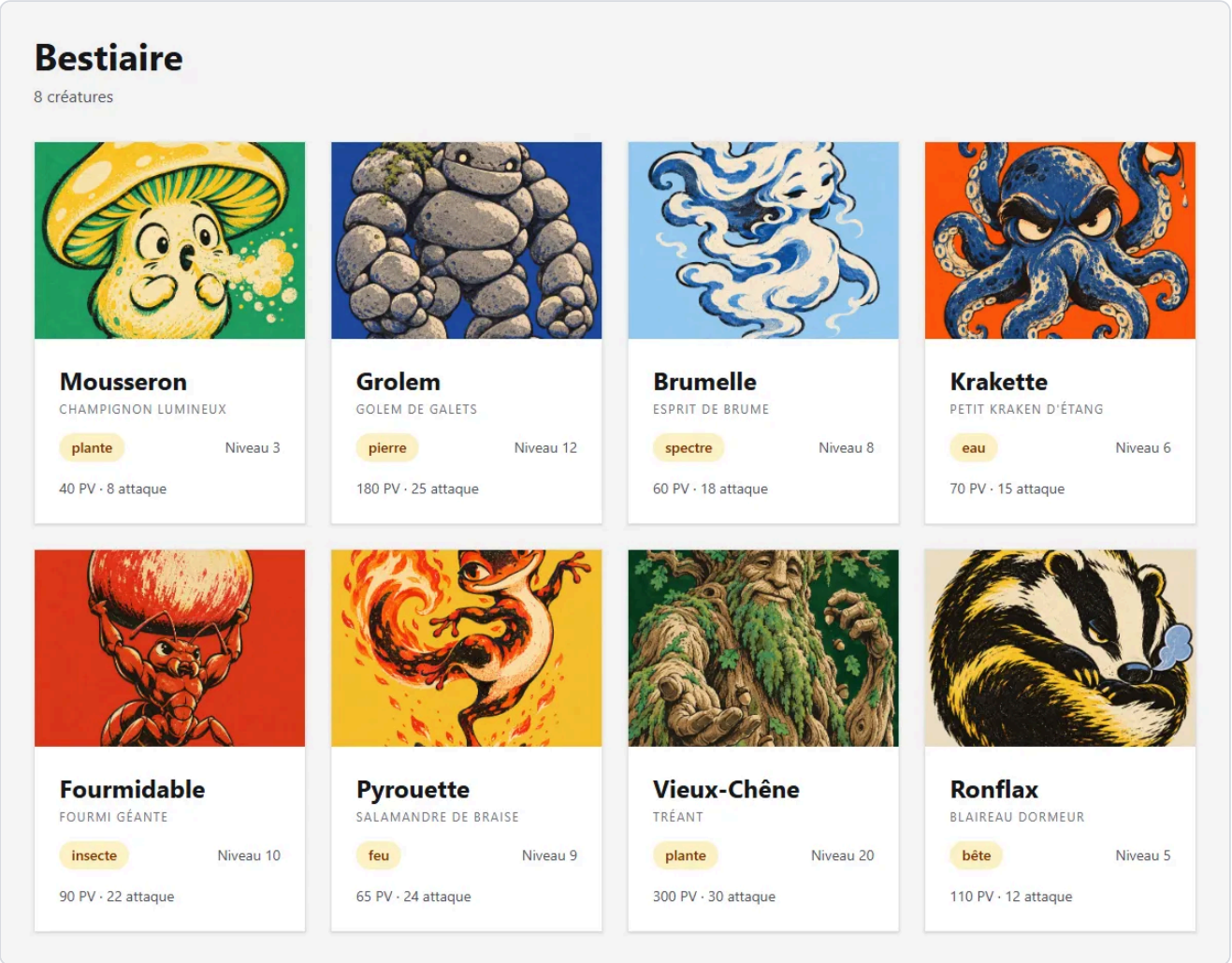

Ta mission

1. Si ce n'est pas déjà fait, télécharge **bestiaire.zip** ZIP : `creatures/` va dans `public/`, `creatures.json` dans `src/data/`.
2. Importe le JSON dans `App.vue`.
3. Affiche les huit créatures en grille de cartes : 1 colonne sur mobile, 2 sur tablette, 4 sur grand écran. Chaque carte montre l'image, le nom, l'espèce, le type en pastille, le niveau et deux stats (PV, attaque).

4. Sous le titre, une ligne indique combien de créatures sont affichées (`creatures.length`).

> **ASTUCE**
>
> Ne repars pas de zéro : recopie une carte que tu as déjà faite (ta carte de profil de « Premiers pas avec Tailwind » ou ta `ProjectCard`) et adapte ses champs.

Rendu attendu



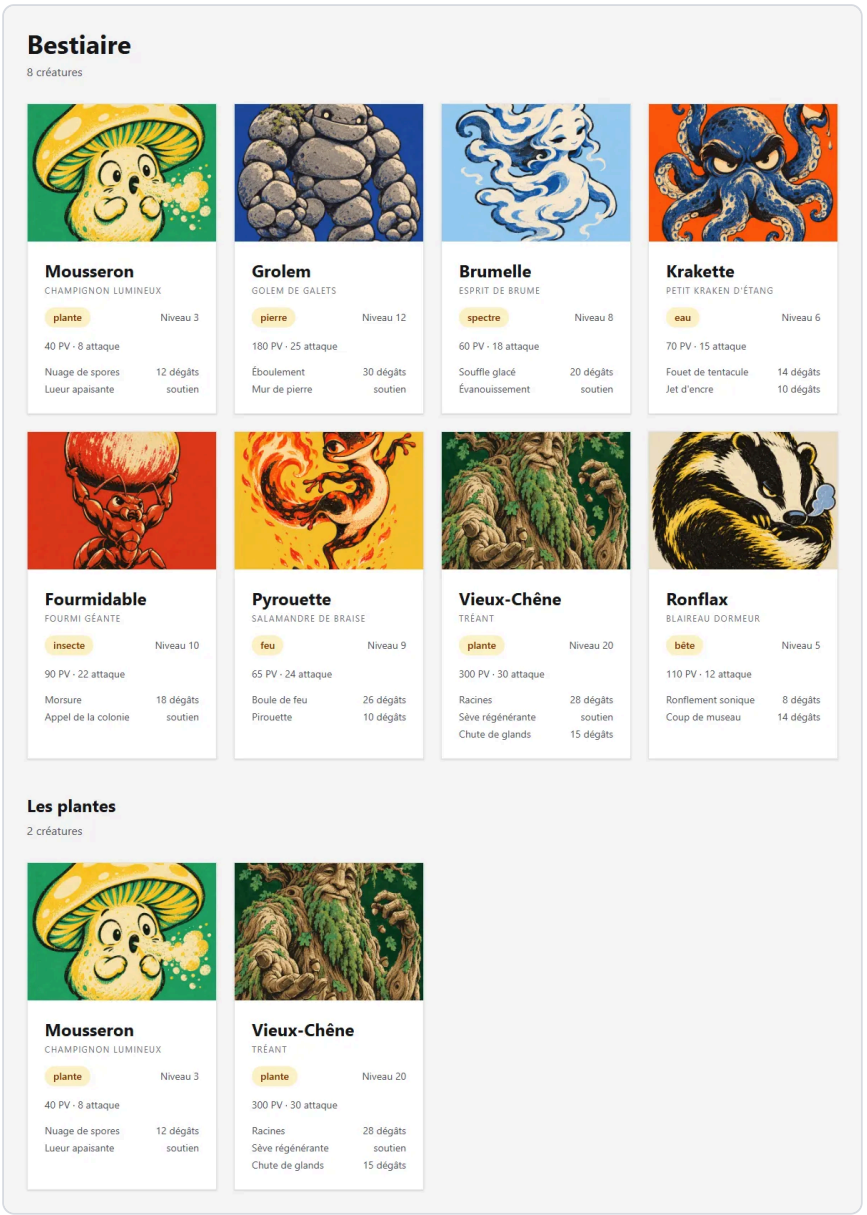
Critères observables

- [ ] Une seule carte est écrite dans le template, pas huit copies
- [ ] Chaque carte a un `:key`
- [ ] Les données viennent du JSON, pas du template
- [ ] Le `alt` de chaque image vient du JSON
- [ ] La grille passe de 1 à 2 puis à 4 colonnes

Bonus 1 ⭐ : extrais la carte dans `src/components/CreatureCard.vue` , avec une prop `creature` de type `Object` (modèle à la section 9).

Bonus 2 ⭐ : dans chaque carte, la liste des capacités de la créature : un `v-for` dans le `v-for`, avec le nom et les dégâts (ou « soutien »), comme à la section 9.

Bonus 3 ⭐⭐ : sous la grille, une section « Les plantes » qui n'affiche que les créatures de type `plante`, avec les mêmes cartes. Indice : dans le script, `const plantes = creatures.filter((creature) ⇒ creature.type === 'plante')` fabrique un second tableau (MDN — Array.filter). Compte-les aussi, avec `plantes.length`.



src/App.vue :

```XML
<script setup>
  import creatures from '@/data/creatures.json'
</script>

<template>
  <main class="min-h-screen bg-stone-100 p-8">
    <h1 class="text-4xl font-bold text-zinc-900">Bestiaire</h1>
    <p class="mt-2 text-zinc-600">{{ creatures.length }} créatures</p>

    <div class="mt-8 grid grid-cols-1 md:grid-cols-2 lg:grid-cols-4 gap-6">
      <!-- Une seule carte, répétée pour chaque créature du JSON -->
      <article
        v-for="creature in creatures"
        :key="creature.id"
        class="bg-white border border-zinc-200 overflow-hidden shadow-sm
hover:shadow-xl transition"
      >
        <img :src="creature.image" :alt="creature.nom" class="w-full h-48 object-
cover" />
        <div class="p-6">
          <h2 class="text-zinc-900 text-2xl font-bold">{{ creature.nom }}</h2>
          <p class="mt-1 text-xs uppercase tracking-widest text-zinc-500">{{
creature.espece }}</p>
          <div class="mt-4 flex items-center justify-between">
            <span class="rounded-full bg-amber-100 px-3 py-1 text-sm font-medium
text-amber-900">
              {{ creature.type }}
            </span>
            <span class="text-sm text-zinc-600">Niveau {{ creature.niveau }}
</span>
          </div>
          <p class="mt-4 text-sm text-zinc-600">
            {{ creature.stats.pv }} PV · {{ creature.stats.attaque }} attaque
          </p>
        </div>
      </article>
    </div>
  </main>
</template>
```

src/components/CreatureCard.vue :

```xml
<script setup>
  defineProps({
    creature: {
      type: Object,
      required: true,
    },
  })
</script>

<template>
  <article
    class="bg-white border border-zinc-200 overflow-hidden shadow-sm
hover:shadow-xl transition"
  >
    <img :src="creature.image" :alt="creature.nom" class="w-full h-48 object-
cover" />
    <div class="p-6">
      <h2 class="text-zinc-900 text-2xl font-bold">{{ creature.nom }}</h2>
      <p class="mt-1 text-xs uppercase tracking-widest text-zinc-500">{{
creature.espece }}</p>
      <div class="mt-4 flex items-center justify-between">
        <span class="rounded-full bg-amber-100 px-3 py-1 text-sm font-medium
text-amber-900">
          {{ creature.type }}
        </span>
        <span class="text-sm text-zinc-600">Niveau {{ creature.niveau }}</span>
      </div>
      <p class="mt-4 text-sm text-zinc-600">
        {{ creature.stats.pv }} PV · {{ creature.stats.attaque }} attaque
      </p>
      <ul class="mt-4 space-y-1 text-sm text-zinc-600">
        <li v-for="capacite in creature.capacites" :key="capacite.nom"
class="flex justify-between">
          <span>{{ capacite.nom }}</span>
          <span v-if="capacite.degats > 0">{{ capacite.degats }} dégâts</span>
          <span v-else>soutien</span>
        </li>
      </ul>
    </div>
  </article>
</template>
```

src/App.vue :

```xml
<script setup>
  import creatures from '@/data/creatures.json'
  import CreatureCard from '@/components/CreatureCard.vue'

  // Bonus 3 : un second tableau, filtré une fois
  const plantes = creatures.filter((creature) => creature.type === 'plante')
</script>

<template>
  <main class="min-h-screen bg-stone-100 p-8">
    <h1 class="text-4xl font-bold text-zinc-900">Bestiaire</h1>
    <p class="mt-2 text-zinc-600">{{ creatures.length }} créatures</p>

    <div class="mt-8 grid grid-cols-1 md:grid-cols-2 lg:grid-cols-4 gap-6">
      <CreatureCard v-for="creature in creatures" :key="creature.id"
:creature="creature" />
    </div>

    <h2 class="mt-12 text-2xl font-bold text-zinc-900">Les plantes</h2>
    <p class="mt-2 text-zinc-600">{{ plantes.length }} créatures</p>

    <div class="mt-8 grid grid-cols-1 md:grid-cols-2 lg:grid-cols-4 gap-6">
      <CreatureCard v-for="creature in plantes" :key="creature.id"
:creature="creature" />
    </div>
  </main>
</template>
```

`plantes` est calculé une fois, dans le script, puis affiché exactement comme `creatures`. Pour une autre section, il suffit de changer le type dans le `filter`.

Exercice 3 — Ma playlist ⭐

Objectif : écrire toi-même un fichier JSON, puis l'afficher.

Ta mission : crée `src/data/playlist.json` avec **au moins cinq morceaux** que tu aimes vraiment. Chaque morceau est un objet avec ces clés :

Clé	Type	Exemple
`id`	Nombre	`1`
`titre`	Chaîne	`"Around the World"`
`artiste`	Chaîne	`"Daft Punk"`
`duree`	Chaîne	`"7:09"`

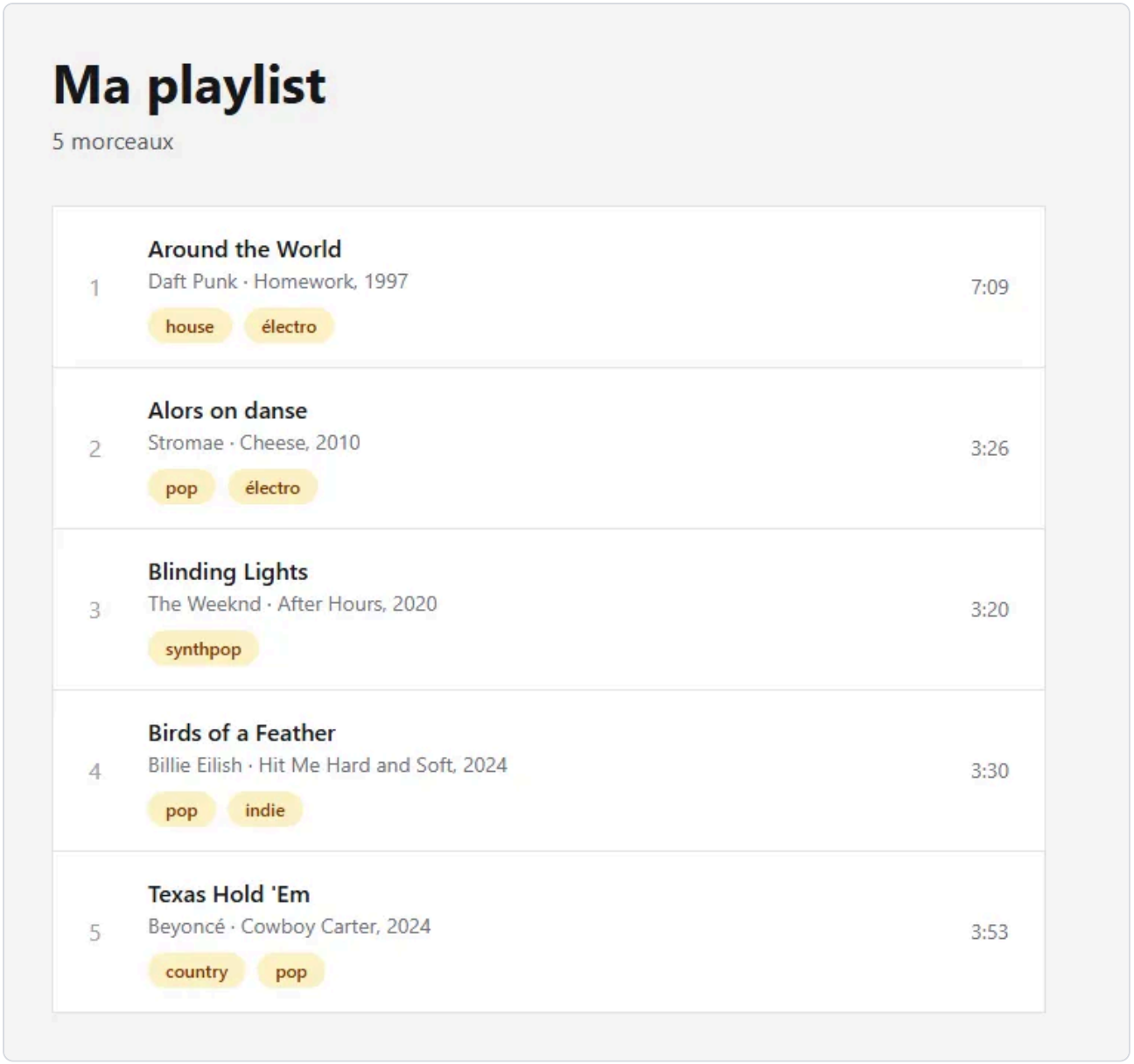
Clé	Type	Exemple
`album`	Objet	`{ "nom": "Homework", "annee": 1997 }`
`genres`	Tableau de chaînes	`["house", "électro"]`

Puis, dans `App.vue` :

1. Importe le fichier.
2. Affiche la playlist en **liste numérotée** : un `<ol>`, un seul `<li>` en `v-for`. Le numéro, c'est l'`id`. Dans chaque ligne : le titre, l'artiste, l'album et son année (`morceau.album.nom`, `morceau.album.annee`), la durée.
3. Les genres en pastilles, avec un second `v-for`.
4. Sous le titre, le nombre de morceaux.

> **ASTUCE**
>
> Les trois erreurs JSON classiques : une virgule après le dernier morceau, une clé sans guillemets, une apostrophe à la place d'un guillemet. VS Code souligne en rouge, et Vite te donne la ligne dans son message d'erreur.

Rendu attendu (avec une playlist d'exemple : la tienne sera différente)



Critères observables

- [ ] Le JSON est valide : rien en rouge dans VS Code, pas d'erreur dans le terminal
- [ ] Un seul `<li>` est écrit dans le template
- [ ] L'album et son année viennent du sous-objet `album`
- [ ] Les genres sortent d'un second `v-for`
- [ ] Les morceaux sont les tiens

Bonus ⭐ : si tu tiens aux pochettes, ajoute une clé `pochette` à chaque morceau, une image carrée dans `public/pochettes/`, et affiche-la en `w-16 h-16 object-cover` au début de la ligne, avec `:src` et `:alt`.

src/data/playlist.json :

```json
[
  {
    "id": 1,
    "titre": "Around the World",
    "artiste": "Daft Punk",
    "duree": "7:09",
    "album": { "nom": "Homework", "annee": 1997 },
    "genres": ["house", "électro"]
  },
  {
    "id": 2,
    "titre": "Alors on danse",
    "artiste": "Stromae",
    "duree": "3:26",
    "album": { "nom": "Cheese", "annee": 2010 },
    "genres": ["pop", "électro"]
  },
  {
    "id": 3,
    "titre": "Blinding Lights",
    "artiste": "The Weeknd",
    "duree": "3:20",
    "album": { "nom": "After Hours", "annee": 2020 },
    "genres": ["synthpop"]
  },
  {
    "id": 4,
    "titre": "Birds of a Feather",
    "artiste": "Billie Eilish",
    "duree": "3:30",
    "album": { "nom": "Hit Me Hard and Soft", "annee": 2024 },
    "genres": ["pop", "indie"]
  },
  {
    "id": 5,
    "titre": "Texas Hold 'Em",
    "artiste": "Beyoncé",
    "duree": "3:53",
    "album": { "nom": "Cowboy Carter", "annee": 2024 },
    "genres": ["country", "pop"]
  }
]
```

src/App.vue :

```xml
<script setup>
  import playlist from '@/data/playlist.json'
</script>

<template>
  <main class="min-h-screen bg-stone-100 p-8">
    <h1 class="text-4xl font-bold text-zinc-900">Ma playlist</h1>
    <p class="mt-2 text-zinc-600">{{ playlist.length }} morceaux</p>

    <ol class="mt-8 max-w-2xl bg-white border border-zinc-200 divide-y divide-
zinc-200">
      <li v-for="morceau in playlist" :key="morceau.id" class="flex items-center
gap-4 px-6 py-4">
        <span class="w-6 text-zinc-400 tabular-nums">{{ morceau.id }}</span>
        <div class="flex-1">
          <p class="font-medium text-zinc-900">{{ morceau.titre }}</p>
          <p class="text-sm text-zinc-500">
            {{ morceau.artiste }} · {{ morceau.album.nom }}, {{
morceau.album.annee }}
          </p>
          <!-- Un v-for dans le v-for : les genres -->
          <div class="mt-2 flex flex-wrap gap-2">
            <span
              v-for="genre in morceau.genres"
              :key="genre"
              class="rounded-full bg-amber-100 px-3 py-1 text-xs font-medium
text-amber-900"
            >
              {{ genre }}
            </span>
          </div>
        </div>
        <span class="text-sm text-zinc-500 tabular-nums">{{ morceau.duree }}
</span>
      </li>
    </ol>
  </main>
</template>
```

Récap

- Un objet regroupe les informations d'une même chose : des paires `clé: valeur` entre accolades.
- On lit avec le point (`creature.nom`), avec des crochets quand la clé est dans une variable.
- `=` modifie ou ajoute, `delete` retire. `const` n'empêche pas de modifier le contenu.

- Les accès s'enchaînent de gauche à droite : `creature.capacites[0].nom`. Une clé absente vaut `undefined` ; aller plus loin plante.
- JSON est du texte aux règles strictes : guillemets doubles, pas de virgule finale, pas de fonction. `JSON.parse` et `JSON.stringify` font le pont.
- Dans Vite, `import creatures from '@/data/creatures.json'` donne directement le tableau d'objets.
- Dans Vue : `{{ creature.nom }}` entre deux balises, `:src="creature.image"` dans un attribut, `v-for` et `:key` pour une liste, une prop `Object` pour passer un objet entier. Modifier l'objet, c'est le chapitre Ref.

3.10 La Réactivité Vue : ref() et Données Dynamiques

Tes composants sont des cadavres statiques pour l'instant. Ils affichent des trucs, ok, mais ils ne **réagissent** à rien. Clique sur un bouton ? Rien. Survole un élément ? Nada. C'est triste. 💀

On va leur donner vie avec **ref()**, le cœur battant de Vue.

Objectifs du chapitre

À la fin de ce chapitre, tu seras capable de :

- ✅ Comprendre ce qu'est la réactivité dans Vue
- ✅ Créer des données réactives avec `ref()`
- ✅ Modifier ces données en réponse à des actions utilisateur
- ✅ Maîtriser la règle `.value` (script) vs sans `.value` (template)

C'est quoi la réactivité ?



En JavaScript classique, quand tu changes une variable, il ne se passe **rien** à l'écran. Le navigateur ne sait pas qu'il doit mettre à jour l'affichage.

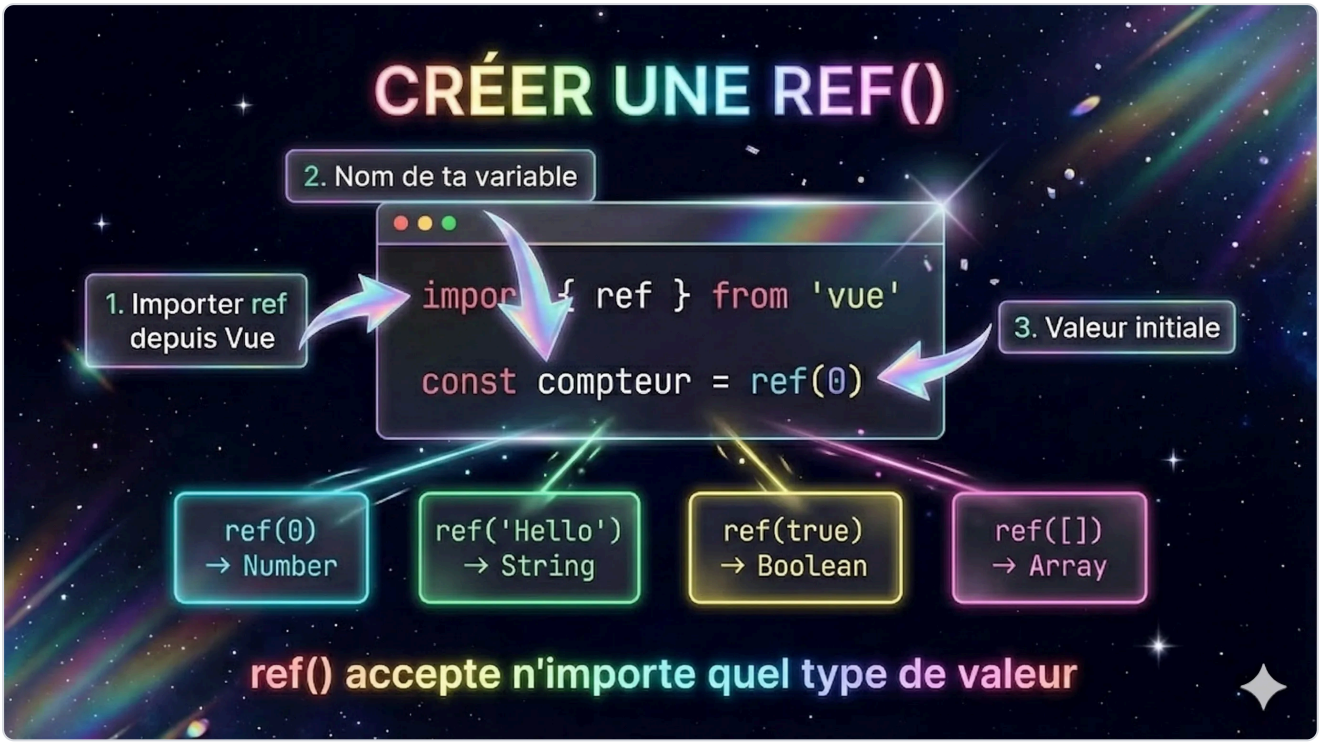
```javascript
// JavaScript classique
let compteur = 0
compteur++
// L'écran affiche toujours 0... 😢
```

Avec Vue et `ref()`, c'est différent. Vue **surveille** la donnée. Dès qu'elle change, Vue met à jour l'affichage automatiquement.

```javascript
// Vue avec ref()
const compteur = ref(0)
compteur.value++
// L'écran affiche 1 instantanément ! ✨
```

C'est ça la réactivité : Vue réagit aux changements de données et synchronise l'affichage tout seul.

Créer une ref()



Syntaxe de base

```XML
<script setup>
import { ref } from 'vue'

// Créer des refs avec différents types de valeurs
const compteur = ref(0)            // Number
const message = ref('Hello')       // String
const isVisible = ref(true)        // Boolean
const tags = ref(['Vue', 'CSS'])   // Array
</script>
```

Deux étapes obligatoires :

1. **Importer** `ref` depuis `'vue'`
2. **Appeler** `ref()` avec ta valeur initiale

Type	Exemple	Résultat
Number	`ref(0)`	Compteur, quantité, index
String	`ref('Hello')`	Texte, message, nom
Boolean	`ref(true)`	État on/off, visible/caché
Array	`ref([])`	Liste d'éléments

Type	Exemple	Résultat
Object	`ref({})`	Données structurées

La règle d'or : .value



C'est **LA** source d'erreur numéro 1 avec `ref()`. Grave ça dans ton cerveau :

Contexte	Syntaxe	Pourquoi
Dans `<script>`	**Avec** `.value`	Tu accèdes à la vraie valeur
Dans `<template>`	**Sans** `.value`	Vue déballe automatiquement

Exemple concret

```XML
<script setup>
import { ref } from 'vue'

const compteur = ref(0)

function incrementer() {
  // ✅ Dans le script : .value obligatoire
  compteur.value++

  // ❌ FAUX : ça ne marchera pas
  // compteur++
}

function afficherValeur() {
  // ✅ Pour lire aussi, .value
  console.log(compteur.value)
}
</script>

<template>
  <!— ✅ Dans le template : PAS de .value —>
  <p>Compteur : {{ compteur }}</p>

  <!— ❌ FAUX : ça afficherait l'objet ref, pas la valeur —>
  <!— <p>{{ compteur.value }}</p> —>

  <button @click="incrementer">+1</button>
</template>
```

Erreurs fréquentes

❌ Erreur	✅ Correction
`compteur++` dans script	`compteur.value++`
`{ { compteur.value }}` dans template	`{ { compteur }}`
`if (isVisible)` dans script	`if (isVisible.value)`
Oublier l'import	`import { ref } from 'vue'`

Modifier une ref avec des événements



Pour que quelque chose se passe quand l'utilisateur interagit, tu as besoin de :

1. **Un événement** (`@click`, `@mouseover`, etc.)
2. **Une fonction** qui modifie la ref
3. Vue s'occupe du reste !

Les événements de base

Événement	Déclencheur	Exemple
`@click`	Clic souris	Bouton, lien
`@mouseover`	Survol	Tooltip, preview
`@mouseout`	Quitte l'élément	Fermer tooltip
`@keyup`	Touche relâchée	Recherche live
`@keyup.enter`	Touche Entrée	Valider formulaire



Exemple : Toggle visibility

Un pattern ultra-courant : afficher/cacher un élément.

```XML
<script setup>
import { ref } from 'vue'

const isMenuOpen = ref(false)

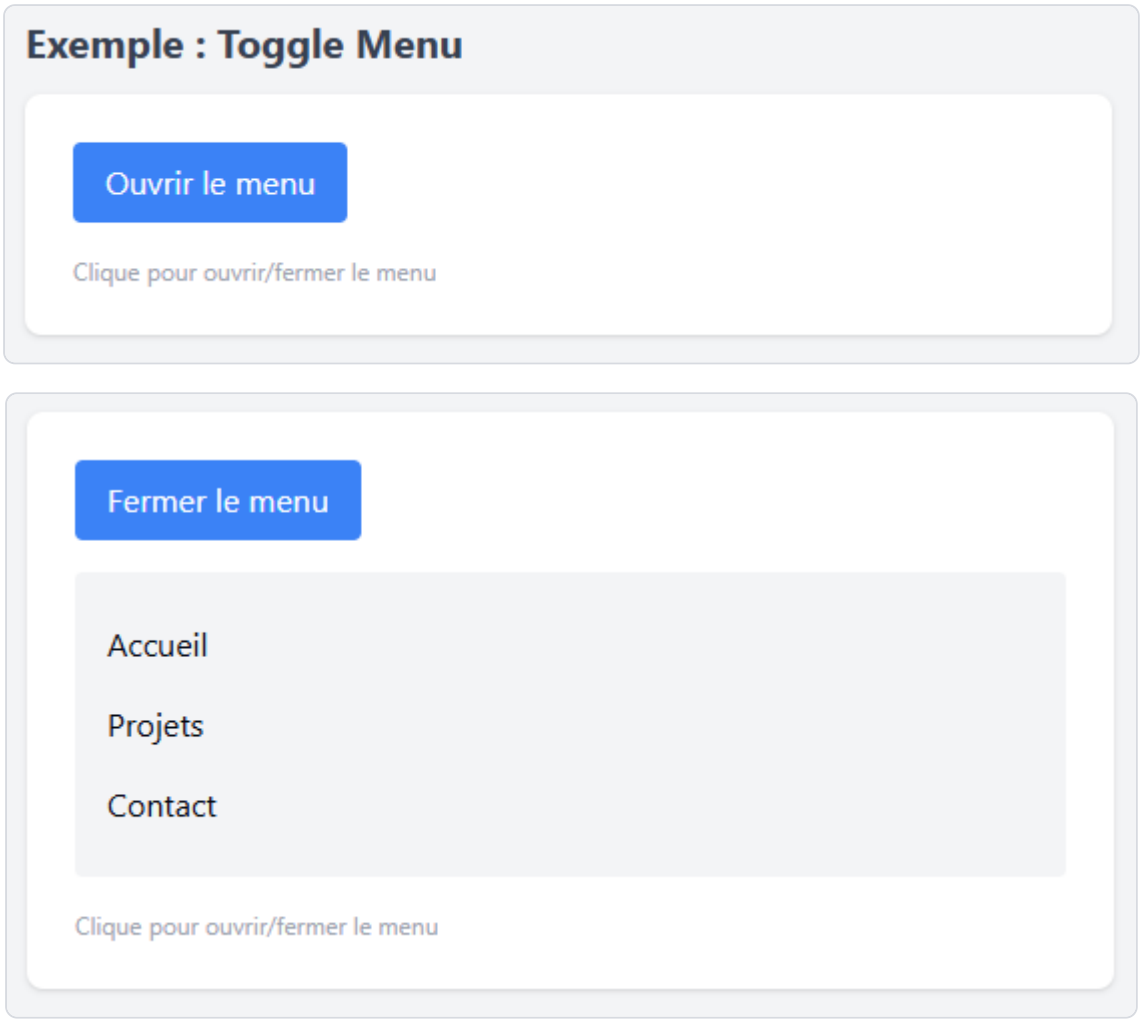
function toggleMenu() {
  isMenuOpen.value = !isMenuOpen.value
}
</script>

<template>
  <div>
    <button @click="toggleMenu" class="bg-blue-500 text-white px-4 py-2 rounded">
      {{ isMenuOpen ? 'Fermer' : 'Ouvrir' }} le menu
    </button>

    <nav v-if="isMenuOpen" class="mt-4 p-4 bg-gray-100 rounded">
      <a href="#" class="block py-2">Accueil</a>
      <a href="#" class="block py-2">Projets</a>
      <a href="#" class="block py-2">Contact</a>
    </nav>
  </div>
</template>
```

Décortiquons :

1. `isMenuOpen` commence à `false` (menu fermé)
2. Au clic, `toggleMenu()` inverse la valeur (`false` → `true`)
3. Vue voit le changement et re-render
4. `v-if="isMenuOpen"` affiche maintenant le menu
5. Le texte du bouton change aussi grâce au ternaire



Exemple : Compteur avec +/-

```xml
<script setup>
import { ref } from 'vue'

const count = ref(0)

function increment() {
  count.value++
}

function decrement() {
  count.value--
}

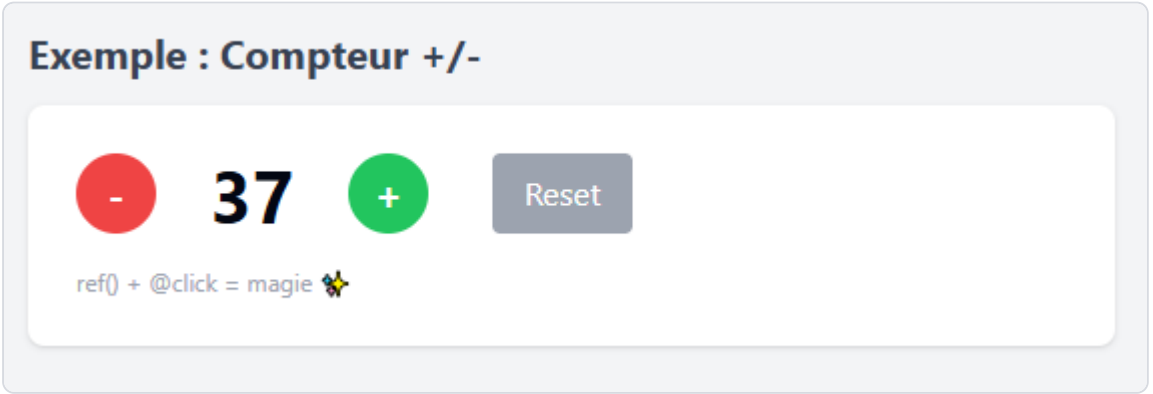
function reset() {
  count.value = 0
}
</script>

<template>
  <div class="flex items-center gap-4">
    <button @click="decrement" class="bg-red-500 text-white w-10 h-10 rounded-full
text-xl">
      -
    </button>

    <span class="text-4xl font-bold w-16 text-center">{{ count }}</span>

    <button @click="increment" class="bg-green-500 text-white w-10 h-10 rounded-full
text-xl">
      +
    </button>

    <button @click="reset" class="bg-gray-400 text-white px-4 py-2 rounded ml-4">
      Reset
    </button>
  </div>
</template>
```



Affichage conditionnel : v-if



Maintenant que tu sais créer des refs booléennes (`ref(true)` / `ref(false)`), tu peux **afficher ou cacher** des éléments selon leur valeur.

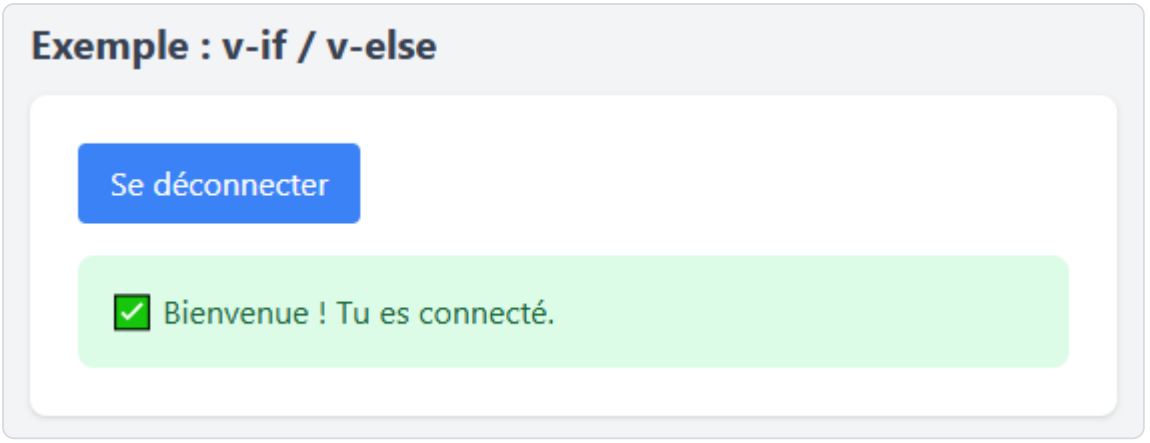
v-if : Afficher si vrai

```XML
<script setup>
import { ref } from 'vue'

const isLoggedIn = ref(false)
</script>

<template>
  <div v-if="isLoggedIn">
    Bienvenue ! Tu es connecté.
  </div>
</template>
```

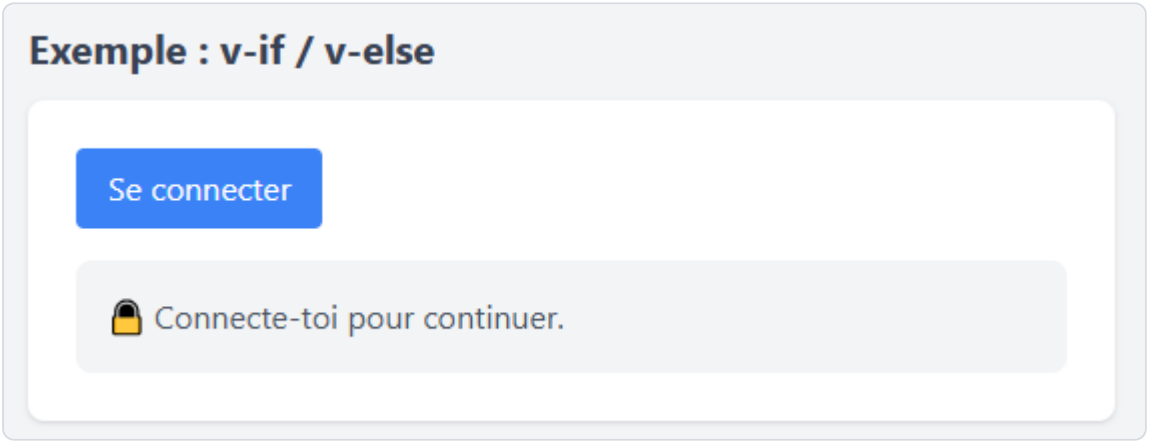
Comportement : Si `isLoggedIn` est `false`, l'élément **n'existe pas** dans le DOM. Il est complètement supprimé.



v-else : L'alternative

```xml
<template>
  <div v-if="isLoggedIn">
    Bienvenue ! Tu es connecté.
  </div>
  <div v-else>
    Connecte-toi pour continuer.
  </div>
</template>
```

⚠️ **Important** : `v-else` doit être **immédiatement après** un `v-if`. Pas d'élément entre les deux !

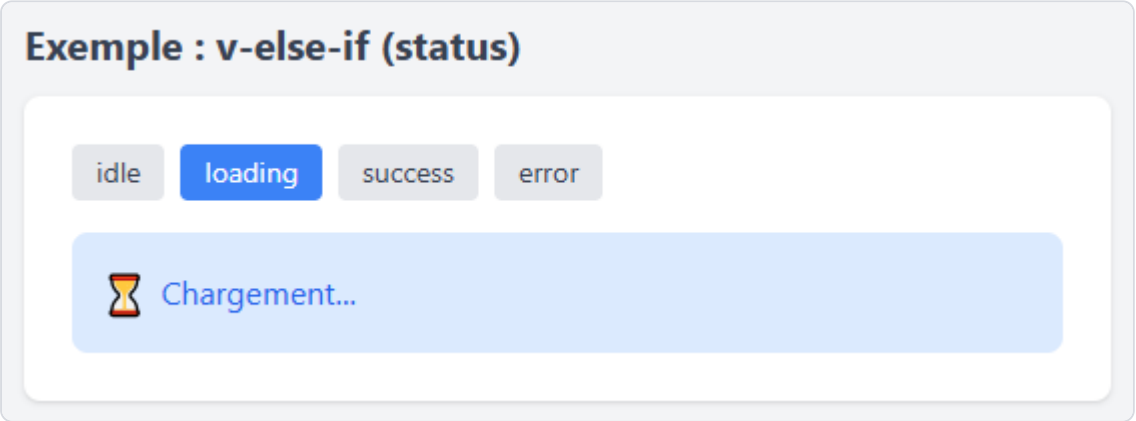


v-else-if : Plusieurs conditions

```xml
<script setup>
import { ref } from 'vue'

const status = ref('loading') // 'loading', 'success', 'error'
</script>

<template>
  <div v-if="status === 'loading'" class="text-blue-500">
    ⏳ Chargement...
  </div>
  <div v-else-if="status === 'success'" class="text-green-500">
    ✅ Succès !
  </div>
  <div v-else-if="status === 'error'" class="text-red-500">
    ❌ Erreur !
  </div>
  <div v-else class="text-gray-500">
    🤷 État inconnu
  </div>
</template>
```



v-if vs v-show : Quelle différence ?

Directive	Comportement	Utilisation
`v-if`	**Supprime** l'élément du DOM	Changements **rares**
`v-show`	**Cache** avec `display: none`	Changements **fréquents**

```xml
<template>
  <!-- v-if : l'élément est SUPPRIMÉ si false -->
  <div v-if="isVisible">Je suis dans le DOM ou pas</div>

  <!-- v-show : l'élément est CACHÉ mais reste dans le DOM -->
  <div v-show="isVisible">Je suis toujours là, juste invisible</div>
</template>
```

Règle simple :

- Menu qui s'ouvre/ferme souvent → `v-show`
- Contenu qui change rarement (login/logout) → `v-if`
- Pas sûr ? → `v-if` (c'est le défaut)

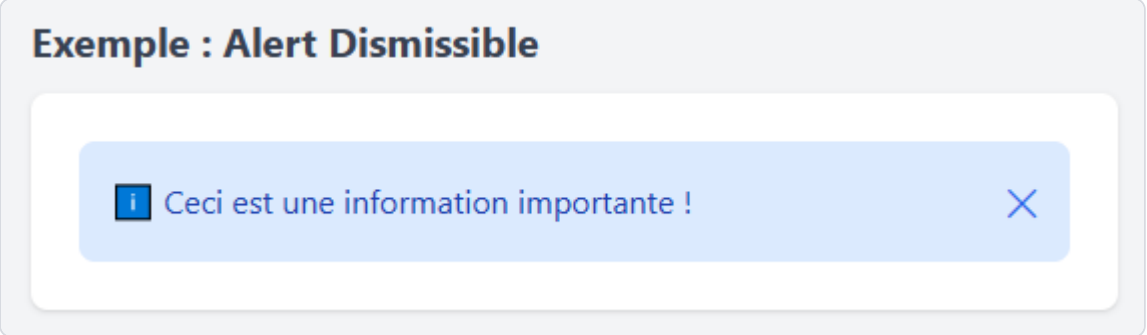
Exemple complet : Alert dismissible

```xml
<script setup>
import { ref } from 'vue'

const showAlert = ref(true)

function dismissAlert() {
  showAlert.value = false
}
</script>

<template>
  <div
    v-if="showAlert"
    class="bg-blue-100 text-blue-800 p-4 rounded-lg flex justify-between items-center"
  >
    <span>ℹ️ Ceci est une information importante !</span>
    <button @click="dismissAlert" class="text-blue-600 hover:text-blue-800">
      ×
    </button>
  </div>
</template>
```



Exercices pratiques

Exercice 1 : Toggle Show/Hide ⭐

Objectif : Maîtriser le pattern toggle de base



Ta mission : Créer un composant `Spoiler.vue` qui masque/révèle du contenu.

Comportement :

- Par défaut, le contenu est caché
- Un bouton "Voir le spoiler" / "Masquer" toggle l'affichage
- Le texte du bouton change selon l'état

Props à implémenter :

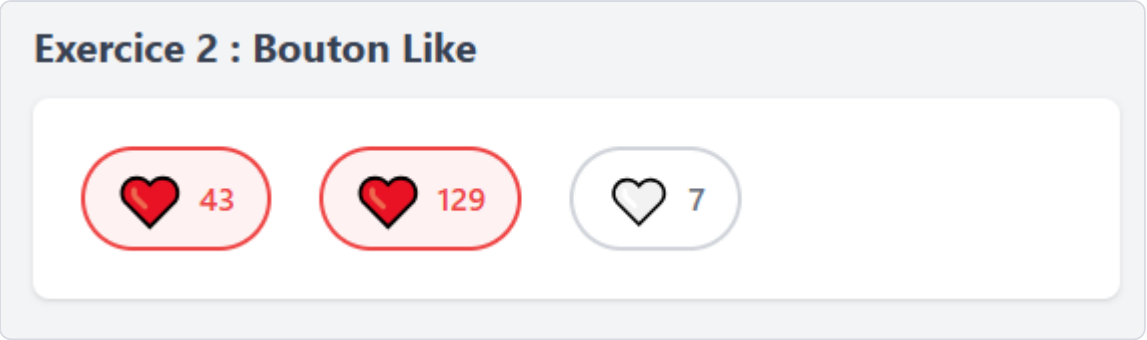
Prop	Type	Description
`title`	String	Titre affiché (ex: "Spoiler saison 3")
`content`	String	Le contenu caché

Exemple d'utilisation :

```xml
<Spoiler
  title="Fin du film"
  content="Le majordome était le coupable depuis le début."
/>
```

Exercice 2 : Bouton Like ⭐⭐

Objectif : Combiner toggle, compteur et classes dynamiques



Ta mission : Créer un composant `LikeButton.vue` avec :

- Un cœur cliquable (🤍 gris / ❤️ rouge)
- Un compteur de likes
- Au clic : toggle like + mise à jour compteur

Props à implémenter :

Prop	Type	Description
`initialCount`	Number	Nombre de likes initial (défaut: 0)

Comportement :

- Clic 1 : cœur rouge, compteur +1
- Clic 2 : cœur gris, compteur -1
- Style différent si liké (bordure rouge, fond rose clair)

Exemple d'utilisation :

```xml
<LikeButton :initialCount="42" />
```

Exercice 3 : Navigation Responsive ⭐⭐

Objectif : Créer un menu hamburger animé





Ta mission : Créer un composant `NavBar.vue` avec :

- Un logo à gauche
- Des liens de navigation (desktop : visibles, mobile : cachés)
- Un bouton hamburger (☰) visible uniquement sur mobile
- Au clic : le menu s'ouvre/ferme avec une animation

Props à implémenter :

Prop	Type	Description
`logo`	String	Texte du logo
`links`	Array	Liste des liens `[{ label: 'Accueil', href: '/' }, ...]`

Comportement :

- Desktop (>768px) : liens en ligne, pas de hamburger
- Mobile : hamburger visible, menu caché par défaut
- Clic hamburger : menu slide de haut en bas
- Icône toggle : ☰ (fermé) → ✕ (ouvert)

Exemple d'utilisation :

```xml
<NavBar
  logo="MonSite"
  :links="[
    { label: 'Accueil', href: '/' },
    { label: 'Projets', href: '/projets' },
    { label: 'Contact', href: '/contact' }
  ]"
/>
```

Exercice 4 : Notation 5 étoiles ⭐⭐⭐

Objectif : Gérer le hover et le clic sur une liste



Ta mission : Créer un composant `StarRating.vue` avec :

- 5 étoiles cliquables (☆ vide / ★ pleine)
- Au survol : preview de la note
- Au clic : la note est enregistrée
- Affichage de la note actuelle

Props à implémenter :

Prop	Type	Description
`initialRating`	Number	Note initiale (0-5, défaut: 0)

Comportement :

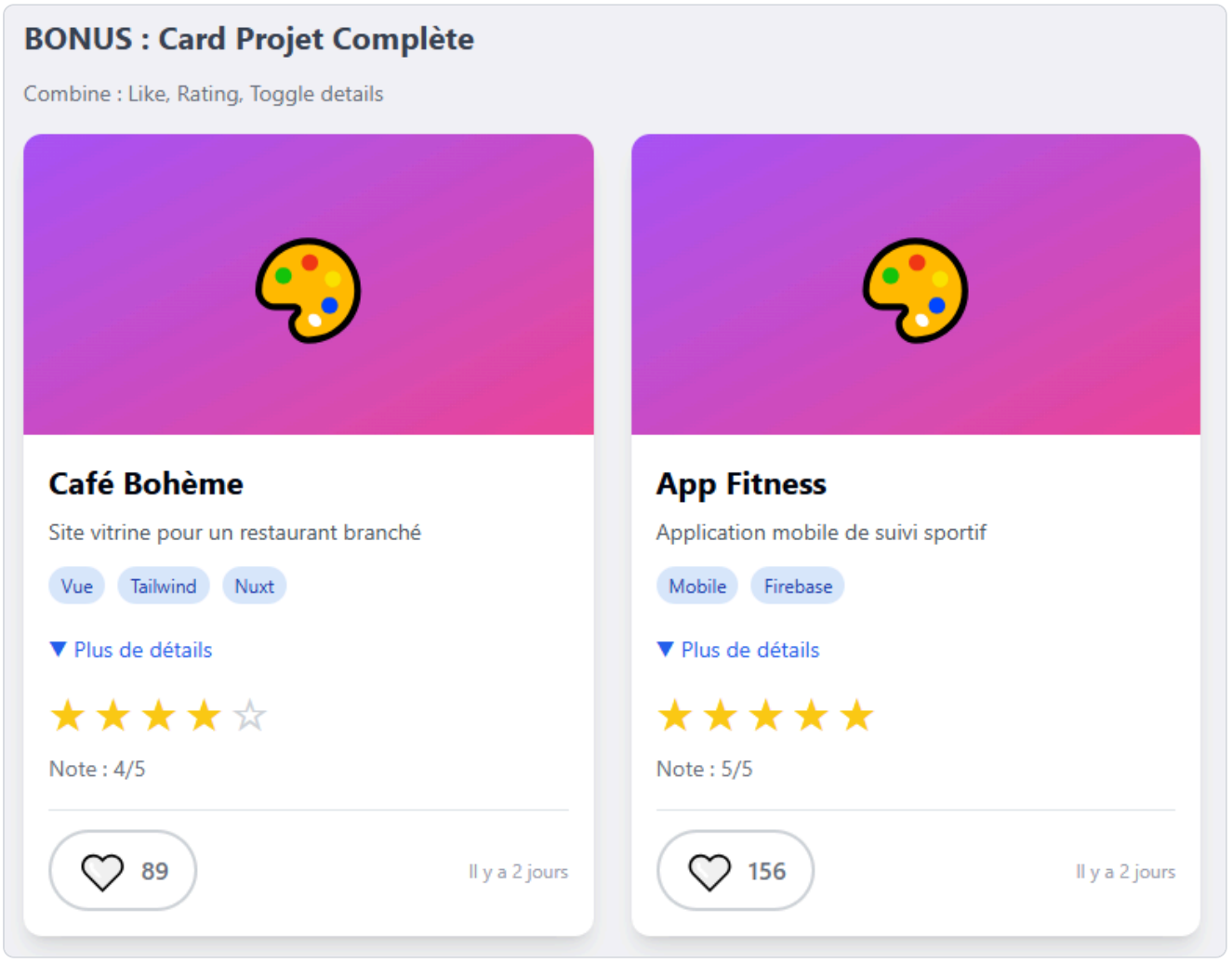
- Hover sur étoile 3 : étoiles 1, 2, 3 se remplissent (preview)
- Mouse out : revient à la note enregistrée
- Clic sur étoile 3 : note = 3 (enregistré)
- Les étoiles remplies sont jaunes/dorées

Exemple d'utilisation :

```xml
<StarRating :initialRating="3" />
```

Exercice BONUS : Card Projet Complète ⭐⭐⭐

Objectif : Combiner tous les concepts dans un composant réaliste



Ta mission : Reprendre ton `ProjectCard.vue` du chapitre précédent et y ajouter :

- Le composant `LikeButton` intégré
- Le composant `StarRating` pour noter le projet
- Un bouton "Voir détails" qui toggle une description longue (comme `Spoiler`)

Exemple d'utilisation :

```xml
<ProjectCard
  title="Café Bohème"
  description="Site vitrine pour un restaurant"
  image="/images/cafe.jpg"
  :tags="['Vue', 'Tailwind']"
  :initialLikes="42"
  :initialRating="4"
/>
```

La suite...

Tu sais maintenant créer des données réactives et les modifier. Mais pour l'instant, c'est toi (le code) qui décide quand ça change.

Dans le prochain chapitre, on verra **v-model** : l'utilisateur tape dans un input, et la donnée se met à jour **automatiquement**. C'est la magie du two-way binding, et c'est ce qui rend les formulaires Vue si agréables à coder. (ﾉ◕ヮ◕)ﾉ

3.11 Les Formulaires Vue : v-model et Two-Way Binding

Jusqu'ici, tes données vont dans **un seul sens** : du script vers le template. C'est bien, mais quand l'utilisateur tape dans un input, comment tu récupères ce qu'il a écrit ? 🤔

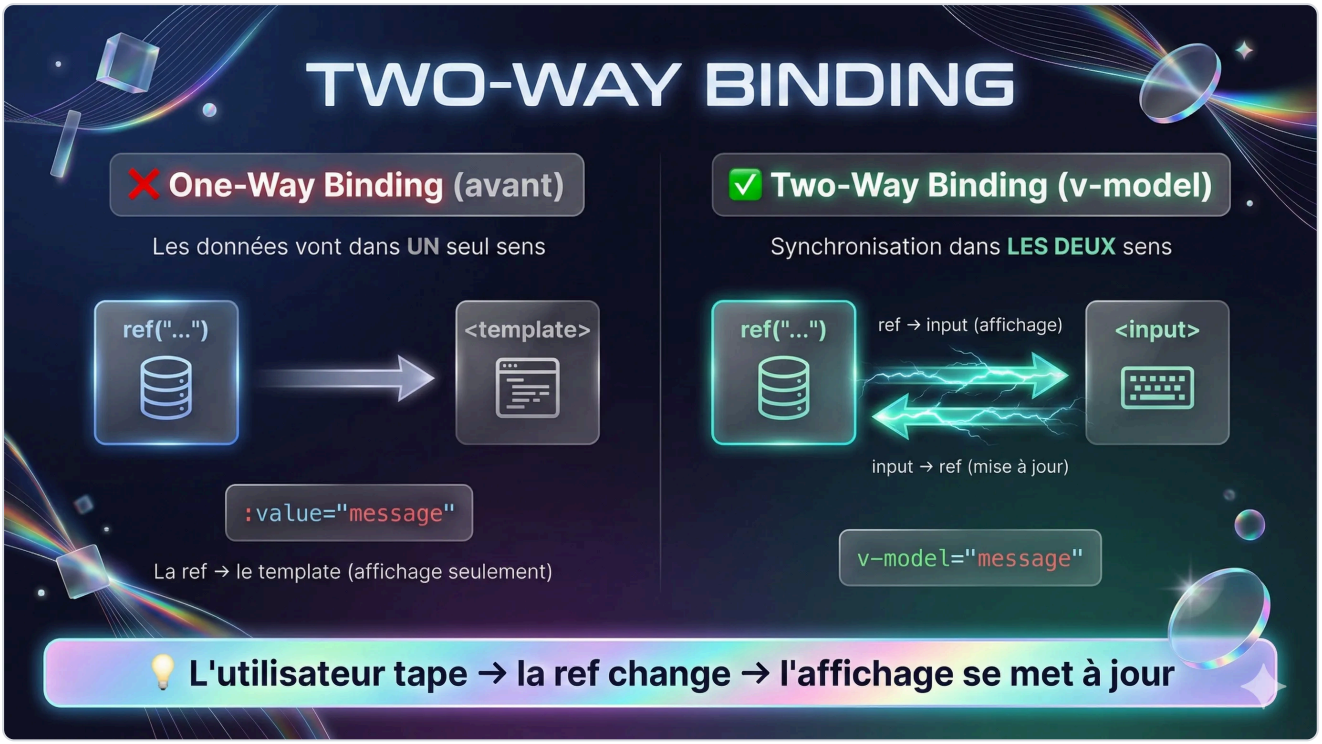
Avec `v-model`, Vue crée une **synchronisation magique** entre l'input et ta ref. L'utilisateur tape → la ref se met à jour. La ref change → l'input se met à jour. C'est le **two-way binding**.

Objectifs du chapitre

À la fin de ce chapitre, tu seras capable de :

- ✅ Comprendre le two-way binding (liaison bidirectionnelle)
- ✅ Utiliser `v-model` sur tous les types d'inputs
- ✅ Créer des formulaires interactifs
- ✅ Filtrer des données selon les choix de l'utilisateur

C'est quoi le Two-Way Binding ?



One-way binding (ce qu'on a fait jusqu'ici) :

```
ref → template (affichage uniquement)
```

Two-way binding (avec v-model) :

```
ref ↔ input (synchronisation dans les deux sens)
```

Sans v-model (la méthode longue)

```XML
<script setup>
import { ref } from 'vue'

const message = ref('')

function updateMessage(event) {
  message.value = event.target.value
}
</script>

<template>
  <input :value="message" @input="updateMessage" />
</template>
```

Ça marche, mais c'est verbeux. 😩

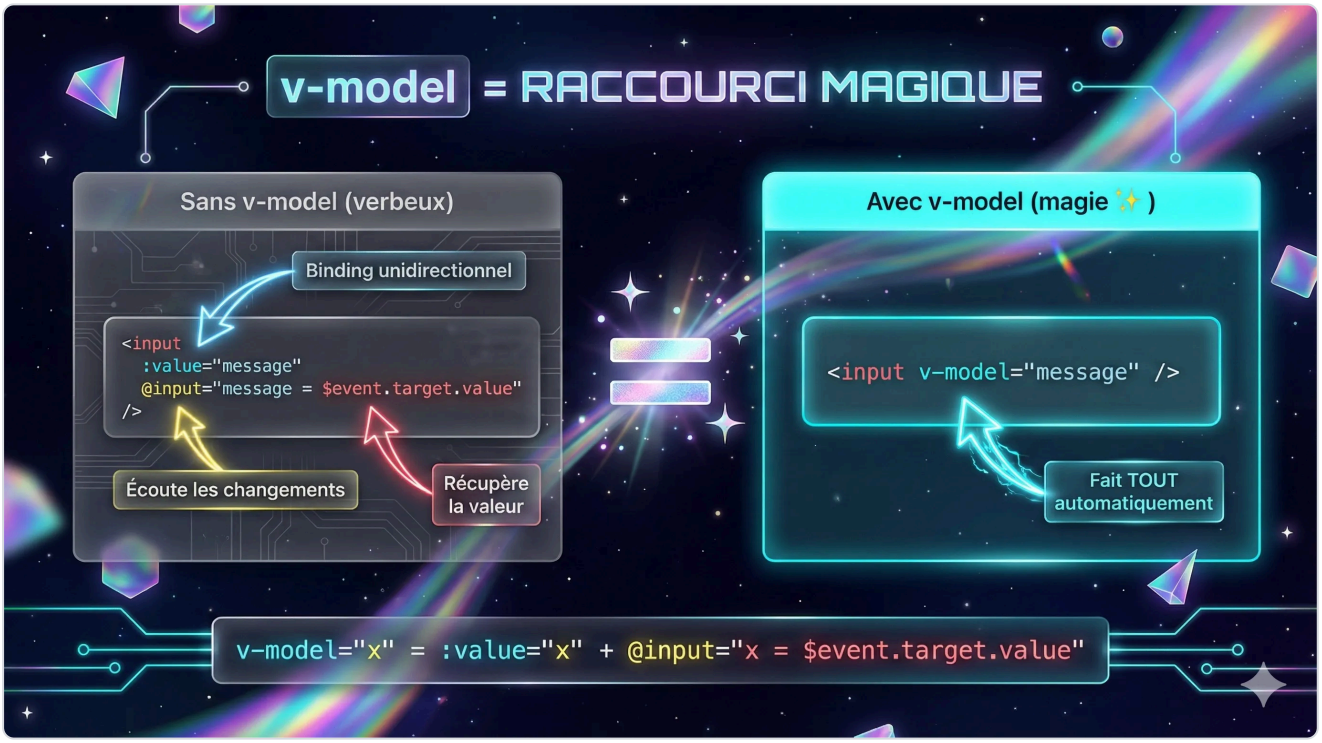
Avec v-model (la magie)

```XML
<script setup>
import { ref } from 'vue'

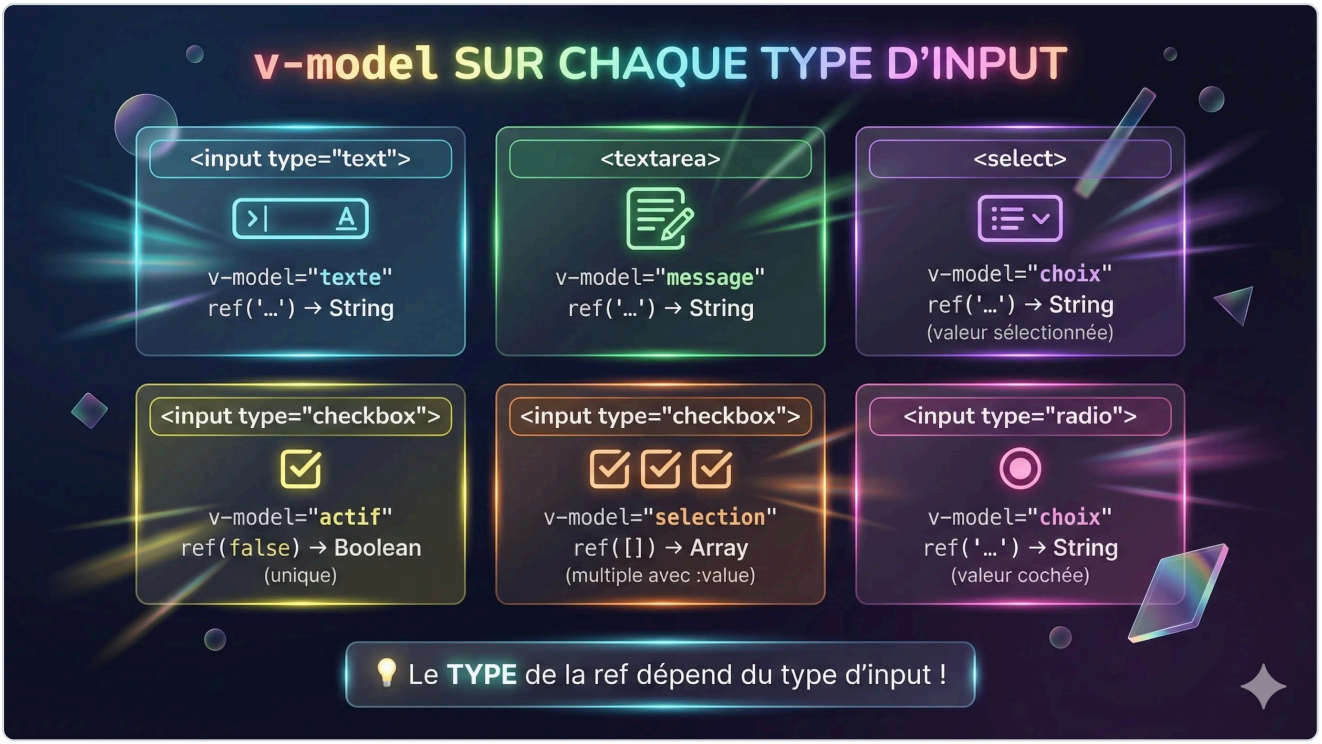
const message = ref('')
</script>

<template>
  <input v-model="message" />
</template>
```

Une ligne. C'est tout. Vue fait le `:value` et le `@input` pour toi automatiquement.



V-model sur les différents types d'inputs



Input Text : Le plus courant

Exemple : Preview live

```XML
<script setup>
import { ref } from 'vue'

const username = ref('')
</script>

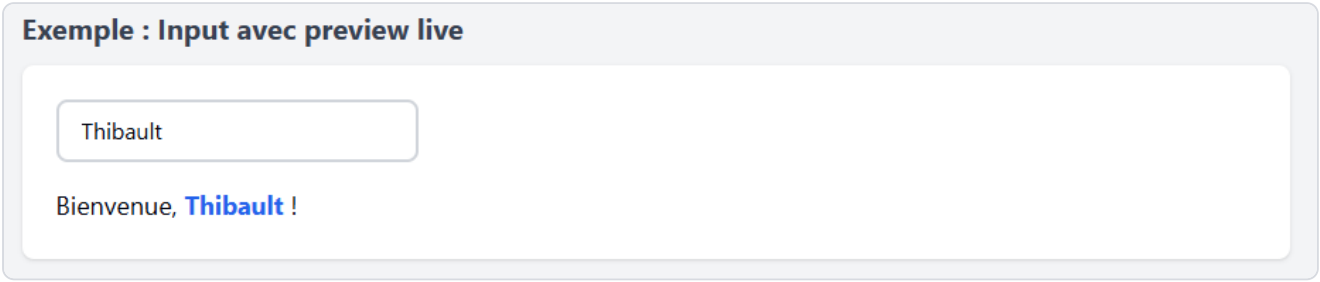
<template>
  <div>
    <input
      v-model="username"
      placeholder="Entre ton pseudo"
      class="border rounded px-3 py-2"
    />

    <p class="mt-4">
      Bienvenue, <strong>{{ username || '...' }}</strong> !
    </p>
  </div>
</template>
```

Ce qui se passe :

1. L'utilisateur tape "Alice"
2. `username.value` devient "Alice" automatiquement
3. Le `{{ username }}` dans le template se met à jour
4. Tout ça sans écrire une seule fonction !

Exemple : Input avec preview live



Textarea : Pareil mais multilignes

```xml
<script setup>
import { ref } from 'vue'

const bio = ref('')
</script>

<template>
  <textarea
    v-model="bio"
    placeholder="Parle-nous de toi..."
    rows="4"
    class="border rounded px-3 py-2 w-full"
  ></textarea>

  <p class="text-sm text-gray-500">
    {{ bio.length }} caractères
  </p>
</template>
```

⚠️ **Attention** : Pour `<textarea>`, on utilise `v-model`, pas le contenu entre les balises !

```xml
<!-- ❌ FAUX -->
<textarea>{{ bio }}</textarea>

<!-- ✅ CORRECT -->
<textarea v-model="bio"></textarea>
```

Exemple : Textarea avec compteur

Slt sa va bien?

15 caractères

Select : Les dropdowns

Exemple : Sélection simple

XML

```xml
<script setup>
import { ref } from 'vue'

const selectedColor = ref('')
</script>

<template>
  <select v-model="selectedColor" class="border rounded px-3 py-2">
    <option value="">-- Choisir une couleur --</option>
    <option value="red">Rouge</option>
    <option value="green">Vert</option>
    <option value="blue">Bleu</option>
  </select>

  <p v-if="selectedColor">
    Tu as choisi : {{ selectedColor }}
  </p>
</template>
```

Exemple : Select (dropdown)

Vert

Tu as choisi : Vert

Exemple : Options dynamiques (avec v-for)

```XML
<script setup>
import { ref } from 'vue'

const selectedCountry = ref('')
const countries = ['France', 'Belgique', 'Suisse', 'Canada']
</script>

<template>
  <select v-model="selectedCountry" class="border rounded px-3 py-2">
    <option value="">-- Choisir un pays --</option>
    <option v-for="country in countries" :key="country" :value="country">
      {{ country }}
    </option>
  </select>
</template>
```

Checkbox : Cases à cocher

Checkbox unique (toggle on/off)

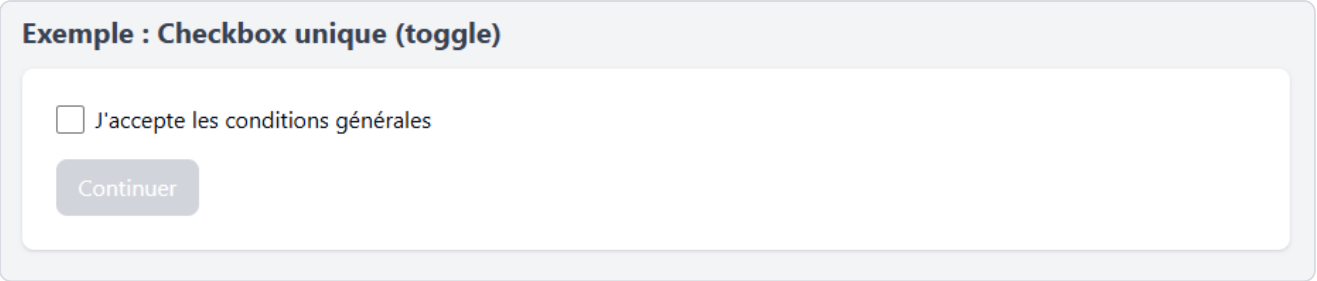
Quand tu as **une seule** checkbox, `v-model` stocke un **booléen**.

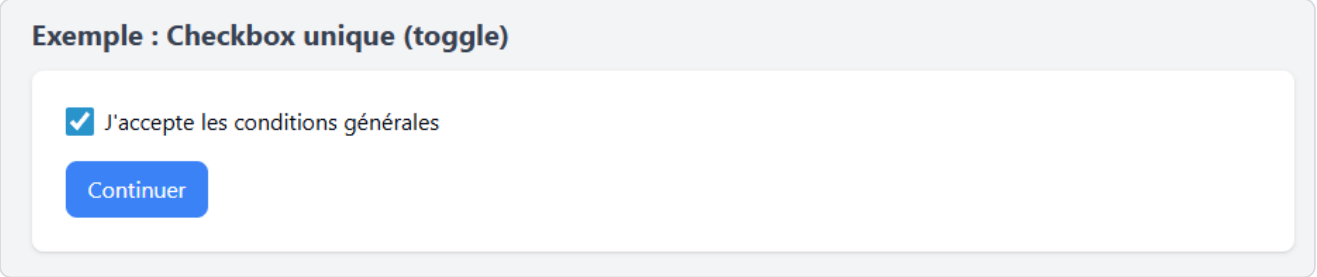
```XML
<script setup>
import { ref } from 'vue'

const acceptTerms = ref(false)
</script>

<template>
  <label class="flex items-center gap-2">
    <input type="checkbox" v-model="acceptTerms" />
    J'accepte les conditions générales
  </label>

  <button
    :disabled="!acceptTerms"
    :class="acceptTerms ? 'bg-blue-500' : 'bg-gray-300'"
    class="mt-4 px-4 py-2 rounded text-white"
  >
    Continuer
  </button>
</template>
```





Checkboxes multiples (sélection)

Quand tu as **plusieurs** checkboxes liées au même `v-model`, il stocke un **tableau**.

```XML
<script setup>
import { ref } from 'vue'

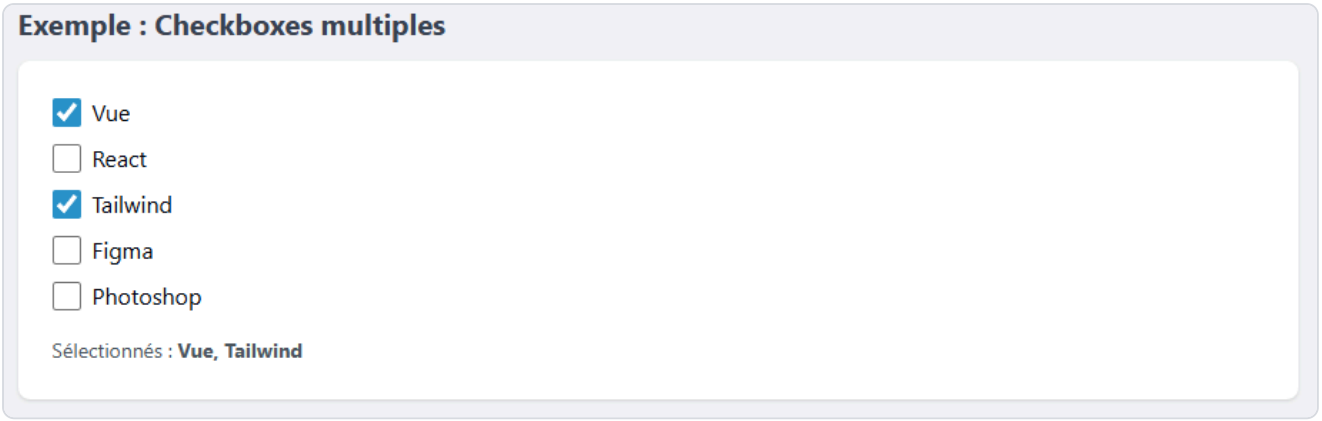
const selectedSkills = ref([])
const skills = ['Vue', 'React', 'Tailwind', 'Figma', 'Photoshop']
</script>

<template>
  <div class="space-y-2">
    <label v-for="skill in skills" :key="skill" class="flex items-center gap-2">
      <input type="checkbox" :value="skill" v-model="selectedSkills" />
      {{ skill }}
    </label>
  </div>

  <p class="mt-4">
    Sélectionnés : {{ selectedSkills.join(', ') || 'Aucun' }}
  </p>
</template>
```

Important : Chaque checkbox a un `:value` différent, mais elles partagent le même `v-model`.

Exemple : Checkboxes multiples



Radio : Choix unique

Les radio buttons servent quand l'utilisateur doit choisir **une seule option** parmi plusieurs.

XML

```
<script setup>
import { ref } from 'vue'

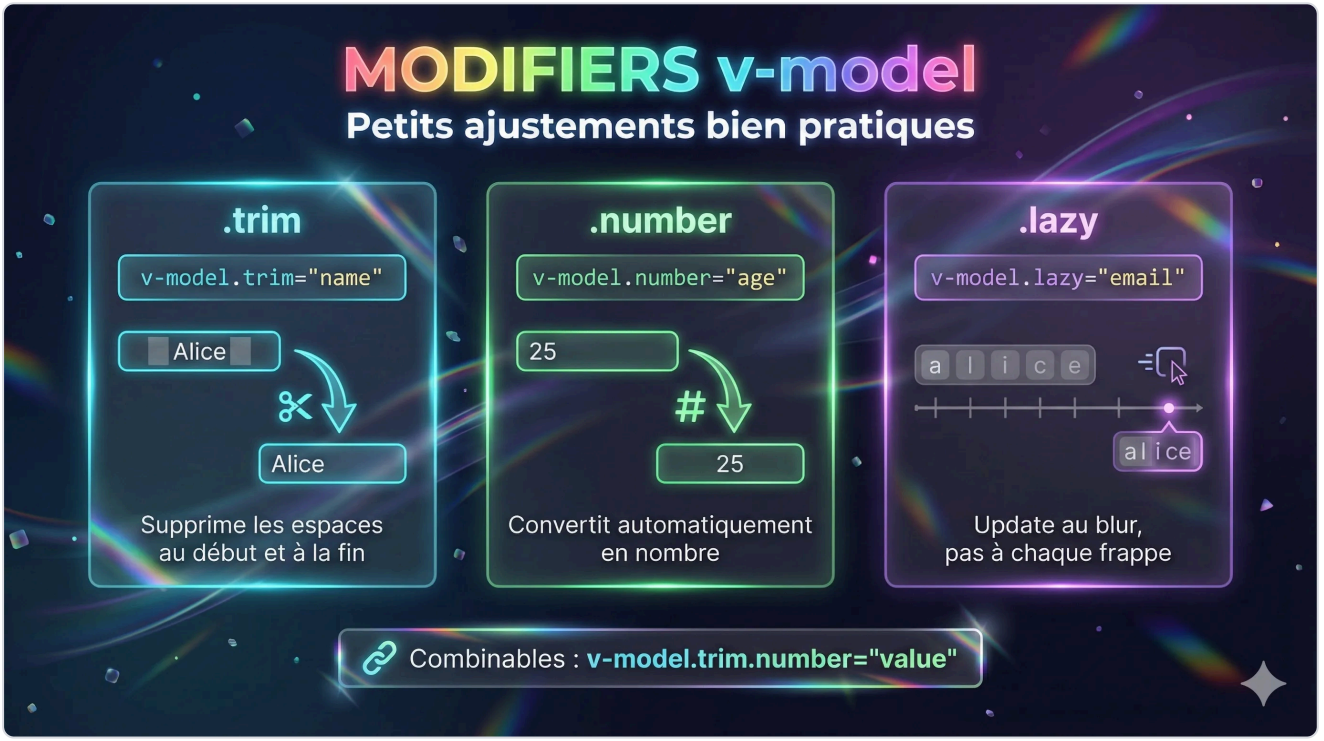
const selectedPlan = ref('')
const plans = [
  { id: 'free', name: 'Gratuit', price: '0€' },
  { id: 'pro', name: 'Pro', price: '9€/mois' },
  { id: 'enterprise', name: 'Enterprise', price: '29€/mois' }
]
</script>

<template>
  <div class="space-y-2">
    <label
      v-for="plan in plans"
      :key="plan.id"
      class="flex items-center gap-2 p-3 border rounded cursor-pointer hover:bg-gray-
50"
      :class="{ 'border-blue-500 bg-blue-50': selectedPlan === plan.id }"
    >
      <input type="radio" :value="plan.id" v-model="selectedPlan" />
      <span class="font-medium">{{ plan.name }}</span>
      <span class="text-gray-500">— {{ plan.price }}</span>
    </label>
  </div>
</template>
```

Exemple : Radio buttons

- Gratuit - 0€
- Pro - 9€/mois
- Enterprise - 29€/mois

Les Modifiers : Petits bonus utiles



Modifier	Effet	Exemple
`.trim`	Supprime les espaces début/fin	`v-model.trim="name"`
`.number`	Convertit en nombre	`v-model.number="age"`
`.lazy`	Update au `blur` au lieu de `input`	`v-model.lazy="email"`

Exemples

```xml
<script setup>
import { ref } from 'vue'

const name = ref('')
const age = ref(0)
const email = ref('')
</script>

<template>
  <!-- .trim : pas d'espaces parasites -->
  <input v-model.trim="name" placeholder="Nom" />

  <!-- .number : toujours un nombre, pas une string -->
  <input v-model.number="age" type="number" placeholder="Âge" />

  <!-- .lazy : update seulement quand on quitte le champ -->
  <input v-model.lazy="email" type="email" placeholder="Email" />
</template>
```

Importer des données JSON avec Vite

Pour les exercices suivants, tu auras besoin d'importer des données depuis un fichier JSON. Avec Vite, c'est ultra simple !

Créer le fichier JSON

```json
// src/data/products.json
[
  {
    "id": 1,
    "name": "MacBook Pro",
    "category": "laptop",
    "price": 2399
  },
  {
    "id": 2,
    "name": "iPhone 15",
    "category": "phone",
    "price": 999
  }
]
```

Importer dans ton composant

```xml
<script setup>
// Import direct du JSON - Vite gère ça automatiquement !
import products from '@/data/products.json'

// products est maintenant un tableau utilisable
console.log(products) // [{ id: 1, name: "MacBook Pro", ... }, ...]
</script>
```

C'est tout ! Pas de `fetch`, pas de `async/await`, Vite transforme le JSON en objet JavaScript directement.

Exercices pratiques

Exercice 1 : Live Preview Card ⭐

Objectif : Créer une carte de visite qui se met à jour en temps réel

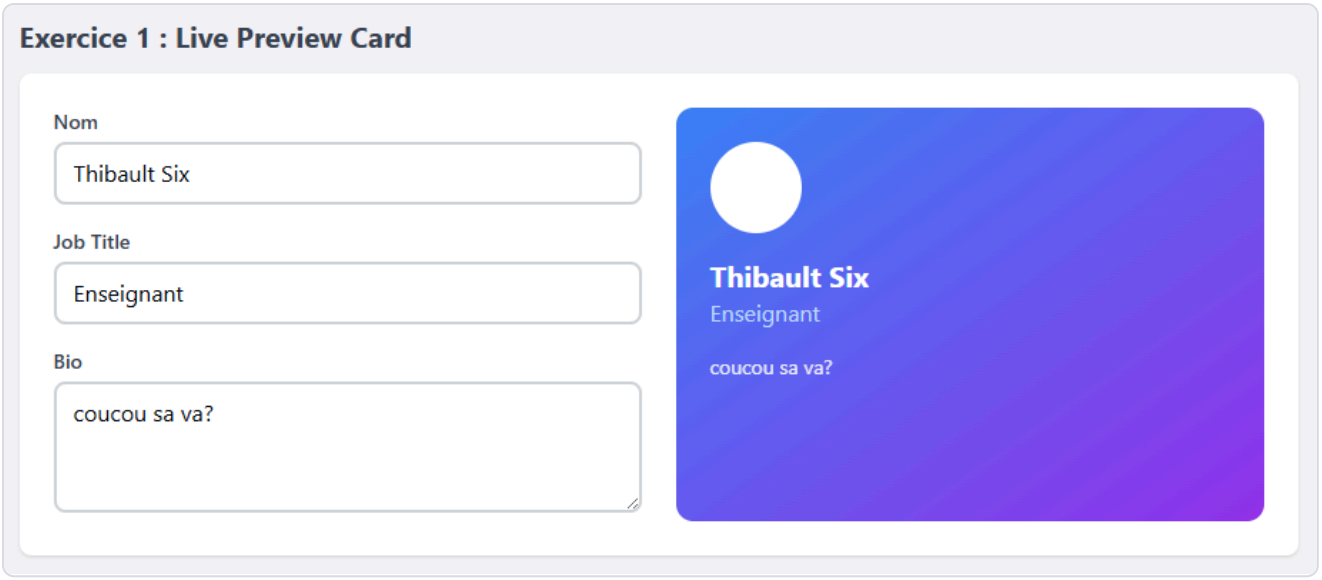
Ta mission : Créer un composant `BusinessCard.vue` avec :

- Un input pour le nom
- Un input pour le job title
- Un textarea pour la bio
- Une preview de la carte qui se met à jour en live

Exemple d'utilisation :

```xml
<BusinessCard />
```

Rendu attendu :



Exercice 2 : Theme Switcher ⭐⭐

Objectif : Changer le thème de la page avec un select

Ta mission : Créer un composant `ThemeSwitcher.vue` avec :

- Un `<select>` avec 3 thèmes : Light, Dark, Colorful
- Le fond et les couleurs de texte changent selon le thème sélectionné
- Un aperçu avec du contenu (titre, paragraphe, bouton)

Coup de pouce - Structure suggérée :

XML

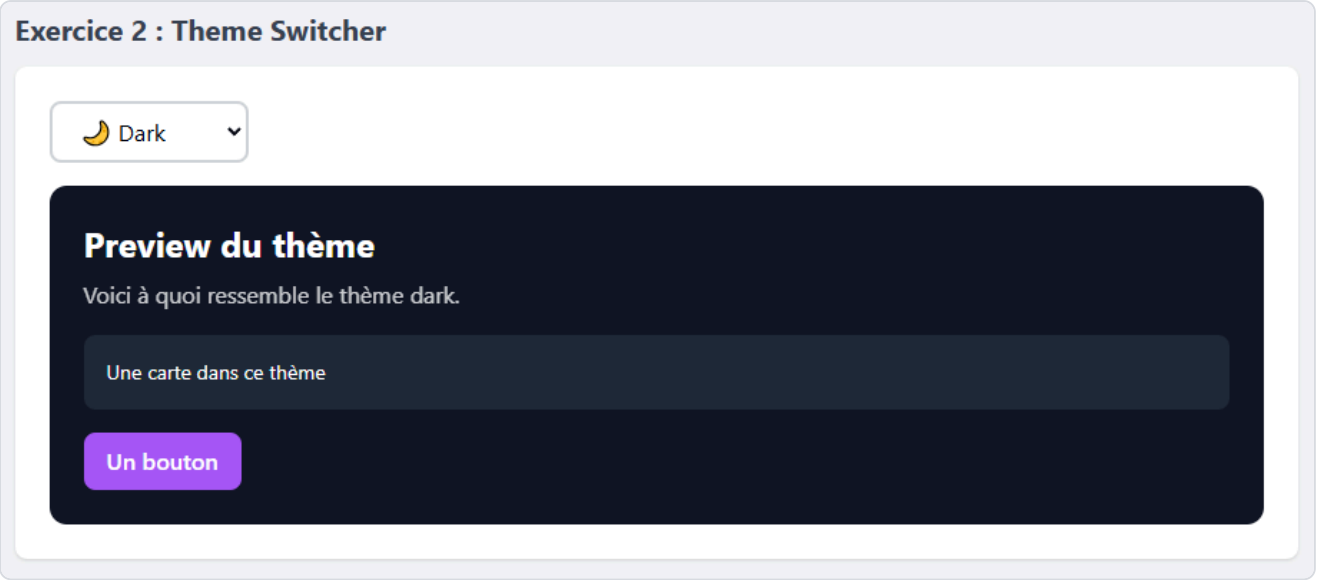
```xml
<script setup>
import { ref } from 'vue'

const selectedTheme = ref('light')

// Définir les classes pour chaque thème
const themes = {
  light: {
    bg: 'bg-white',
    text: 'text-gray-800',
    button: 'bg-blue-500 text-white'
  },
  dark: {
    bg: 'bg-gray-900',
    text: 'text-white',
    button: 'bg-purple-500 text-white'
  },
  colorful: {
    bg: 'bg-gradient-to-br from-pink-500 to-yellow-500',
    text: 'text-white',
    button: 'bg-white text-pink-500'
  }
}
</script>

<template>
  <div>
    <!-- Select pour choisir le thème -->
    <select v-model="selectedTheme">
      <!-- ... options ... -->
    </select>

    <!-- Aperçu avec les classes dynamiques -->
    <div :class="[themes[selectedTheme].bg, themes[selectedTheme].text]">
      <!-- Contenu preview -->
    </div>
  </div>
</template>
```



Exercice 3 : Formulaire Contact ⭐⭐

Objectif : Créer un formulaire complet avec validation visuelle

Ta mission : Créer un composant `ContactForm.vue` avec :

- Input : Nom (requis)
- Input : Email (requis)
- Select : Sujet (3-4 options)
- Textarea : Message (requis)
- Checkbox : "J'accepte d'être recontacté"
- Bouton Envoyer (désactivé si champs vides)
- Preview du message avant envoi

Props à implémenter : Aucune, tout est interne

Comportement :

- Le bouton est grisé tant que les champs requis sont vides
- La preview montre un résumé de ce qui sera envoyé
- Au clic sur "Envoyer", afficher une alerte de confirmation (pas de vrai envoi)

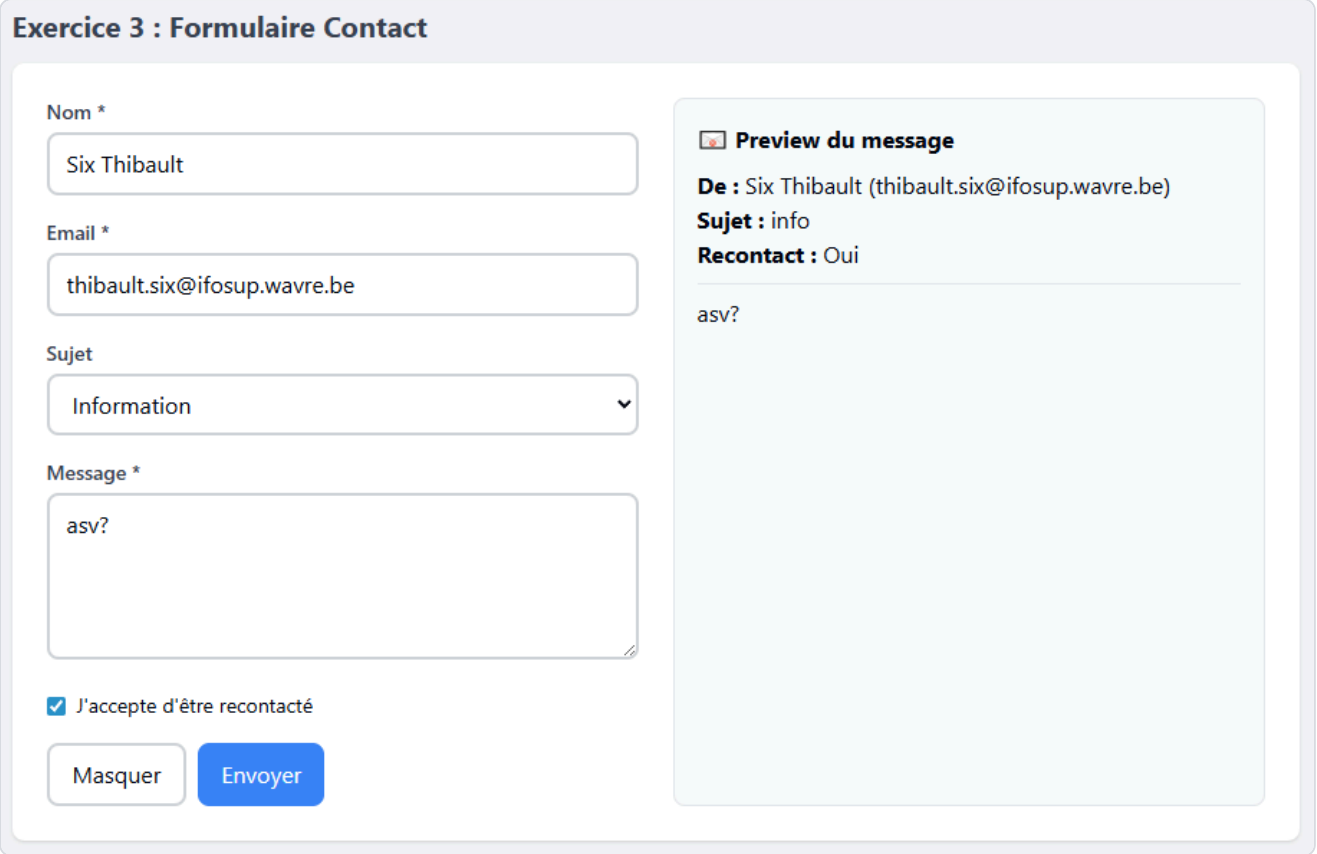
Coup de pouce - Validation simple :

```XML
<script setup>
import { ref, computed } from 'vue'

const name = ref('')
const email = ref('')
const message = ref('')

// computed pour vérifier si le form est valide
const isFormValid = computed(() => {
  return name.value.trim() !== '' &&
         email.value.trim() !== '' &&
         message.value.trim() !== ''
})
</script>

<template>
  <button :disabled="!isFormValid">
    Envoyer
  </button>
</template>
```



Exercice 4 : Filtres Produits ⭐⭐⭐

Objectif : Filtrer une liste de produits avec plusieurs critères

Ta mission : Créer un composant `ProductFilter.vue` avec :

- Une liste de produits (depuis un fichier JSON)
- Un input de recherche (filtre par nom)
- Des checkboxes pour filtrer par catégorie
- Un select pour trier (prix croissant/décroissant, nom A-Z)

Étape 1 : Créer le fichier de données

Crée le fichier `src/data/products.json` :

```json
[
  { "id": 1, "name": "MacBook Pro", "category": "laptop", "price": 2399, "image": "💻"
},
  { "id": 2, "name": "iPhone 15", "category": "phone", "price": 999, "image": "📱" },
  { "id": 3, "name": "iPad Air", "category": "tablet", "price": 699, "image": "📱" },
  { "id": 4, "name": "Dell XPS", "category": "laptop", "price": 1299, "image": "💻" },
  { "id": 5, "name": "Samsung Galaxy", "category": "phone", "price": 899, "image":
"📱" },
  { "id": 6, "name": "Surface Pro", "category": "tablet", "price": 999, "image": "📱"
},
  { "id": 7, "name": "ThinkPad X1", "category": "laptop", "price": 1599, "image": "💻"
},
  { "id": 8, "name": "Pixel 8", "category": "phone", "price": 699, "image": "📱" }
]
```

Étape 2 : Structure du composant

```xml
<script setup>
import { ref, computed } from 'vue'
import products from '@/data/products.json'

// Filtres
const searchQuery = ref('')
const selectedCategories = ref([])
const sortBy = ref('name-asc')

// Liste des catégories uniques (extraites des produits)
const categories = [...new Set(products.map(p => p.category))]

// Produits filtrés et triés (computed)
const filteredProducts = computed(() => {
  let result = products

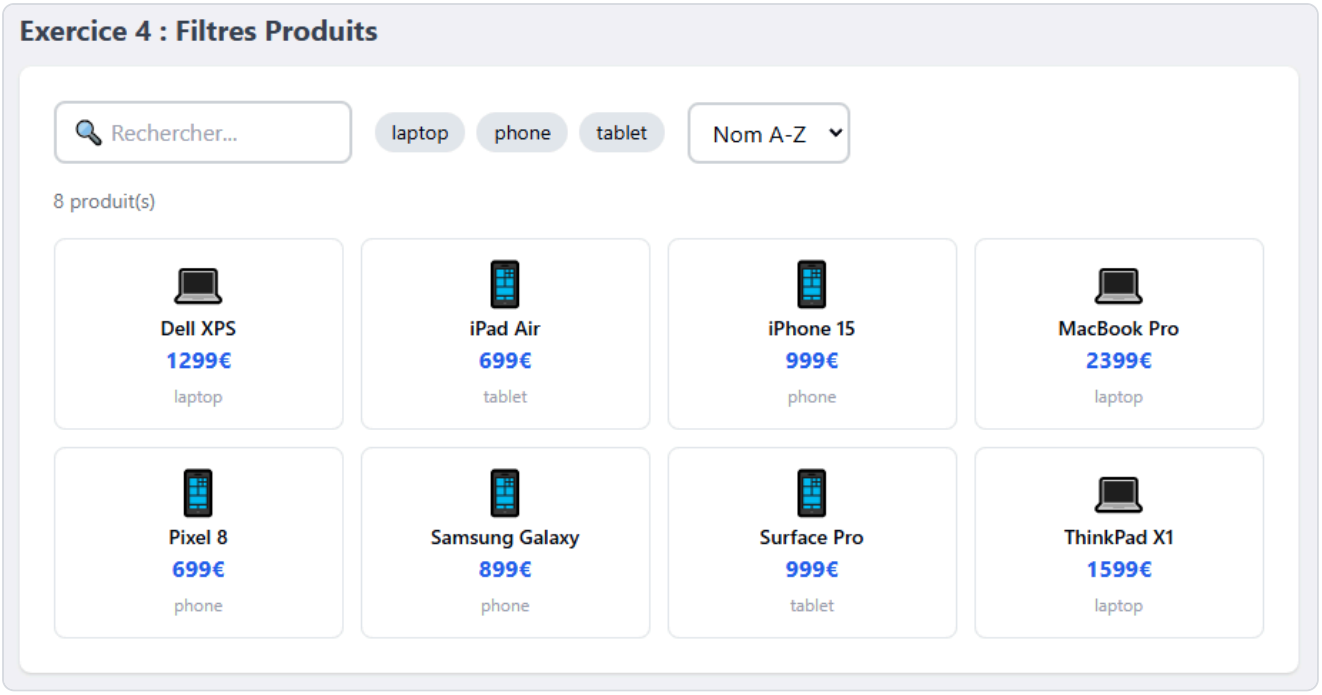
  // 1. Filtre par recherche
  if (searchQuery.value) {
    result = result.filter(p =>
      p.name.toLowerCase().includes(searchQuery.value.toLowerCase())
    )
  }

  // 2. Filtre par catégories
  if (selectedCategories.value.length > 0) {
    result = result.filter(p =>
      selectedCategories.value.includes(p.category)
    )
  }

  // 3. Tri
  // ... à toi de jouer !

  return result
})
</script>
```

Rendu attendu :



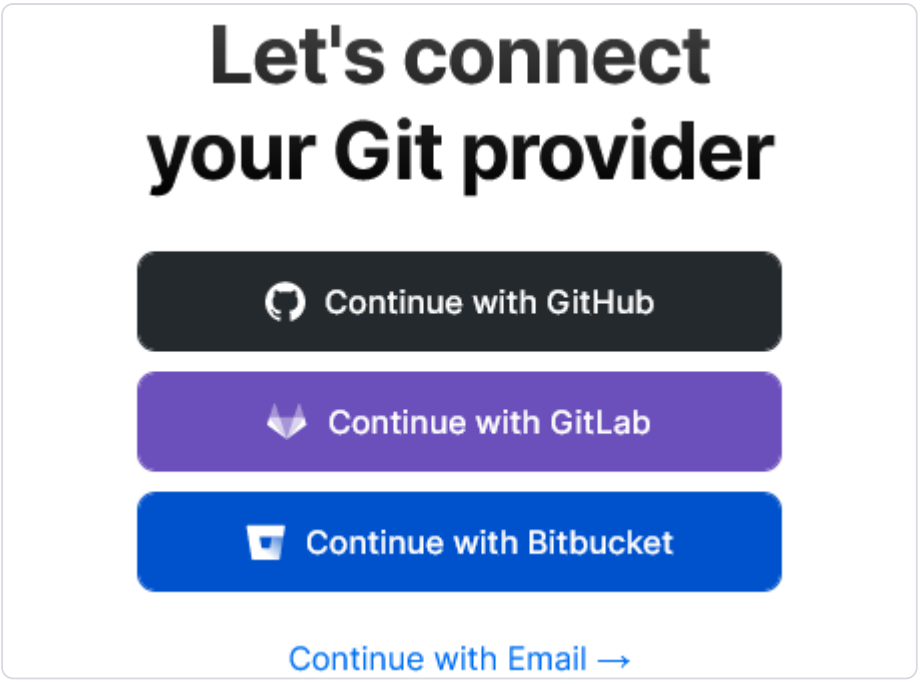
3.12 Hébergement avec Vercel

Dans cette section, nous allons apprendre comment héberger notre projet en utilisant Vercel. Vercel est une plateforme d'hébergement qui permet de déployer facilement des applications web.

Étape 1: Créer un compte Vercel

Tout d'abord, nous devons créer un compte Vercel. Pour cela, suivez les étapes suivantes :

1. Rendez-vous sur le site web de Vercel à l'adresse vercel.com.
2. Cliquez sur le bouton "Sign Up" pour créer un nouveau compte.
3. Choisissez l'option de connexion avec GitHub pour lier votre compte GitHub à Vercel.
4. Suivez les instructions pour autoriser Vercel à accéder à votre compte GitHub.



Étape 2 : Sauvegarder le projet sur GitHub depuis VS Code

Dans cette étape, nous allons voir comment initialiser un dépôt Git local, configurer votre identité Git, effectuer des commits et pousser votre code vers GitHub directement depuis Visual Studio Code (VS Code). Cela vous permettra de conserver l'historique de votre code et de le partager facilement avec d'autres, tout en simplifiant le processus de déploiement sur Vercel.

1. Initialiser un dépôt Git local

Tout d'abord, nous devons initialiser un dépôt Git dans votre projet pour commencer à suivre les modifications.

- **Ouvrez votre projet** dans VS Code.
- **Initialisez Git** :
  - Cliquez sur l'icône **Contrôle de code source** dans la barre latérale gauche.
  - Cliquez sur le bouton **"Initialiser le dépôt"**.

> ***Pourquoi ?*** *Initialiser un dépôt Git permet de commencer à suivre les modifications de votre projet. Git enregistrera les changements au fil du temps, facilitant ainsi la gestion des versions.*

2. Configurer votre identité Git

Avant de faire votre premier commit, il est important de configurer votre nom et votre adresse e-mail pour que Git puisse les associer aux commits que vous créez.

- **Ouvrez le terminal intégré** dans VS Code : **Affichage** > **Terminal**.
- **Configurez votre nom d'utilisateur** (utilisez votre pseudonyme GitHub) :

```
git config --global user.name "votre-pseudo-github"
```

- **Configurez votre adresse e-mail** (utilisez l'adresse e-mail associée à votre compte GitHub) :

```
git config --global user.email "votre-adresse-email@example.com"
```

> ***Pourquoi ?*** *Configurer votre identité Git permet d'associer vos commits à votre compte GitHub, ce qui facilite le suivi de vos contributions et la collaboration avec d'autres développeurs.*

3. Créer un premier commit

Un commit est un enregistrement des modifications apportées à votre code.

- **Sélectionnez les fichiers à committer** :
  - Dans l'onglet **Contrôle de code source**, vous verrez la liste des fichiers modifiés sous **"Changes"**.
  - Cliquez sur **"Tout ajouter"** (icône **"+"** en haut de la section) pour ajouter tous les fichiers au **staging**.
- **Entrez un message de commit** :
  - Dans le champ en haut de l'onglet, saisissez un message descriptif, par exemple, `"Initial commit"`.
- **Validez le commit** :
  - Appuyez sur **Entrée** ou cliquez sur l'icône **"✔️"**.

> ***Pourquoi ?*** *Créer des commits avec des messages clairs vous aide à comprendre quelles modifications ont été apportées et pourquoi, ce qui est essentiel pour le suivi et la maintenance du projet.*

4. Pousser le code vers GitHub

Nous allons maintenant pousser votre code vers GitHub. Si vous n'avez pas encore lié VS Code à votre compte GitHub, vous serez invité à le faire.

- **Pousser les modifications** :
  - Une fois le commit effectué, le bouton **"Synchroniser les modifications"** (icône de synchronisation 🔄) apparaît dans l'onglet **Contrôle de code source**.
  - Cliquez sur ce bouton pour pousser vos commits vers GitHub.
- **Se connecter à GitHub** :
  - Si c'est la première fois que vous poussez vers GitHub depuis VS Code, une notification apparaîtra pour vous demander de vous authentifier.
  - Cliquez sur **"Se connecter à GitHub..."**.

- Une fenêtre de navigateur s'ouvrira pour vous permettre d'autoriser VS Code à accéder à votre compte GitHub.
- **Connectez-vous à GitHub** si nécessaire et autorisez VS Code en cliquant sur **"Authorize GitHub"**.
- Revenez à VS Code après la confirmation.

> ***Pourquoi ?*** *Se connecter à GitHub depuis VS Code vous permet de gérer vos dépôts GitHub directement depuis l'éditeur, simplifiant ainsi votre flux de travail.*

5. Créer le dépôt GitHub depuis VS Code

Si vous n'avez pas encore créé de dépôt pour ce projet sur GitHub, VS Code vous proposera de le faire automatiquement.

- **Créer le dépôt distant** :
  - Après avoir cliqué sur **"Synchroniser les modifications"**, une notification apparaîtra indiquant qu'aucun dépôt distant n'est configuré.
  - Cliquez sur **"Publier sur GitHub"**.
- **Configurer le dépôt** :
  - **Choisissez la visibilité** du dépôt : sélectionnez **Public** ou **Privé** selon vos besoins.
  - **Nom du dépôt** : par défaut, le nom du dépôt sera le nom du dossier de votre projet. Vous pouvez le modifier si vous le souhaitez.
- **Confirmer la création** :
  - VS Code créera le dépôt sur GitHub et configurera le lien entre votre dépôt local et le dépôt distant automatiquement.
- **Terminer le push** :
  - VS Code poussera vos commits vers le nouveau dépôt GitHub.

> ***Pourquoi ?*** *Cette fonctionnalité vous évite d'avoir à créer manuellement le dépôt sur GitHub, rendant le processus plus fluide et rapide.*

6. Vérifier sur GitHub

- **Accédez à votre dépôt en ligne** :
  - Une notification apparaîtra avec un lien vers le dépôt GitHub nouvellement créé.
  - Cliquez sur ce lien pour ouvrir le dépôt dans votre navigateur.
  - Vous devriez voir tous les fichiers de votre projet ainsi que votre commit initial.

> ***Pourquoi ?*** *Vérifier que le code est bien en ligne confirme que toutes les étapes ont été réalisées correctement et que votre projet est prêt pour le déploiement ou la collaboration.*

7. Poursuivre le travail avec Git et VS Code

Désormais, chaque fois que vous apporterez des modifications à votre code :

- **Étape 1** : Enregistrez vos modifications.
- **Étape 2** : Allez dans l'onglet **Contrôle de code source** pour voir les fichiers modifiés.
- **Étape 3** : Ajoutez les fichiers au staging si nécessaire.
- **Étape 4** : Entrez un message de commit descriptif.
- **Étape 5** : Cliquez sur **"Valider"** pour créer un nouveau commit.
- **Étape 6** : Une fois le commit réalisé, le bouton **"Synchroniser les modifications"** apparaîtra dans l'onglet **Contrôle de code source**.
- **Étape 7** : Cliquez sur **"Synchroniser les modifications"** pour pousser vos commits vers GitHub.

> ***Pourquoi ?*** *Ce flux de travail vous permet de garder votre dépôt GitHub à jour avec vos dernières modifications, facilitant le suivi, la collaboration et le déploiement continu.*

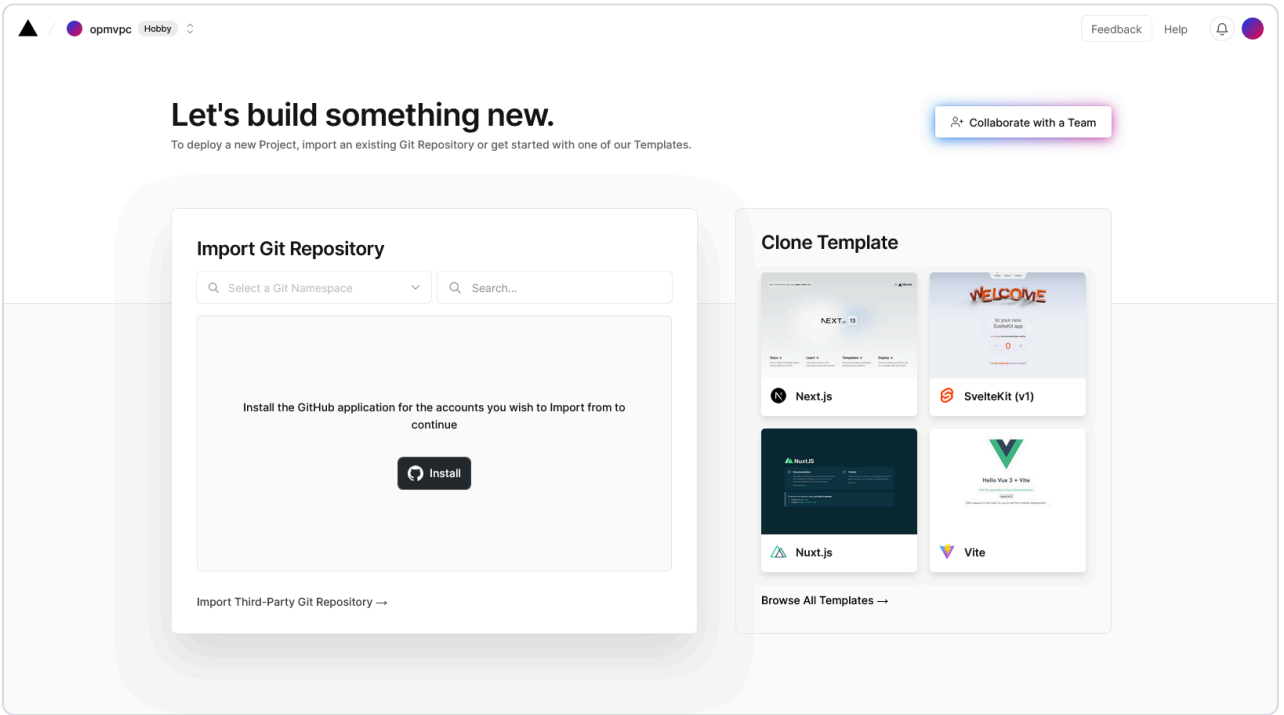
Conclusion

Vous avez maintenant sauvegardé votre projet sur GitHub en utilisant VS Code, sans avoir à créer manuellement le dépôt sur GitHub. Cette méthode simplifie le processus de mise en place du contrôle de version et facilite le déploiement sur des plateformes comme Vercel.

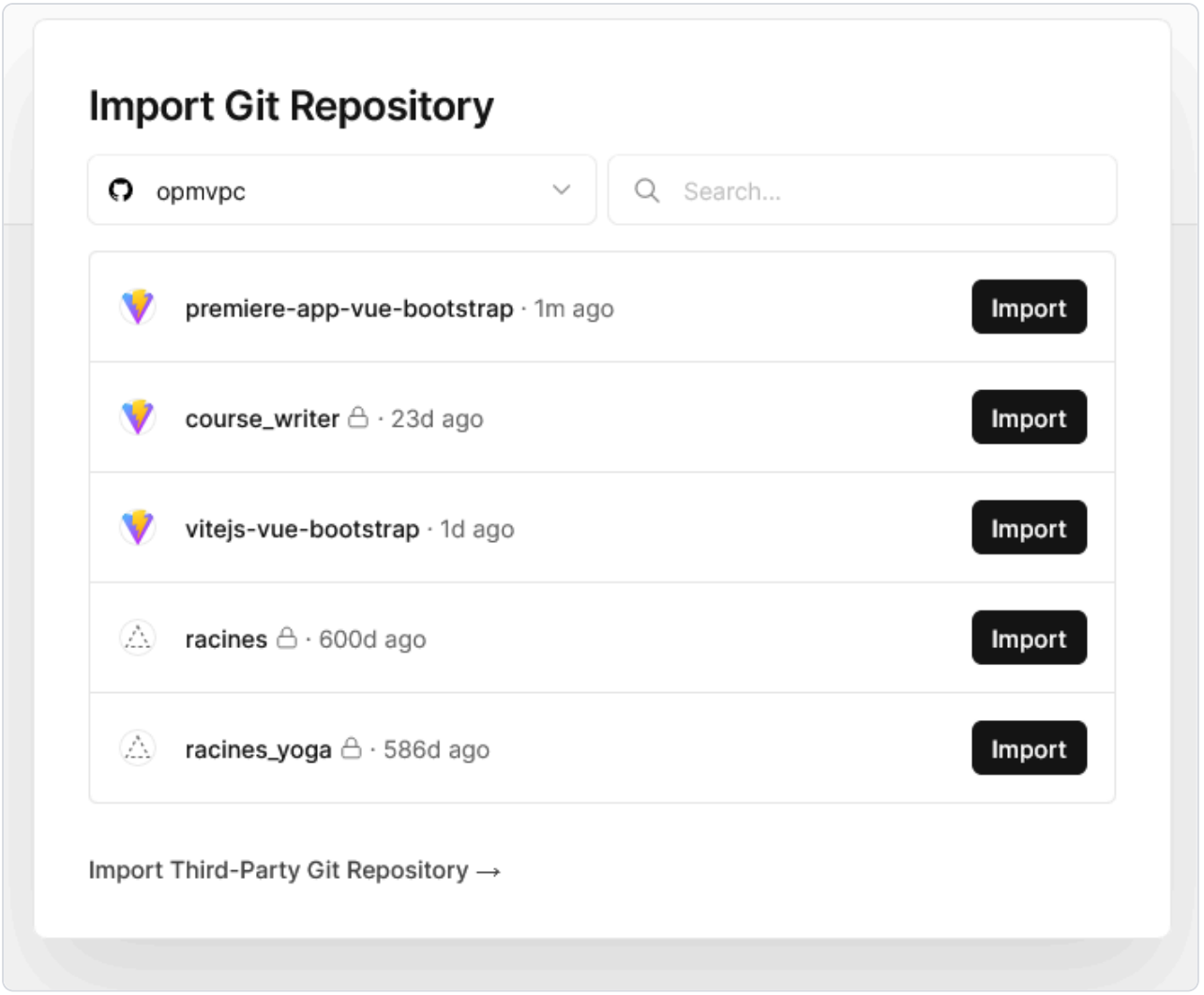
Étape 3: Lier votre compte GitHub à Vercel

Maintenant, nous allons lier notre compte GitHub à Vercel. Voici les étapes à suivre :

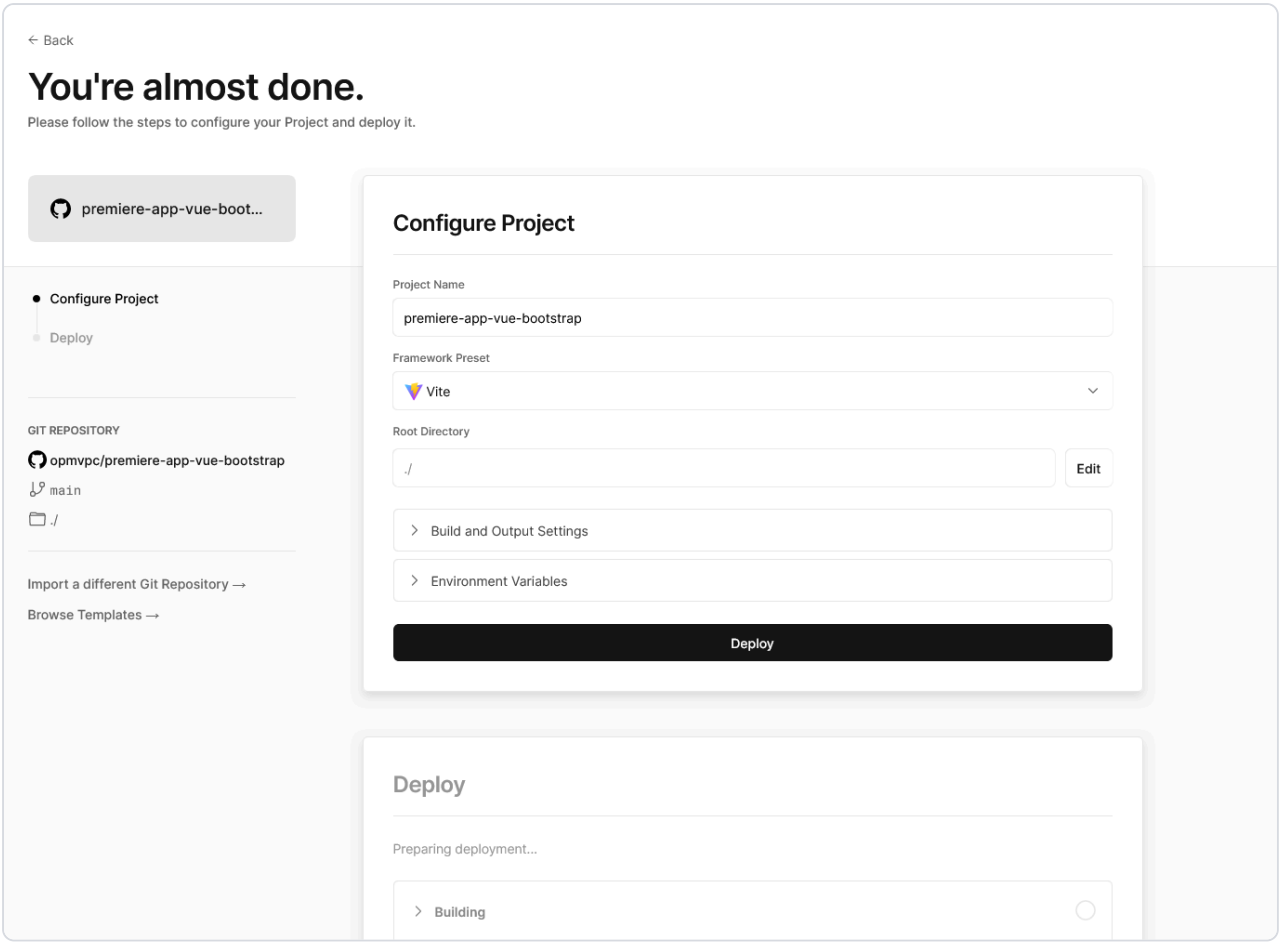
1. Retournez sur le site web de Vercel.
2. Cliquez sur le bouton "Install" pour autoriser Vercel à accéder à votre compte GitHub.



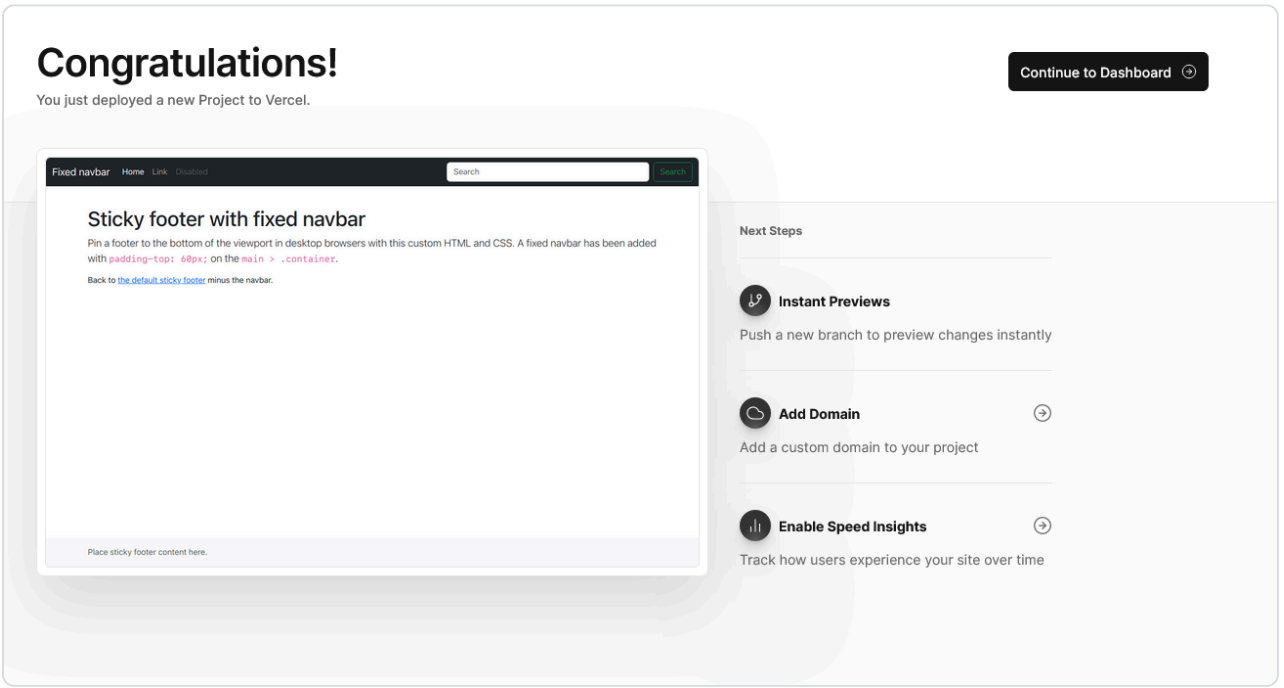
3. Une fois installé, vous verrez une liste de vos projets GitHub.



4. Cliquez sur le bouton "Import" du projet que vous venez de créer dans StackBlitz.
5. Vous serez redirigé vers la page de configuration de votre projet.



6. Vercel détectera automatiquement que vous utilisez Vue.js.
7. Cliquez sur le bouton "Deploy" pour déployer votre projet.
8. Attendez quelques secondes et normalement, si tout s'est bien passé, vous pourrez accéder à votre site en cliquant sur le lien de prévisualisation fourni par Vercel.



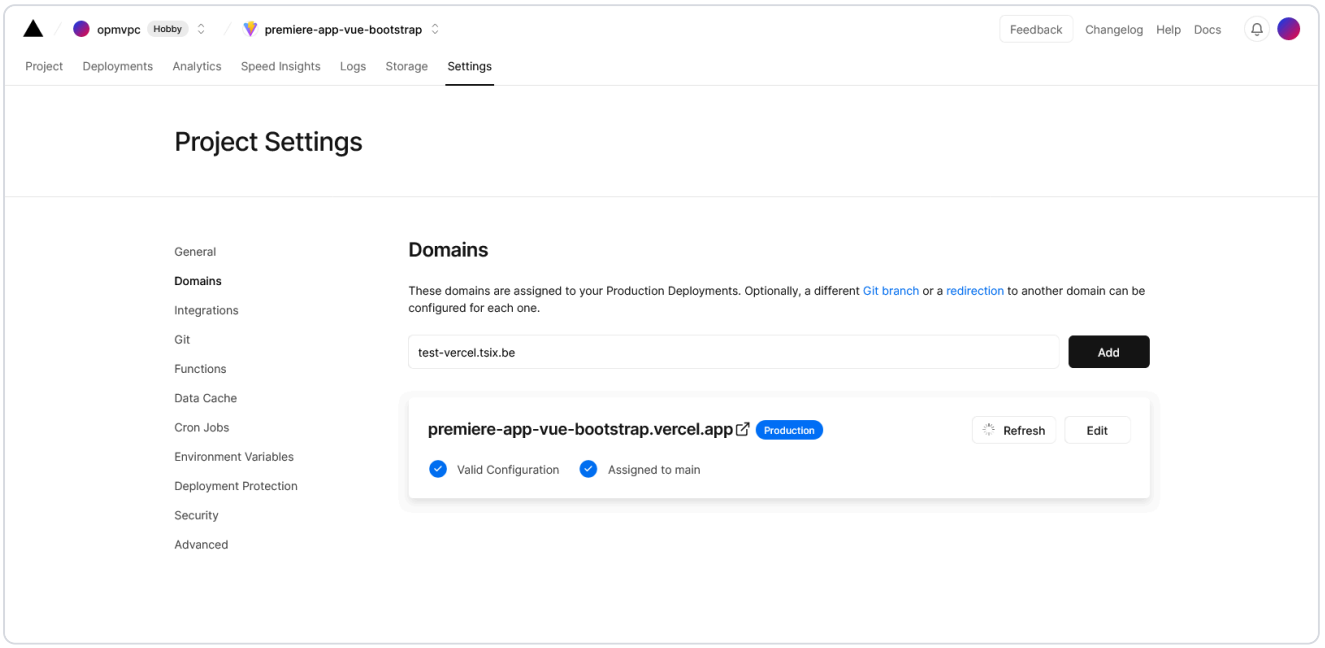
Et voilà ! Vous avez maintenant hébergé votre projet sur Vercel. Vous pouvez partager le lien de prévisualisation avec d'autres personnes pour qu'elles puissent voir votre site en ligne.

Chaque fois que vous ferez des modifications sur cette application, la nouvelle version sera automatiquement mise en ligne.

Nom de domaine

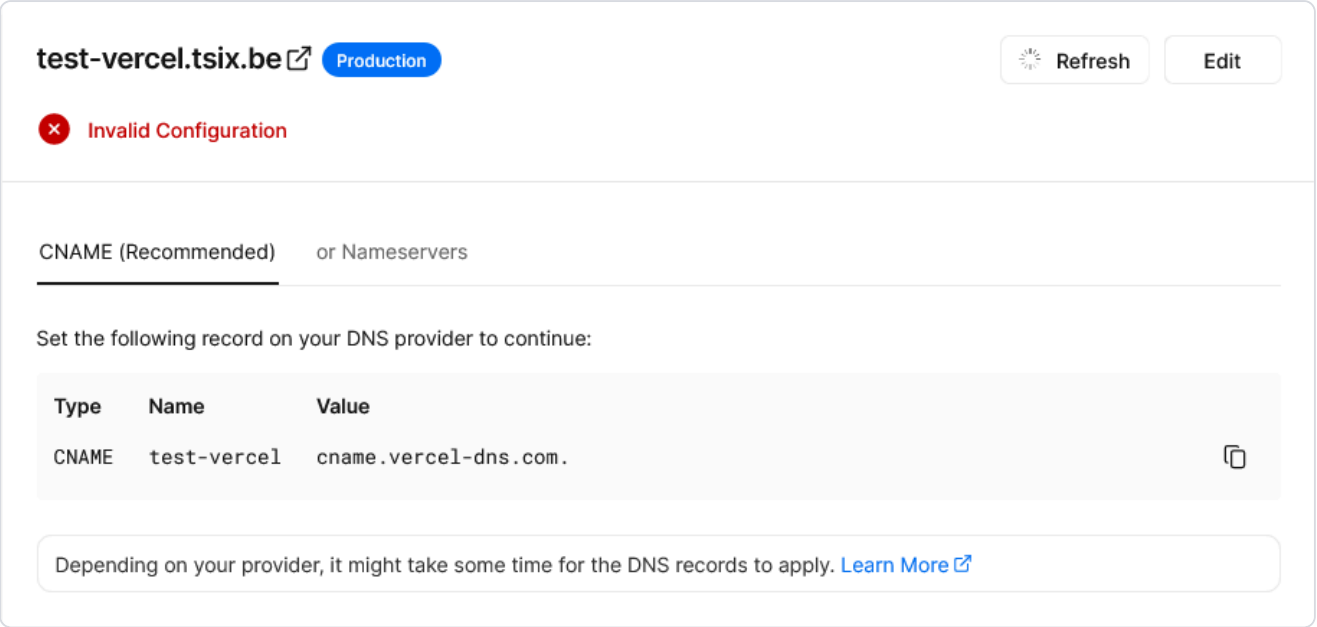
Vous pouvez lier votre nom de domaine à un site hébergé sur Vercel.

Pour cela, allez dans les paramètres de votre projet, puis dans l'onglet "Domains".

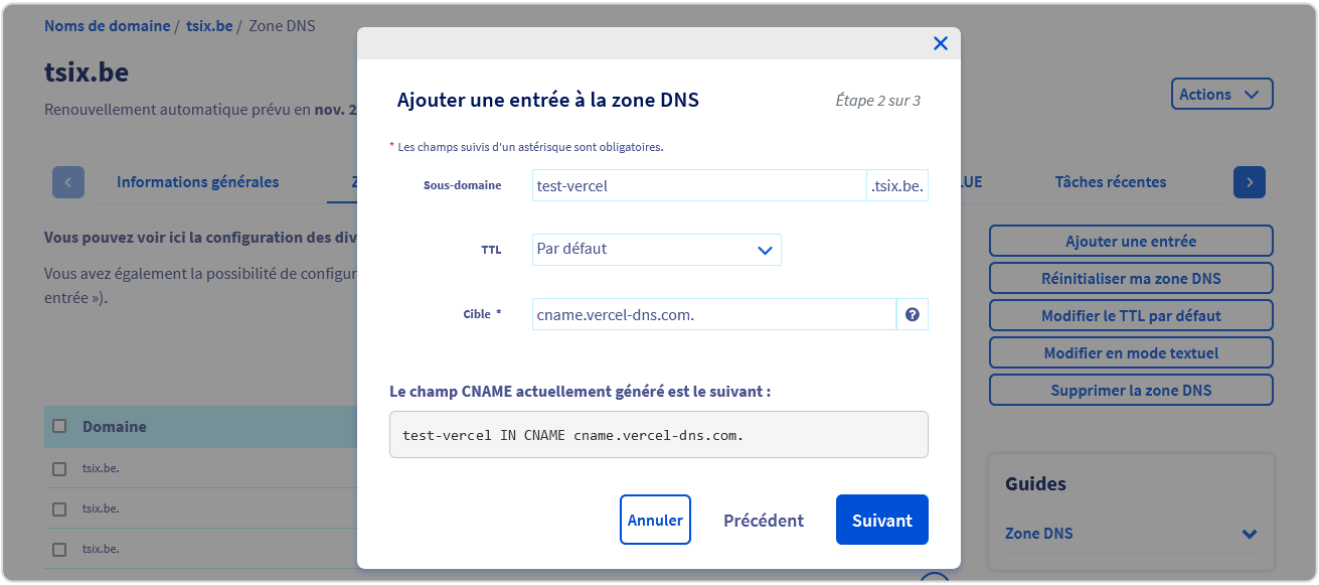


Ajoutez votre nom de domaine avec un sous domaine, puis cliquez sur “Add”.

Nous devons ensuite suivre les instructions.



Allez dans l’interface de gestion de votre nom de domaine et ajoutez un champs CNAME avec les paramètres fournis.



Nous pouvons désormais accéder à notre application avec notre nom de domaine. Vercel émet automatiquement un certificat SSL pour notre site.

3.13 Héberger un site Vue.js sur OVH

Ce guide explique comment héberger un site Vue.js sur OVH, étape par étape. Nous allons construire le projet, le tester, uploader les fichiers via FTP, puis configurer le domaine sur OVH.

Étape 1 : Construction du projet Vue.js

1.1. Ouvrir le terminal

Ouvrez le terminal dans le dossier racine de votre projet Vue.js.

1.2. Lancer la commande `npm run build`

Pour construire le projet, exécutez :

```bash
npm run build
```

Cette commande génère les fichiers statiques optimisés nécessaires au site, dans un dossier `dist`.



1.3. Prévisualiser le projet

Vérifiez le fonctionnement localement avec :

```bash
npm run preview
```

Cela lance un serveur local pour tester le site avant la mise en ligne.

```
C:\laragon\www\exo-vuejs-intro(main -> origin)
λ npm run preview

> exo-vuejs-intro@0.0.0 preview
> vite preview

  ➜  Local:   http://localhost:4173/
  ➜  Network: use --host to expose
  ➜  press h + enter to show help
```

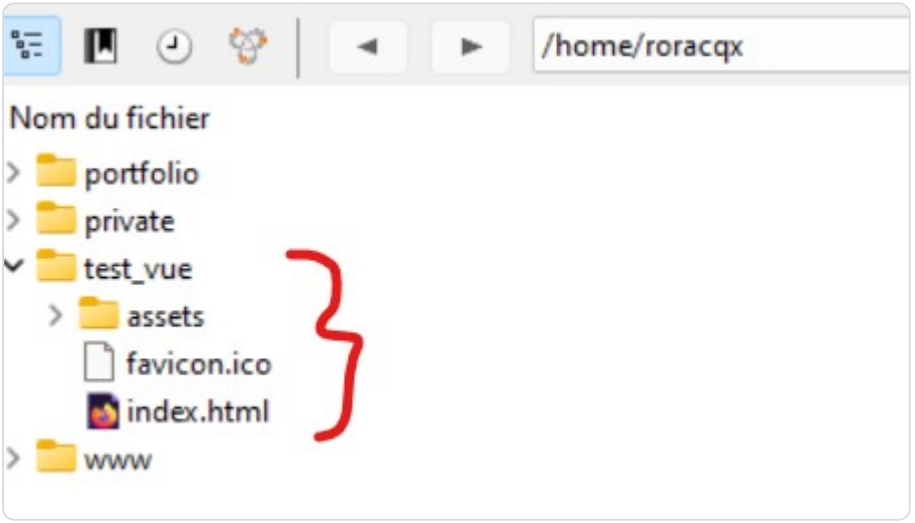
Étape 2 : Uploader les fichiers sur OVH

2.1. Utilisation du client FTP

Transférez les fichiers du dossier `dist` vers le serveur OVH en utilisant un client FTP comme Cyberduck.

2.2. Arborescence des fichiers sur le FTP

Assurez-vous que les fichiers (notamment `index.html` et `assets`) sont placés correctement dans le répertoire du serveur.

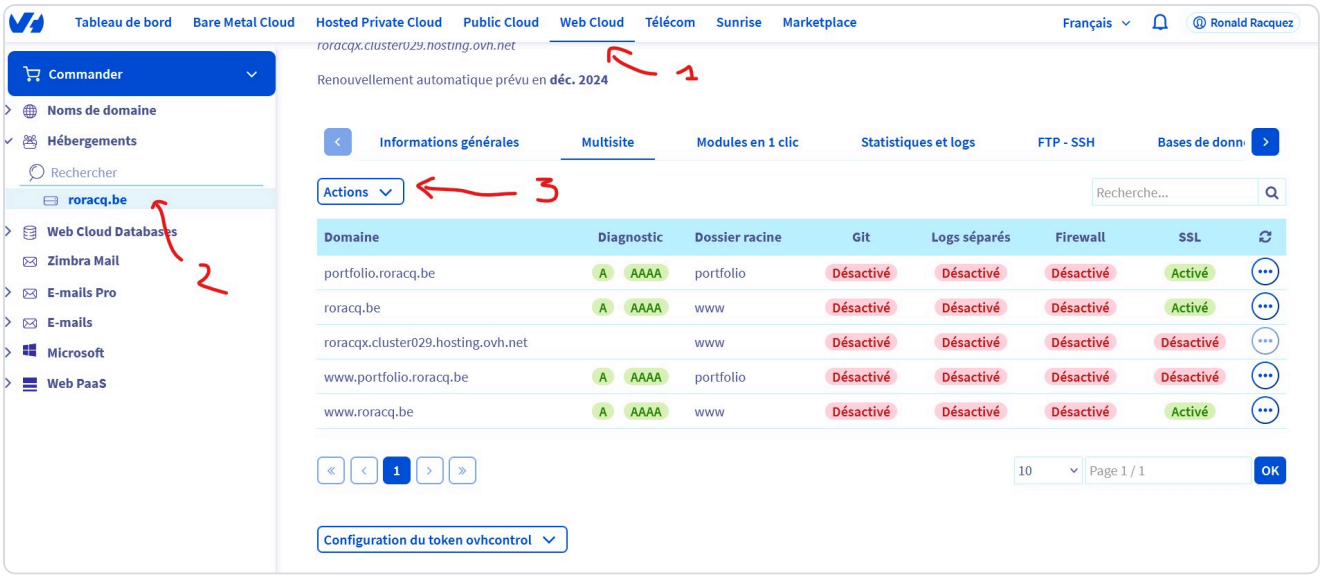


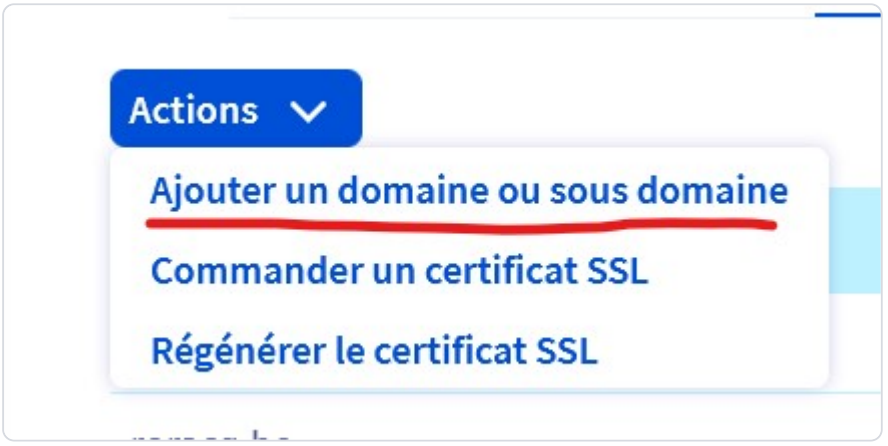
Étape 3 : Configurer le domaine sur OVH

3.1. Accéder à la configuration

1. Connectez-vous à l'administration d'OVH.

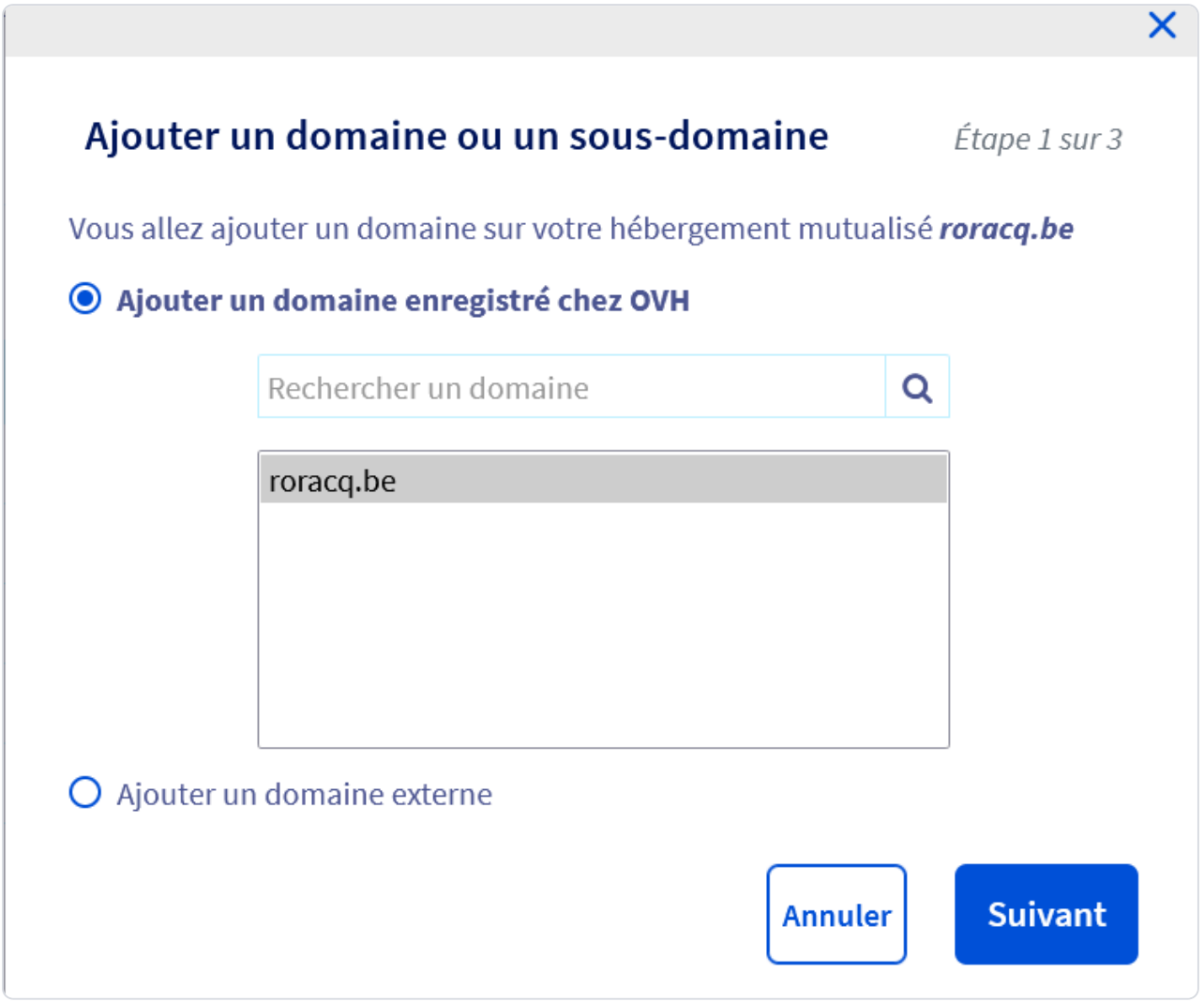
2. Allez dans "Web Cloud" (1), sélectionnez votre hébergement (2), puis cliquez sur "Actions" et choisissez "Ajouter un domaine ou sous-domaine" (3).



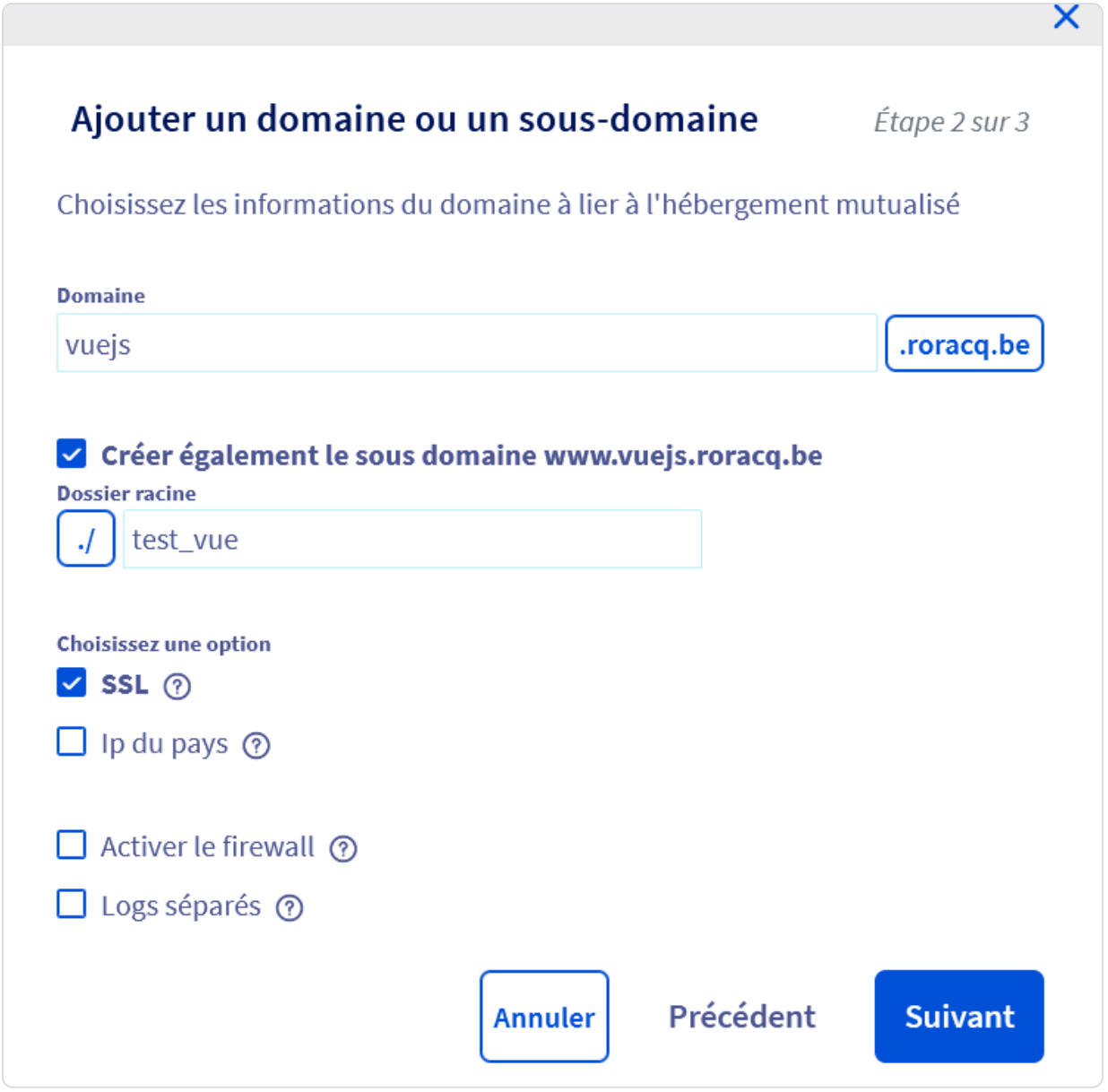


3.2. Ajouter un sous-domaine

1. Choisissez "Ajouter un domaine enregistré chez OVH" et sélectionnez votre domaine principal.



2. Entrez le nom du sous-domaine (ex. "vuejs"), ce qui donnera `vuejs.roracq.be`.

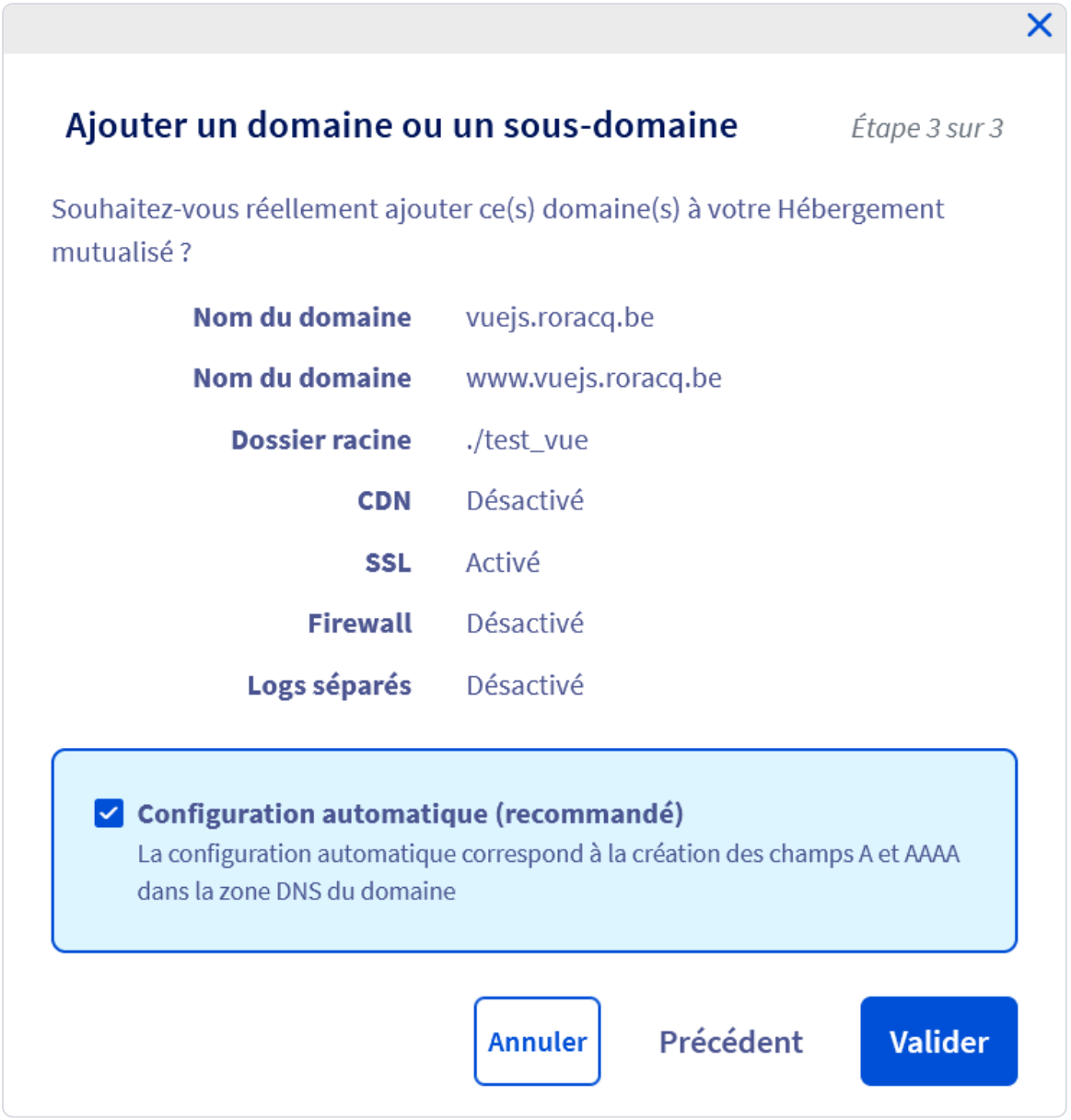


3. Indiquez le dossier racine (à savoir `test_vue`).

4. Activez l'option SSL, puis cliquez sur "Suivant".

3.3. Valider la configuration

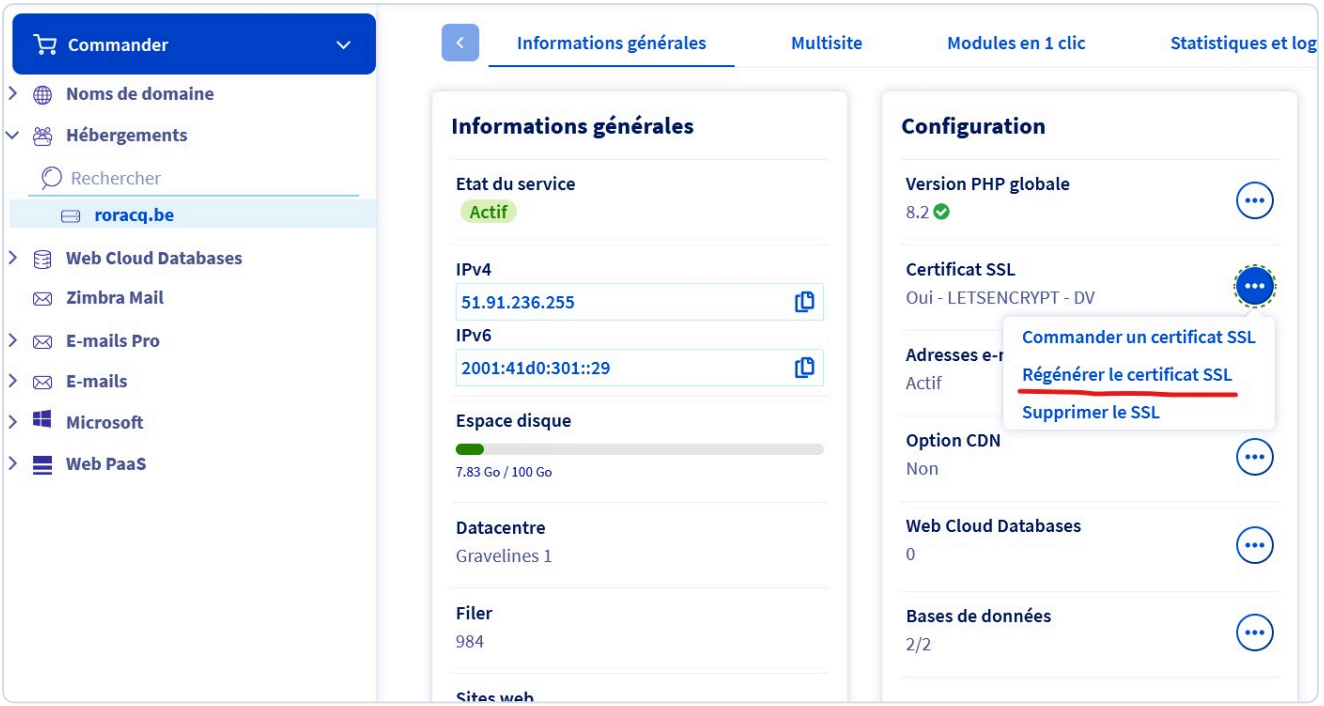
Vérifiez et cliquez sur "Valider" pour ajouter le sous-domaine.



Étape 4 : Régénération du certificat SSL

Après avoir ajouté le sous-domaine, le certificat SSL sera généré automatiquement. Cela peut prendre plusieurs heures. Si le SSL n'est pas actif après ce délai, vous pouvez manuellement le régénérer dans l'interface OVH :

- Sous "Informations générales", cliquez sur les trois points à côté de "Certificat SSL" et sélectionnez "Régénérer le certificat SSL".



Étape 5 : Configurer le fichier `.htaccess`

Pour éviter une erreur 404 lors de l'accès direct à une page autre que la page d'accueil, configurez Apache via un fichier `.htaccess`. Sur les hébergements OVH, Apache est utilisé, d'où l'importance de ce fichier.

Créez un fichier `.htaccess` à la racine du projet et ajoutez :

```
<IfModule mod_negotiation.c>
    Options -MultiViews
</IfModule>

<IfModule mod_rewrite.c>
    RewriteEngine On
    RewriteBase /
    RewriteRule ^index\.html$ - [L]
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteCond %{REQUEST_FILENAME} !-d
    RewriteRule . /index.html [L]
</IfModule>
```

Ce fichier redirige toutes les requêtes vers `index.html` si le fichier ou dossier demandé n'existe pas, permettant ainsi à Vue Router de gérer les routes côté client.

Étape 6 : Tests de vérification

- **Charger une URL spécifique** : Essayez de charger directement une URL comme http://vuejs.roracq.be/flags pour vérifier la configuration `.htaccess`. Si tout est correct, cette page se chargera sans erreur 404.
- **Tester la redirection HTTPS** : Assurez-vous que votre site utilise HTTPS. Visitez votre site avec `https://` pour vérifier que le SSL est actif. Si ce n'est pas le cas, régénérez le certificat.
- **Naviguer sur plusieurs pages** : Parcourez le site pour vérifier que toutes les pages fonctionnent et que les assets se chargent correctement.

Conclusion

Vous avez maintenant mis en ligne votre site Vue.js sur OVH, ajouté un sous-domaine, configuré `.htaccess` pour gérer les routes, et sécurisé le site avec SSL. Testez bien toutes les pages pour garantir que tout fonctionne.

Si vous rencontrez des problèmes, revenez à ce guide ou consultez la documentation d'OVH.