Cheatsheet IntelliJ → Neovim

Contents

Neovim comme IDE **5**
Guide complet de migration depuis IntelliJ 5

Chapitre 1 — Pourquoi Neovim peut remplacer IntelliJ **5**
Objectif du chapitre 5
1.1 — IntelliJ : ce que tu utilises vraiment (sans t'en rendre compte) 5
Ce qu'IntelliJ fait très bien 6
Le coût caché 6
1.2 — Mythe à déconstruire : "Neovim est juste un éditeur" 6
1.3 — IDE monolithique vs IDE composable 6
IntelliJ 6
Neovim 7
1.4 — Pourquoi des développeurs expérimentés migrent 7
1.5 — Ce que Neovim fait aussi bien qu'IntelliJ 7
1.6 — Ce que Neovim fait mieux 8
1.7 — Limites honnêtes de Neovim 8
1.8 — Le piège principal : vouloir utiliser Neovim comme IntelliJ 8
1.9 — Le nouveau mental model 8
1.10 — Exercice de réflexion 9
1.11 — Checklist de fin de chapitre 9

Chapitre 2 — Le mental model Vim (sans douleur) **9**
Objectif du chapitre 9
2.1 — Le problème des éditeurs classiques 9
2.2 — Le postulat fondamental de Vim 10
2.3 — Les modes essentiels 10
Mode Normal 10
Mode Insertion 10
Mode Visuel 10
Mode Commande 10
2.4 — Le réflexe le plus important 10
2.5 — Vim n'est pas une liste de raccourcis 10
2.6 — Pourquoi ça devient naturel 11
2.7 — Erreurs fréquentes 11
2.8 — Which-key comme filet de sécurité 11
2.9 — Exercice guidé 11

2.10 — Checklist de fin de chapitre . . . 12

Chapitre 3 — Navigation sans souris (devenir efficace rapidement) 12
Objectif du chapitre . . . 12
3.1 — Le vrai problème de la navigation dans les IDE classiques . . . 12
3.2 — Le modèle de navigation de Neovim . . . 12
Fichier . . . 13
Buffer . . . 13
Fenêtre (window) . . . 13
3.3 — Pourquoi ce modèle est supérieur . . . 13
3.4 — Ouvrir un fichier (remplacer Ctrl+P) . . . 13
3.5 — Rechercher dans tout le projet (Ctrl+Shift+F) . . . 14
3.6 — Naviguer entre fichiers ouverts (buffers) . . . 14
3.7 — Explorateur de fichiers (usage ponctuel) . . . 14
3.8 — Navigation par symboles . . . 15
3.9 — Aller / retour dans l'historique . . . 15
3.10 — Exercice pratique . . . 15
3.11 — Checklist de fin de chapitre . . . 15

Chapitre 4 — Édition efficace (écrire moins, faire plus) 16
Objectif du chapitre . . . 16
4.1 — Le problème de la sélection visuelle . . . 16
4.2 — Commandes d'édition essentielles . . . 16
4.3 — Insertion intelligente . . . 17
4.4 — Le réflexe fondamental . . . 17
4.5 — Commenter du code . . . 17
4.6 — Annuler et refaire . . . 17
4.7 — Exercice pratique . . . 17
4.8 — Checklist de fin de chapitre . . . 18

Chapitre 5 — LSP : le cerveau de ton IDE 18
Objectif du chapitre . . . 18
5.1 — Pourquoi l'intelligence du code a été externalisée . . . 18
5.2 — Qu'est-ce qu'un LSP . . . 18
5.3 — Fonctionnalités fournies par le LSP . . . 19
5.4 — Commandes essentielles . . . 19
5.5 — Pourquoi les refactorings sont sûrs . . . 19
5.6 — Diagnostics et navigation . . . 19
5.7 — Mason et la gestion des serveurs . . . 20
5.8 — Exercice pratique . . . 20
5.9 — Checklist de fin de chapitre . . . 20

Chapitre 6 — Qualité de code & refactor (sans douleur) 20
Objectif du chapitre . . . 20
6.1 — Le faux problème de la qualité de code . . . 21
6.2 — Séparation claire des responsabilités . . . 21
LSP . . . 21
Formatter . . . 21

Linter 21
6.3 — Le formatage automatique (format on save) 21
6.4 — ESLint et diagnostics intelligents 22
6.5 — Refactor propre : quand utiliser quoi 22
6.6 — Exercice pratique 22
6.7 — Checklist de fin de chapitre 22

Chapitre 7 — Diagnostics & debugging mental **23**
Objectif du chapitre 23
7.1 — Le debugging est un problème cognitif 23
7.2 — Ce qu'est un diagnostic 23
7.3 — Navigation entre diagnostics 23
7.4 — Processus de correction recommandé 24
7.5 — Erreurs en cascade 24
7.6 — Exercice pratique 24
7.7 — Checklist de fin de chapitre 24

Chapitre 8 — Git sans quitter Neovim **24**
Objectif du chapitre 24
8.1 — Le problème des workflows Git trop visuels 25
8.2 — Gitsigns : voir les changements dans le code 25
8.3 — Pourquoi le staging par hunk est essentiel 25
8.4 — Neogit : interface Git complète 25
8.5 — Exercice pratique 26
8.6 — Checklist de fin de chapitre 26

Chapitre 9 — Terminal, tests et debug **26**
Objectif du chapitre 26
9.1 — Pourquoi le contexte est crucial 26
9.2 — Terminal intégré 27
9.3 — Tests avec Neovim 27
9.4 — Debug avec nvim-dap 27
9.5 — Exercice pratique 27
9.6 — Checklist de fin de chapitre 28

Chapitre 10 — Sessions, projets et continuité **28**
Objectif du chapitre 28
10.1 — Le coût caché de la reprise de contexte 28
10.2 — Projet vs session 28
10.3 — Workflow de session recommandé 29
10.4 — Travailler sur plusieurs projets 29
10.5 — Exercice pratique 29
10.6 — Checklist de fin de chapitre 29

Chapitre 11 — React & TypeScript en profondeur (IDE-ready) **30**
Objectif du chapitre 30
11.1 — Pourquoi React est un bon révélateur d'IDE 30
11.2 — Ce que voit réellement le LSP dans un fichier TSX 30
11.3 — Navigation entre composants 30

11.4 — Comprendre rapidement un composant inconnu . . . 31
11.5 — Props et types : refactor sans danger . . . 31
11.6 — Imports automatiques . . . 31
11.7 — Hooks et erreurs courantes . . . 31
11.8 — Exercice pratique . . . 32
11.9 — Checklist de fin de chapitre . . . 32

Chapitre 12 — Workflow React réel (de la feature au commit) 32
Objectif du chapitre . . . 32
12.1 — Pourquoi un workflow réel est indispensable . . . 32
12.2 — Scénario de départ . . . 33
12.3 — Étape 1 : comprendre avant d'agir . . . 33
12.4 — Étape 2 : modifier la source de vérité . . . 33
12.5 — Étape 3 : corriger les usages . . . 33
12.6 — Étape 4 : ajuster le rendu . . . 33
12.7 — Étape 5 : refactor léger . . . 34
12.8 — Étape 6 : tests . . . 34
12.9 — Étape 7 : Git . . . 34
12.10 — Checklist de fin de chapitre . . . 34

Chapitre 13 — Comprendre et maîtriser sa configuration Neovim 34
Objectif du chapitre . . . 34
13.1 — Pourquoi comprendre sa config est essentiel . . . 35
13.2 — Démarrage de Neovim . . . 35
13.3 — Structure typique d'une configuration moderne . . . 35
13.4 — Le rôle du gestionnaire de plugins . . . 35
13.5 — Mason et la gestion des outils externes . . . 35
13.6 — Diagnostiquer un problème . . . 36
13.7 — Checklist de fin de chapitre . . . 36

Chapitre 14 — Personnaliser sans casser 36
Objectif du chapitre . . . 36
14.1 — Pourquoi les configurations cassent . . . 36
14.2 — Règle numéro un : une intention par modification . . . 37
14.3 — Chargement différé obligatoire . . . 37
14.4 — Supprimer est une compétence . . . 37
14.5 — Checklist de fin de chapitre . . . 37

Chapitre 15 — Autonomie et évolution long terme 37
Objectif du chapitre . . . 37
15.1 — L'objectif n'était pas de quitter IntelliJ . . . 38
15.2 — Ce que tu sais faire maintenant . . . 38
15.3 — La règle des 80 / 20 . . . 38
15.4 — Entretenir sa compétence . . . 38
15.5 — Conclusion . . . 38

Annexes — Références, muscle memory et checklists 39

Annexe A — Cheatsheet IntelliJ → Neovim 39

Navigation et recherche 39
Édition 39
Refactor et LSP 39
Diagnostics 40
Git 40
Terminal, tests et debug 40

Annexe B — Muscle memory quotidienne **40**

Annexe C — Checklist de workflow sain **41**

Annexe D — Rappels mentaux essentiels **41**

Fin des annexes **41**

Neovim comme IDE

Guide complet de migration depuis IntelliJ

Temps recommandé : 1 heure par jour
Public : développeur utilisant déjà Vim, débutant Neovim IDE

Chapitre 1 — Pourquoi Neovim peut remplacer IntelliJ

(et quand il ne faut pas)

Objectif du chapitre

À la fin de ce chapitre, tu dois :

- comprendre ce qu'est réellement un IDE
- comprendre ce que Neovim fait différemment d'IntelliJ
- savoir pourquoi Neovim peut le remplacer
- connaître les limites réelles
- adopter le bon mental model avant d'aller plus loin

1.1 — IntelliJ : ce que tu utilises vraiment (sans t'en rendre compte)

Quand tu utilises IntelliJ, tu n'utilises pas simplement un éditeur de texte.

Tu utilises en réalité :

- un moteur d'analyse de code
- une base d'index globale du projet
- des actions contextuelles intelligentes
- un moteur de refactorisation

- un orchestrateur d’outils (Git, tests, debug, Docker…)

L’éditeur n’est qu’une petite partie de l’ensemble.

Ce qu’IntelliJ fait très bien

- indexation profonde du code
- refactorings fiables
- navigation précise
- UX guidée
- configuration quasi invisible

Le coût caché

- lourdeur (RAM / CPU)
- lenteur au démarrage
- faible scriptabilité
- fonctionnement interne opaque
- personnalisation limitée en profondeur

Tu es productif, mais dépendant.

1.2 — Mythe à déconstruire : “Neovim est juste un éditeur”

C’est faux.

Neovim moderne est une **plateforme IDE composable**.

Il fournit :

- un moteur d’édition extrêmement performant
- une API Lua complète
- une intégration native du LSP
- une architecture modulaire
- un chargement paresseux des fonctionnalités

Neovim ne fait pas moins.
Il fait différemment.

1.3 — IDE monolithique vs IDE composable

IntelliJ

- tout est intégré
- tout est lié
- beaucoup d’abstraction
- peu de visibilité interne

Neovim

- chaque brique est explicite :
  - LSP
  - formatter
  - linter
  - Git
  - debug
- chaque brique est :
  - observable
  - remplaçable
  - configurable

Avec Neovim, tu sais ce qui se passe.

1.4 — Pourquoi des développeurs expérimentés migrent

Les développeurs qui passent à Neovim ne le font pas par snobisme.

Ils le font parce qu'ils veulent :

- comprendre leurs outils
- les maîtriser
- travailler de la même façon en local et en SSH
- avoir un workflow cohérent partout

Cas typiques :

- backend
- fullstack
- DevOps
- monorepos
- projets long terme

1.5 — Ce que Neovim fait aussi bien qu'IntelliJ

- navigation
- rename
- code actions
- diagnostics
- Git
- debug
- tests

Fonctionnellement, Neovim peut remplacer IntelliJ dans la majorité des cas modernes.

1.6 — Ce que Neovim fait mieux

- démarrage quasi instantané
- faible consommation de ressources
- scriptabilité totale
- portabilité parfaite
- automatisation naturelle
- contrôle fin du workflow

Neovim devient ton IDE, pas “un IDE”.

1.7 — Limites honnêtes de Neovim

Neovim n’est pas magique.

IntelliJ reste supérieur pour :

- Java / Kotlin très lourds
- debug JVM complexe
- analyse statique propriétaire
- onboarding de profils très juniors

Pour JS / TS / React / backend / DevOps :
Neovim est parfaitement adapté.

1.8 — Le piège principal : vouloir utiliser Neovim comme IntelliJ

Mauvaise approche :

- chercher les menus
- chercher la souris
- vouloir tout voir
- vouloir tout mémoriser

Bonne approche :

- accepter le mode Normal
- utiliser Which-key
- apprendre par couches
- pratiquer régulièrement

1.9 — Le nouveau mental model

IntelliJ :

> “Je cherche une action dans l’interface”

Neovim :

> “Je compose une action avec le clavier”

Les actions sont des verbes, pas des boutons.

1.10 — Exercice de réflexion

Questions à te poser :

- qu’est-ce que j’utilise vraiment dans IntelliJ ?
- qu’est-ce que je n’utilise jamais ?
- quelles actions je fais 100 fois par jour ?

Tu verras que 80 % de ton usage repose sur :

- navigation
- recherche
- rename
- Git

1.11 — Checklist de fin de chapitre

☐ Je comprends le modèle composable
☐ Je ne cherche plus à imiter IntelliJ
☐ J’accepte la phase d’apprentissage
☐ Je garde IntelliJ comme filet temporaire

Chapitre 2 — Le mental model Vim (sans douleur)

Objectif du chapitre

À la fin de ce chapitre, tu dois :

- comprendre pourquoi Vim fonctionne par modes
- arrêter de lutter contre ce modèle
- penser en intentions, pas en touches
- éviter les frustrations classiques

2.1 — Le problème des éditeurs classiques

Dans les éditeurs classiques :

- le clavier sert à tout
- les raccourcis s’accumulent
- la souris reste nécessaire
- la charge cognitive augmente

2.2 — Le postulat fondamental de Vim

Pourquoi utiliser le même mode pour écrire et agir ?

Vim répond :

- un mode = une intention
- aucune ambiguïté
- aucun conflit

2.3 — Les modes essentiels

Mode Normal

- mode par défaut
- navigation
- commandes
- 80 à 90 % du temps

Mode Insertion

- saisie de texte
- à quitter dès que possible

Mode Visuel

- sélection explicite
- outil ponctuel

Mode Commande

- actions globales

2.4 — Le réflexe le plus important

Si tu n’es pas sûr :

```
Esc
```

C’est ton bouton reset mental.

2.5 — Vim n’est pas une liste de raccourcis

Vim fonctionne par grammaire :

```
[action] + [objet]
```

Tu décris ce que tu veux faire.
Vim s’occupe du reste.

2.6 — Pourquoi ça devient naturel

Au début :

- lenteur
- hésitation
- retours fréquents en Normal

Puis :

- gestes automatiques
- intentions claires
- fluidité

2.7 — Erreurs fréquentes

- rester en insertion
- vouloir tout apprendre
- comparer chaque geste à IntelliJ

2.8 — Which-key comme filet de sécurité

Règle d’or :

> si tu hésites → `\<leader\>` → observe

Tu n’as pas besoin de tout mémoriser.

2.9 — Exercice guidé

- ouvre un fichier
- reste en mode Normal
- appuie sur `\<leader\>`
- observe
- `Esc`
- répète

2.10 — Checklist de fin de chapitre

☐ Je sais toujours dans quel mode je suis
☐ J'utilise `Esc` automatiquement
☐ Je n'ai plus peur du mode Normal

Chapitre 3 — Navigation sans souris (devenir efficace rapidement)

Objectif du chapitre

À la fin de ce chapitre, tu dois :

- naviguer dans un projet sans utiliser la souris
- ouvrir n'importe quel fichier en quelques secondes
- comprendre les notions de fichier, buffer et fenêtre
- remplacer définitivement Ctrl+P, Ctrl+Tab et l'arborescence cliquable
- adopter un modèle de navigation plus fiable que celui d'IntelliJ

3.1 — Le vrai problème de la navigation dans les IDE classiques

Dans IntelliJ, la navigation repose sur :

- l'arborescence du projet
- Ctrl+P pour ouvrir un fichier
- Ctrl+Tab pour naviguer entre fichiers

Cela fonctionne tant que :

- le projet est petit
- peu de fichiers sont ouverts
- le contexte est simple

Dans des projets réels (monorepos, gros frontends, backend legacy), cela devient :

- lent
- visuel
- imprécis
- fatigant mentalement

Le problème n'est pas l'outil, mais le modèle mental.

3.2 — Le modèle de navigation de Neovim

Neovim repose sur trois concepts fondamentaux :

Fichier

- un fichier sur le disque
- pas forcément ouvert

Buffer

- un fichier chargé en mémoire
- peut être visible ou non

Fenêtre (window)

- une vue sur un buffer

Un buffer n’est pas un onglet.
Une fenêtre n’est pas un fichier.

Ce découplage est essentiel pour comprendre Neovim.

3.3 — Pourquoi ce modèle est supérieur

Avantages :

- aucun risque de “perdre” un fichier
- navigation instantanée
- multitâche naturel
- moins de charge cognitive

Inconvénient :

- nécessite un petit temps d’adaptation

Une fois compris, ce modèle devient plus fiable que celui des onglets.

3.4 — Ouvrir un fichier (remplacer Ctrl+P)

Commande principale :

`\<leader\>ff`

Cette commande :

- lance une recherche floue
- parcourt le projet
- ouvre un fichier immédiatement

Tu ne dois pas :

- scroller dans la liste
- chercher visuellement
- taper le chemin complet

Tu dois :

- taper quelques lettres
- faire confiance à la recherche

3.5 — Rechercher dans tout le projet (Ctrl+Shift+F)

Commande :

```
\<leader\>fg
```

Cette recherche :

- utilise ripgrep
- est extrêmement rapide
- affiche le contexte

Elle est idéale pour :

- retrouver une fonction
- comprendre un flux
- localiser une logique métier

3.6 — Naviguer entre fichiers ouverts (buffers)

Commandes essentielles :

```
]b buffer suivant
[b buffer précédent
```

C’est l’équivalent conceptuel de Ctrl+Tab, mais sans interface visuelle.

Fermer un buffer :

```
\<leader\>bd
```

Fermer tous les autres :

```
\<leader\>bo
```

Fermer un buffer ne supprime jamais le fichier sur le disque.

3.7 — Explorateur de fichiers (usage ponctuel)

Commande :

```
\<leader\>e
```

L’explorateur sert à :

- comprendre la structure

- créer / supprimer des fichiers

Il ne doit pas devenir ton outil principal de navigation.

3.8 — Navigation par symboles

Commande :

`\<leader\>fw`

Elle permet de :

- naviguer entre fonctions
- trouver des composants
- explorer un fichier complexe

Très utile en React / TypeScript.

3.9 — Aller / retour dans l'historique

Commandes fondamentales :

- Ctrl+o revenir en arrière
- Ctrl+i avancer

Ces commandes fonctionnent comme l'historique d'un navigateur.

3.10 — Exercice pratique

- ouvre un projet réel
- utilise uniquement <leader>ff pour ouvrir des fichiers
- navigue avec]b et [b
- ferme des buffers inutilement
- n'utilise pas la souris

3.11 — Checklist de fin de chapitre

☐ Je comprends fichiers / buffers / fenêtres
☐ J'utilise <leader>ff automatiquement
☐ Je n'ai plus besoin de l'arborescence
☐ Ma navigation est plus fluide

Chapitre 4 — Édition efficace (écrire moins, faire plus)

Objectif du chapitre

À la fin de ce chapitre, tu dois :

- éditer du code sans sélectionner à la souris
- modifier du texte précisément
- réduire le nombre de frappes inutiles
- comprendre pourquoi Vim est plus précis que les éditeurs classiques

---

4.1 — Le problème de la sélection visuelle

Dans les IDE classiques :

- on sélectionne
- on agit
- on corrige la sélection

Ce modèle est :

- imprécis
- lent
- source d'erreurs

Vim part du principe inverse :

> décrire ce que l'on veut modifier

4.2 — Commandes d'édition essentielles

Supprimer :

```
x supprimer un caractère
dd supprimer une ligne
```

Copier / coller :

```
yy copier une ligne
p coller après
P coller avant
```

Dupliquer une ligne :

```
yyp
```

4.3 — Insertion intelligente

```
i insérer avant
a insérer après
o nouvelle ligne en dessous
O nouvelle ligne au-dessus
```

Les commandes o et O accélèrent énormément l’édition.

4.4 — Le réflexe fondamental

Dès que tu as fini d’écrire :

```
Esc
```

Le mode Normal est le mode de contrôle.

4.5 — Commenter du code

Ligne courante :

```
gcc
```

Sélection :

```
v
gc
```

4.6 — Annuler et refaire

```
u annuler
Ctrl+r refaire
```

Tu peux expérimenter sans crainte.

4.7 — Exercice pratique

- supprime une ligne
- annule
- duplique une ligne
- commente / décommente
- ajoute une ligne avec o / O

4.8 — Checklist de fin de chapitre

☐ J’utilise dd, yyp, gcc naturellement
☐ Je reviens toujours en mode Normal
☐ Je sélectionne moins
☐ Mon édition est plus précise

Chapitre 5 — LSP : le cerveau de ton IDE

Objectif du chapitre

À la fin de ce chapitre, tu dois :

- comprendre ce qu’est réellement un LSP
- savoir ce qu’il fournit
- faire confiance aux refactorings
- utiliser Neovim comme un IDE intelligent

5.1 — Pourquoi l’intelligence du code a été externalisée

Historiquement, les IDE faisaient tout eux-mêmes.

Le LSP a introduit :

- un protocole standard
- une séparation éditeur / intelligence
- une meilleure interopérabilité

Neovim s’appuie sur ce modèle.

5.2 — Qu’est-ce qu’un LSP

Un LSP est :

- un processus externe
- spécialisé par langage
- capable d’analyser ton projet

Neovim :

- affiche
- déclenche
- orchestre

Le LSP :

- comprend
- analyse
- décide

5.3 — Fonctionnalités fournies par le LSP

- aller à la définition
- trouver les références
- rename
- diagnostics
- documentation
- code actions

5.4 — Commandes essentielles

Aller à la définition :

```
gd
```

Trouver les références :

```
gr
```

Documentation :

```
K
```

Renommer :

```
\<leader\>rn
```

Code actions :

```
\<leader\>ca
```

5.5 — Pourquoi les refactorings sont sûrs

Le LSP :

- connaît le scope
- comprend les types
- respecte les imports

Ce n’est pas une recherche texte.

5.6 — Diagnostics et navigation

Naviguer :

```
]d
[d
```

Liste globale :

`\<leader\>xx`

5.7 — Mason et la gestion des serveurs

Commande :

`:Mason`

Mason permet :

- d’installer les LSP
- de gérer les versions
- d’éviter la pollution système

5.8 — Exercice pratique

- ouvre un fichier TypeScript
- utilise gd et gr
- renomme une variable avec <leader>rn
- corrige une erreur avec <leader>ca

5.9 — Checklist de fin de chapitre

☐ J’utilise gd et gr
☐ Je fais confiance au rename
☐ Je lis les diagnostics calmement
☐ Je comprends le rôle du LSP

Chapitre 6 — Qualité de code & refactor (sans douleur)

Objectif du chapitre

À la fin de ce chapitre, tu dois :

- maintenir un code propre automatiquement
- comprendre le rôle de chaque outil (LSP, formatter, linter)
- refactoriser sans casser
- arrêter de perdre du temps sur le style
- atteindre un confort équivalent à IntelliJ sur la qualité de code

---

6.1 — Le faux problème de la qualité de code

Dans beaucoup d’équipes, la qualité de code devient :

- subjective
- source de débats
- chronophage

Avec Neovim, la qualité doit devenir :

- automatique
- explicite
- non émotionnelle

6.2 — Séparation claire des responsabilités

Dans un workflow sain :

LSP

- compréhension du code
- types
- symboles
- refactorings sûrs

Formatter

- indentation
- retours à la ligne
- style

Linter

- règles métier
- conventions d’équipe
- détection de bugs potentiels

IntelliJ mélange ces rôles.
Neovim les sépare volontairement.

6.3 — Le formatage automatique (format on save)

Le formatage doit :

- s’exécuter à la sauvegarde
- être prévisible
- ne jamais te bloquer

Résultat :

- plus de discussions sur le style
- commits propres
- charge mentale réduite

Tu écris.
Neovim corrige.

6.4 — ESLint et diagnostics intelligents

Les messages ESLint apparaissent comme diagnostics.

Bon réflexe : 1. naviguer à l'erreur 2. lire le message 3. essayer une action 4. corriger manuellement si nécessaire

---

6.5 — Refactor propre : quand utiliser quoi

Renommer :

`\<leader\>rn`

À utiliser pour :

- variables
- fonctions
- props React
- types

Toujours préférer cela à une recherche texte.

6.6 — Exercice pratique

- introduis une violation ESLint
- corrige-la avec <leader>ca
- renomme une variable avec <leader>rn
- sauvegarde et observe le formatage

6.7 — Checklist de fin de chapitre

☐ Le formatage ne me préoccupe plus
☐ Je comprends LSP / formatter / linter
☐ Je refactorise sans peur
☐ Mon code est plus propre sans effort

Chapitre 7 — Diagnostics & debugging mental

Objectif du chapitre

À la fin de ce chapitre, tu dois :

- lire une erreur calmement
- identifier la cause racine
- corriger méthodiquement
- réduire le stress lié aux bugs

7.1 — Le debugging est un problème cognitif

Les erreurs provoquent :

- stress
- précipitation
- corrections hasardeuses

Un bon développeur :

- ralentit
- lit
- comprend
- agit

7.2 — Ce qu’est un diagnostic

Un diagnostic est :

- une information
- localisée
- contextualisée

Ce n’est ni une attaque, ni une urgence.

7.3 — Navigation entre diagnostics

Diagnostic suivant :

```
]d
```

Diagnostic précédent :

```
[d
```

Vue globale des diagnostics :

```
\<leader\>xx
```

7.4 — Processus de correction recommandé

1. aller à l’erreur
2. lire le message entièrement
3. comprendre le pourquoi
4. essayer une action automatique si disponible
5. corriger manuellement
6. sauvegarder
7. observer

7.5 — Erreurs en cascade

Ne corrige jamais les symptômes.
Corrige la première erreur logique.

Une seule erreur peut en masquer dix autres.

7.6 — Exercice pratique

- crée une erreur TypeScript volontairement
- observe les diagnostics
- navigue avec]d et [d
- corrige uniquement la cause racine
- sauvegarde et observe la disparition des erreurs

7.7 — Checklist de fin de chapitre

☐ Je lis les erreurs calmement
☐ Je ne panique plus
☐ Je corrige méthodiquement
☐ Je fais confiance aux diagnostics

Chapitre 8 — Git sans quitter Neovim

Objectif du chapitre

À la fin de ce chapitre, tu dois :

- utiliser Git sans sortir de Neovim
- comprendre chaque action Git
- produire des commits propres
- remplacer le Git tool window d’IntelliJ

8.1 — Le problème des workflows Git trop visuels

Les interfaces graphiques :

- masquent les intentions
- encouragent les commits larges
- réduisent la compréhension

Neovim favorise :

- l'intention
- la précision
- la lisibilité de l'historique

8.2 — Gitsigns : voir les changements dans le code

Navigation entre les changements :

```
]c    hunk suivant
[c    hunk précédent
```

Actions sur les changements :

```
\<leader\>hs    stage hunk
\<leader\>hr    reset hunk
\<leader\>hp    preview hunk
\<leader\>hb    blame ligne
```

8.3 — Pourquoi le staging par hunk est essentiel

Un commit correspond à une intention claire.

Ne mélange jamais :

- refactor
- feature
- correction de bug

Un historique propre est un outil de communication.

8.4 — Neogit : interface Git complète

Ouvrir Neogit :

```
\<leader\>gg
```

Dans Neogit :

```
s    stage
u    unstage
c    commit
P    push
F    pull
?    aide
```

Tout se fait sans quitter le clavier.

8.5 — Exercice pratique

- modifie plusieurs lignes dans un fichier
- observe les hunks avec Gitsigns
- stage un seul hunk
- crée un commit clair
- consulte l'historique

8.6 — Checklist de fin de chapitre

☐ Je stage par intention
☐ Mes commits sont lisibles
☐ Je ne quitte plus Neovim pour Git
☐ Je comprends ce que je versionne

Chapitre 9 — Terminal, tests et debug

Objectif du chapitre

À la fin de ce chapitre, tu dois :

- lancer des commandes sans quitter Neovim
- exécuter des tests efficacement
- utiliser le debug quand nécessaire
- conserver le contexte du projet

9.1 — Pourquoi le contexte est crucial

Chaque alt-tab :

- casse le flux
- augmente la fatigue mentale
- fragmente la concentration

Neovim vise la continuité du contexte.

9.2 — Terminal intégré

Ouvrir un terminal :

`\<leader\>tf`

Afficher ou masquer le terminal :

`Ctrl+\`

Le terminal est utilisé pour :

- lancer des tests
- exécuter des scripts
- démarrer un serveur de développement
- inspecter rapidement une sortie

9.3 — Tests avec Neovim

Commandes typiques pour les tests :

```
\<leader\>tt    lancer le test le plus proche
\<leader\>tf    lancer les tests du fichier
\<leader\>ta    lancer tous les tests
```

L'objectif n'est pas de mémoriser les touches, mais de comprendre le flux.

9.4 — Debug avec nvim-dap

Commandes clés du debugger :

```
\<leader\>db    ajouter ou retirer un breakpoint
\<leader\>dc    continuer l'exécution
\<leader\>do    step over
\<leader\>di    step into
\<leader\>du    afficher ou masquer l'interface debug
```

Le debugger est un outil ponctuel, pas un mode de travail permanent.

9.5 — Exercice pratique

- ouvre un projet avec des tests
- lance un test depuis Neovim
- fais volontairement échouer un test
- corrige le code
- relance le test
- ajoute un breakpoint
- observe l'exécution

9.6 — Checklist de fin de chapitre

☐ Je garde le contexte dans Neovim
☐ Je lance les tests sans friction
☐ Je sais quand utiliser le debugger
☐ Mon workflow est fluide

Chapitre 10 — Sessions, projets et continuité

Objectif du chapitre

À la fin de ce chapitre, tu dois :

- reprendre ton travail instantanément
- gérer plusieurs projets proprement
- ne plus perdre de contexte
- travailler plus sereinement au quotidien

10.1 — Le coût caché de la reprise de contexte

À chaque reprise de travail, tu dois souvent :

- retrouver les fichiers ouverts
- te rappeler ce que tu faisais
- reconstituer mentalement le contexte

Ce coût est invisible, mais réel.
Il fatigue et ralentit.

10.2 — Projet vs session

Il est crucial de distinguer les deux notions.

Un projet correspond à :

- un dossier racine
- un dépôt Git
- une configuration LSP

Une session correspond à :

- des buffers ouverts
- une disposition de fenêtres
- un état de travail précis

Un projet peut avoir plusieurs sessions.
Une session appartient toujours à un projet.

10.3 — Workflow de session recommandé

Bonnes pratiques :

- une session par tâche
- pas de session éternelle
- fermeture volontaire en fin de travail

Une session doit servir ton intention actuelle, pas accumuler l'historique.

10.4 — Travailler sur plusieurs projets

Pour plusieurs projets en parallèle :

- une instance Neovim par projet
- pas de mélange de buffers
- pas de mélange de contextes

Neovim démarre vite.
Il vaut mieux plusieurs instances claires qu'une seule instance confuse.

10.5 — Exercice pratique

- ouvre un projet
- ouvre plusieurs fichiers liés à une tâche
- quitte Neovim
- rouvre la session
- vérifie que le contexte est intact

Observe le gain mental par rapport à une reprise manuelle.

10.6 — Checklist de fin de chapitre

- ☐ Je reprends mon travail rapidement
- ☐ Je distingue projet et session
- ☐ Je ne mélange plus les contextes
- ☐ Je travaille plus sereinement

Chapitre 11 — React & TypeScript en profondeur (IDE-ready)

Objectif du chapitre

À la fin de ce chapitre, tu dois :

- naviguer efficacement dans un projet React / TypeScript
- comprendre comment le LSP interprète JSX et TSX
- refactoriser des composants sans casser
- gérer props, types et imports sereinement
- atteindre un confort équivalent ou supérieur à IntelliJ pour React

---

11.1 — Pourquoi React est un bon révélateur d'IDE

React cumule :

- beaucoup de fichiers
- des composants imbriqués
- des props typées
- des hooks
- une logique parfois distribuée

Si ton IDE gère bien React, il gère la majorité des cas réels.

11.2 — Ce que voit réellement le LSP dans un fichier TSX

Pour le LSP :

- JSX est une structure syntaxique
- un composant est une fonction
- les props sont des types
- les hooks sont des appels analysables

React n'est pas magique.
C'est du TypeScript enrichi.

11.3 — Navigation entre composants

Aller à la définition :

`gd`

Trouver les usages :

`gr`

Ces commandes fonctionnent :

- pour des composants locaux
- pour des composants importés
- pour des bibliothèques typées

11.4 — Comprendre rapidement un composant inconnu

Workflow recommandé : 1. ouvrir le composant 2. lire la signature des props 3. inspecter les types 4. trouver les usages 5. revenir en arrière

Tu comprends un composant sans parcourir tout le projet.

11.5 — Props et types : refactor sans danger

Renommer une prop :

```
\<leader\>rn
```

Le LSP :

- met à jour le type
- met à jour tous les usages
- respecte les scopes

Une recherche texte est dangereuse ici.

11.6 — Imports automatiques

Cas classique :

- tu utilises un composant non importé

Solution :

```
\<leader\>ca
```

L'import est ajouté :

- au bon endroit
- avec le bon nom
- sans erreur

11.7 — Hooks et erreurs courantes

Erreurs fréquentes :

- hook mal utilisé
- dépendances manquantes

- type incorrect

Approche saine :

- lire le diagnostic
- comprendre le message
- appliquer une action si possible
- corriger manuellement sinon

11.8 — Exercice pratique

- ouvre un composant React
- navigue avec gd et gr
- renomme une prop
- corrige les diagnostics
- vérifie les imports

11.9 — Checklist de fin de chapitre

☐ Je navigue facilement dans React
☐ Je comprends props et types
☐ Je refactorise sans peur
☐ Neovim est confortable pour le front

Chapitre 12 — Workflow React réel (de la feature au commit)

Objectif du chapitre

À la fin de ce chapitre, tu dois :

- développer une feature React complète
- enchaîner navigation, édition, refactor, tests et Git
- conserver un flux continu
- produire des commits clairs et prêts à relire

12.1 — Pourquoi un workflow réel est indispensable

Les tutoriels montrent rarement :

- du code existant
- des erreurs
- des ajustements

- des tests cassés

Ce chapitre simule une vraie journée de développement.

12.2 — Scénario de départ

Contexte :

- projet React avec TypeScript
- composant existant nommé UserCard
- nouvelle fonctionnalité : afficher le rôle utilisateur

12.3 — Étape 1 : comprendre avant d’agir

- ouvrir le composant
- lire les props
- inspecter les types
- vérifier les usages

Aucun code n’est écrit tant que ce n’est pas clair.

12.4 — Étape 2 : modifier la source de vérité

- ajouter la prop dans le type
- sauvegarder
- observer les diagnostics

Les erreurs apparaissent volontairement.

12.5 — Étape 3 : corriger les usages

- naviguer entre diagnostics
- corriger chaque appel
- sauvegarder régulièrement

Le LSP te guide.

12.6 — Étape 4 : ajuster le rendu

- modifier le JSX
- afficher la nouvelle information
- laisser le formatter agir

12.7 — Étape 5 : refactor léger

- renommer une variable si nécessaire
- clarifier un nom
- extraire du JSX

Toujours avec des outils assistés.

12.8 — Étape 6 : tests

- lancer les tests
- lire l'erreur
- corriger
- relancer

Les tests confirment ton intention.

12.9 — Étape 7 : Git

- vérifier les changements
- stage par intention
- écrire un commit clair

Exemple de message : feat(user): display user role in UserCard

---

12.10 — Checklist de fin de chapitre

☐ Je développe une feature complète
☐ Mon workflow est fluide
☐ Mes commits sont propres
☐ Je garde le contexte jusqu'au bout

Chapitre 13 — Comprendre et maîtriser sa configuration Neovim

Objectif du chapitre

À la fin de ce chapitre, tu dois :

- comprendre la structure de ta configuration
- savoir où modifier quoi
- diagnostiquer un problème
- ne plus avoir peur de toucher à ta config

13.1 — Pourquoi comprendre sa config est essentiel

Copier une configuration sans la comprendre mène à :

- la peur de modifier
- des bugs incompris
- une dépendance à long terme

L'objectif de ce chapitre est l'autonomie.

13.2 — Démarrage de Neovim

Au lancement, Neovim : 1. charge le core 2. lit le fichier init.lua 3. initialise le gestionnaire de plugins 4. charge les plugins 5. attache les LSP

Rien n'est magique.
Tout est traçable.

13.3 — Structure typique d'une configuration moderne

Structure courante :

```
~/.config/nvim/
├── init.lua
├── lua/
│   ├── config/
│   └── plugins/
```

Le dossier config définit le comportement.
Le dossier plugins définit les outils.

13.4 — Le rôle du gestionnaire de plugins

Un gestionnaire moderne gère :

- l'installation
- le chargement différé
- les dépendances
- les performances

Il ne doit jamais être invisible.

13.5 — Mason et la gestion des outils externes

Mason permet de gérer :

- serveurs LSP
- formatters
- linters
- debuggers

Sans polluer le système global.

13.6 — Diagnostiquer un problème

Outils essentiels :

- vérifier l’état des plugins
- inspecter les LSP actifs
- consulter les messages d’erreur

La majorité des problèmes sont visibles sans ajouter de logs.

13.7 — Checklist de fin de chapitre

☐ Je comprends la structure de ma config
☐ Je sais où modifier quoi
☐ Je sais diagnostiquer un problème
☐ Je ne crains plus ma configuration

Chapitre 14 — Personnaliser sans casser

Objectif du chapitre

À la fin de ce chapitre, tu dois :

- personnaliser avec intention
- éviter les anti-patterns courants
- garder une configuration stable
- faire évoluer ton IDE sereinement

14.1 — Pourquoi les configurations cassent

Les configurations cassent souvent à cause de :

- trop de plugins
- absence d’intention claire
- accumulation sans nettoyage

Le problème n’est pas Neovim, mais la méthode.

14.2 — Règle numéro un : une intention par modification

Avant chaque ajout, pose-toi la question :

- quel problème précis est-ce que je résous ?

Si tu n’as pas de réponse claire, n’ajoute rien.

14.3 — Chargement différé obligatoire

Un plugin doit être chargé :

- sur un événement
- sur une touche
- sur un type de fichier

Jamais tout au démarrage.

14.4 — Supprimer est une compétence

Un plugin inutile :

- ralentit l’éditeur
- augmente la complexité
- fragilise la configuration

Supprimer est une action saine.

14.5 — Checklist de fin de chapitre

☐ J’ajoute avec intention
☐ Je supprime sans peur
☐ Ma configuration est lisible
☐ Mon IDE reste stable

Chapitre 15 — Autonomie et évolution long terme

Objectif du chapitre

À la fin de ce chapitre, tu dois :

- être autonome avec Neovim
- faire évoluer ton environnement sans casser
- conserver ton IDE sur le long terme
- ne plus dépendre d’un outil opaque

15.1 — L’objectif n’était pas de quitter IntelliJ

L’objectif réel était :

- reprendre le contrôle
- comprendre ses outils
- réduire la friction quotidienne

IntelliJ devient un choix, pas une dépendance.

15.2 — Ce que tu sais faire maintenant

Tu sais :

- naviguer efficacement
- éditer précisément
- refactoriser sans peur
- lire les diagnostics calmement
- travailler avec Git
- développer des fonctionnalités complètes
- maintenir ta configuration

15.3 — La règle des 80 / 20

Vingt pour cent des fonctionnalités couvrent quatre-vingts pour cent de l’usage.

Le reste est du confort, pas une priorité.

15.4 — Entretenir sa compétence

Bonnes habitudes :

- pratiquer un peu chaque jour
- nettoyer régulièrement
- améliorer de façon ciblée

La stabilité vient de la discipline.

15.5 — Conclusion

Neovim n’est pas seulement un éditeur.
Ce n’est pas simplement un IDE.

C’est un environnement de travail conscient.

Tu n’as pas appris des raccourcis.
Tu as appris une méthode.

Annexes — Références, muscle memory et checklists

Ces annexes sont conçues pour :

- consultation rapide
- impression
- révision quotidienne
- sécurisation du workflow long terme

Annexe A — Cheatsheet IntelliJ → Neovim

Navigation et recherche

IntelliJ	Action	Neovim
Ctrl+P	Ouvrir un fichier	\<leader\>ff
Ctrl+Shift+F	Rechercher dans le projet	\<leader\>fg
Ctrl+B	Aller à la définition	gd
Alt+F7	Trouver les usages	gr
Ctrl+O	Structure du fichier	\<leader\>fw
Ctrl+Tab	Fichier suivant	]b
Ctrl+Shift+Tab	Fichier précédent	[b
–	Historique arrière	Ctrl+o
–	Historique avant	Ctrl+i

Édition

IntelliJ	Action	Neovim
Ctrl+D	Dupliquer une ligne	yyp
Ctrl+Y	Supprimer une ligne	dd
Ctrl+/	Commenter une ligne	gcc
Alt+Flèche haut	Déplacer une ligne	ddkP
Alt+Flèche bas	Déplacer une ligne	ddp
Ctrl+Z	Annuler	u
Ctrl+Shift+Z	Refaire	Ctrl+r

Refactor et LSP

IntelliJ	Action	Neovim
Shift+F6	Renommer	\<leader\>rn
Alt+Entrée	Code actions	\<leader\>ca

Ctrl+Q	Documentation	K
Ctrl+P	Signature	Ctrl+k

Diagnostics

Action	Neovim
Diagnostic suivant	]d
Diagnostic précédent	[d
Liste globale des diagnostics	\<leader\>xx

Git

IntelliJ	Action	Neovim
Fenêtre Git	Statut Git	\<leader\>gg
–	Hunk suivant	]c
–	Hunk précédent	[c
–	Stage hunk	\<leader\>hs
–	Reset hunk	\<leader\>hr
–	Preview hunk	\<leader\>hp
–	Blame ligne	\<leader\>hb

Terminal, tests et debug

Action	Neovim
Terminal intégré	\<leader\>tf
Test le plus proche	\<leader\>tt
Tests du fichier	\<leader\>tf
Tous les tests	\<leader\>ta
Ajouter un breakpoint	\<leader\>db
Continuer le debug	\<leader\>dc
Afficher ou masquer l’UI debug	\<leader\>du

Annexe B — Muscle memory quotidienne

À pratiquer chaque jour pendant 5 à 10 minutes :

- ouvrir un fichier avec <leader>ff
- naviguer entre buffers avec]b et [b
- aller à la définition avec gd
- revenir avec Ctrl+o
- renommer une variable avec <leader>rn
- corriger un diagnostic avec <leader>ca

- stage un hunk avec <leader>hs
- commit avec une intention claire

La répétition crée l’automatisme.

Annexe C — Checklist de workflow sain

Avant chaque session :

- je sais sur quoi je travaille
- une session correspond à une tâche

Pendant le travail :

- je reste en mode Normal autant que possible
- je fais confiance aux diagnostics
- je stage par intention

Après le travail :

- je ferme proprement
- je laisse un état clair
- je peux reprendre facilement demain

Annexe D — Rappels mentaux essentiels

- Neovim est composable
- le clavier exprime des intentions
- le LSP comprend le code
- les outils travaillent pour toi
- la simplicité est un choix actif

Fin des annexes

Ce guide est conçu pour être relu, consulté partiellement, et utilisé sur le long terme.

Neovim n’est pas appris en un jour. Il se construit par couches successives.