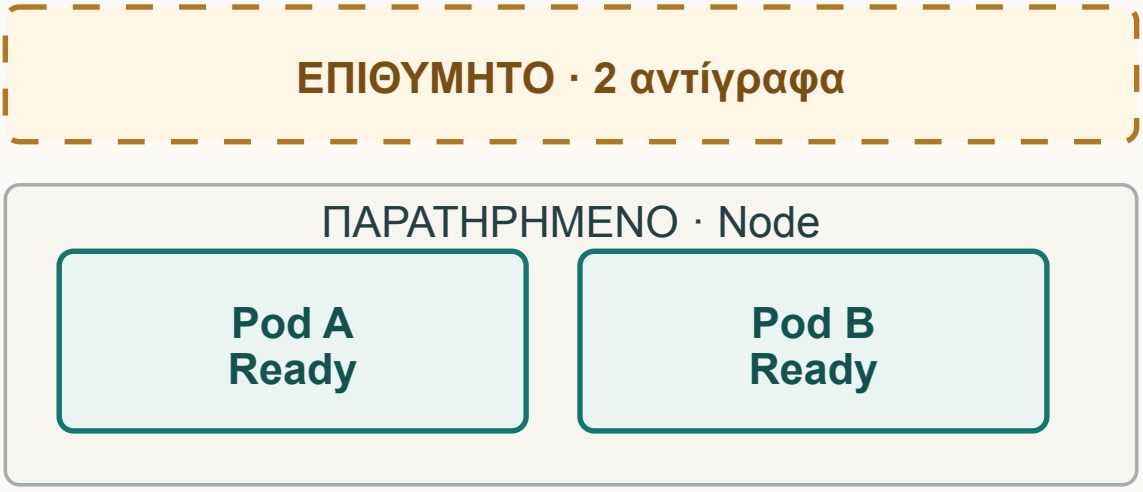
ΕΝΑ ΟΠΤΙΚΟ ΒΙΒΛΙΟ ΣΤΑ ΕΛΛΗΝΙΚΑ

Kubernetes

για όποιον ήδη κάνει deploy

Από το Docker και το Kamal
στους controllers, στα Pods και στο cluster.



24 Ένας χάρτης που μεγαλώνει

ΚΕΦΑΛΑΙΑ · ΠΡΩΤΗ ΠΛΗΡΗΣ ΓΡΑΦΗ · 09.2026

ΠΡΙΝ ΑΡΧΙΣΕΙΣ

Να φαίνεται αυτό που συμβαίνει.

Αυτό το βιβλίο γράφτηκε για κάποιον που ήδη κάνει deploy. Ξέρει Docker, Rails, Postgres και έναν proxy. Χρειάζεται να δει τι προστίθεται όταν ανάμεσα στη δήλωσή του και στην εφαρμογή μπαίνει ένα cluster.

Τα 24 κεφάλαια ακολουθούν την ίδια εφαρμογή. Πρώτα θα τρέχει μόνο web. Έπειτα θα αποκτήσει δρομολόγηση, ρυθμίσεις, βάση και worker. Κάθε καινούριο αντικείμενο προστίθεται στον ίδιο χάρτη. Οι σκόπιμες βλάβες έχουν αρχική κατάσταση, έλεγχο και επαναφορά.

Πώς θα το διαβάσεις

Τα κεφάλαια 1 έως 7 είναι η είσοδος. Από το 8 έως το 14 χτίζεις την εφαρμογή. Στα 15 έως 19 τη βάζεις υπό πίεση. Τα 20 και 21 οργανώνουν τη διάγνωση. Το 22 συνθέτει ολόκληρο deploy και τα 23 και 24 τοποθετούν τα επόμενα εργαλεία στη σωστή θέση.

Δούλεψε από τον φάκελο του βιβλίου. Τα manifests που βλέπεις στο PDF εισάγονται από τα πραγματικά αρχεία του εργαστηρίου. Οι ετικέτες και τα διαγράμματα περιγράφουν συμπεριφορές. Ονόματα, UIDs και χρόνοι που αλλάζουν δεν παρουσιάζονται ως σταθερές του Kubernetes.

Η κατάσταση αυτής της έκδοσης

Πρόκειται για την πρώτη πλήρη γραφή. Η συγκεντρωτική καταγραφή επαλήθευσης βρίσκεται στο τέλος και στο VERIFICATION.md. Ξεχωρίζει τους πραγματικούς ελέγχους από τις εκκρεμότητες. Η επιτυχία ενός τοπικού πειράματος δεν αποτελεί υπόσχεση διαθεσιμότητας στην παραγωγή.

Το επιθυμητό έχει κεχριμπαρένιο διακεκομμένο περίγραμμα. Το παρατηρημένο έχει πετρόλ συνεχές περίγραμμα. Τα ονόματα και οι γραμμές κρατούν τη διάκριση και χωρίς χρώμα.

Ο ΧΑΡΤΗΣ ΤΟΥ ΒΙΒΛΙΟΥ

Περιεχόμενα

ΚΕΦΑΛΑΙΟ 1

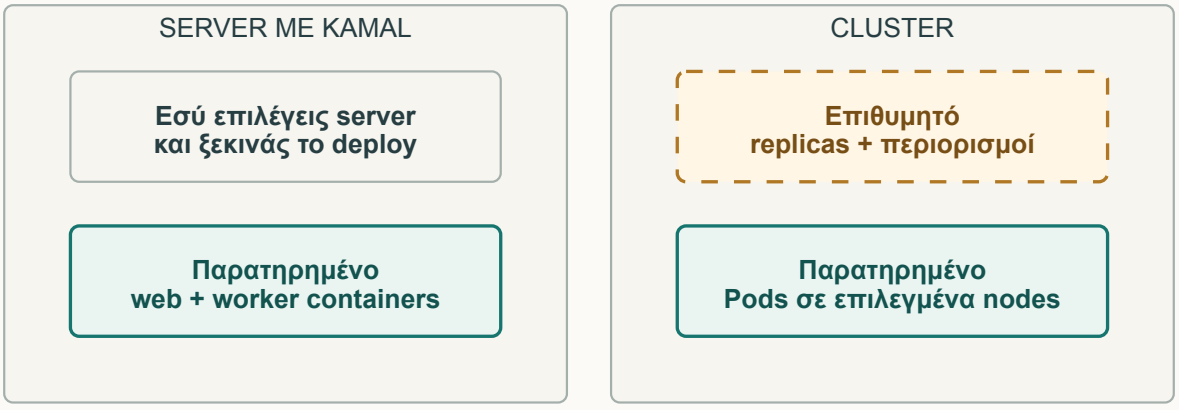
Τι σου δίνει που δεν σου δίνει το Kamal

Το επόμενο εργαλείο πρέπει να λύσει υπαρκτό πρόβλημα.

Έχεις μια Rails εφαρμογή, έναν worker και μια βάση. Το Kamal χτίζει το image, το στέλνει στους servers και κάνει deploy. Έχεις logs, ένα πιστοποιητικό που ανανεώνεται και ένα backup που έχεις δοκιμάσει να επαναφέρεις. Αν αυτά καλύπτουν τη δουλειά σου, το Kubernetes δεν είναι κάποιο χρέος που έμεινε απλήρωτο.

Η μετάβαση αρχίζει να έχει νόημα όταν το πρόβλημα αλλάζει. Δεν σε απασχολεί πια μόνο πώς θα περάσει μια έκδοση σε γνωστούς servers. Σε απασχολεί ποιος διατηρεί τα αντίγραφα, πού θα τοποθετηθεί μια νέα διεργασία, πώς περιγράφεται η πρόσβασή της και πώς εφαρμόζονται οι ίδιοι κανόνες σε πολλές εφαρμογές.

Το Kubernetes προσθέτει ένα κοινό API και controllers που δουλεύουν διαρκώς. Αποθηκεύεις τη δήλωση «θέλω τρία αντίγραφα αυτής της εφαρμογής». Το σύστημα παρατηρεί τα αντικείμενα που έχει και προσπαθεί να φτάσει εκεί. Ο στόχος μένει ενεργός και αφού κλείσεις το laptop σου.



Σχήμα 1.1 · Η ευθύνη μετακινείται. Στον μεμονωμένο server επιλέγεις εσύ τη θέση κάθε υπηρεσίας. Στο cluster δηλώνεις περιορισμούς και ο scheduler επιλέγει θέση για κάθε νέο Pod.

Η διαφορά δεν είναι ότι ο ένας κόσμος έχει αυτοματισμούς και ο άλλος όχι. Το Docker μπορεί ήδη να επανεκκινεί containers. Η καινούρια μονάδα αυτοματισμού είναι ένα σύνολο αντικειμένων που περιγράφει εφαρμογή, πλήθος, πρόσβαση και πόρους σε περισσότερα μηχανήματα.

Το κέρδος που θα μετρήσουμε

Ένας worker node σταματά. Πάνω του υπήρχε ένα από τα δύο web Pods. Έχεις διαθέσιμο χώρο αλλού. Ένας controller μπορεί να δημιουργήσει αντικαταστάτη και ο scheduler να τον τοποθετήσει σε λειτουργικό node. Αυτή είναι χρήσιμη συμπεριφορά. Δεν είναι στιγμιαία ούτε άνευ όρων.

Πρέπει να ανιχνευθεί η βλάβη, να περάσει η σχετική αναμονή, να επιτρέπουν οι κανόνες τοποθέτησης άλλη θέση και να μπορεί να ξεκινήσει η εφαρμογή. Αν το μοναδικό volume βρίσκεται στον χαμένο server, η ύπαρξη ελεύθερης CPU αλλού δεν σου φέρνει αυτόματα τα δεδομένα.

Σχήμα 1.2 · Η αναπλήρωση χρειάζεται διαθέσιμη θέση. Το σχήμα δείχνει stateless web Pods. Δεν υπόσχεται μεταφορά τοπικού δίσκου.

Στο κεφάλαιο 5 θα διαγράψουμε ένα Pod για να δούμε τον υπεύθυνο του πλήθους. Στο 17 θα σταματήσουμε πραγματικό worker του kind. Εκεί θα κρατήσουμε χρόνους και αποτελέσματα HTTP. Έτσι θα ξεχωρίσουμε την αναπλήρωση από την εντύπωση ότι μια εφαρμογή δεν έλειψε ποτέ.

Το δεύτερο κέρδος είναι η επανάληψη με κοινό λεξιλόγιο. Deployment, Service, Secret, Role. Η ομάδα περιγράφει τη λειτουργία με αντικείμενα που διαβάζονται από τα ίδια εργαλεία. Ένα admission policy μπορεί να ελέγχει αιτήματα πριν γίνουν δεκτά. Ένα pipeline μπορεί να περιμένει το rollout, χωρίς να γνωρίζει ποιος server φιλοξενεί κάθε αντίγραφο.

Το κοινό API δεν εξαφανίζει τις διαφορές των υποδομών. Ένα volume, ένα LoadBalancer και μια ταυτότητα για υπηρεσίες του cloud εξακολουθούν να εξαρτώνται από συγκεκριμένες υλοποιήσεις. Το ίδιο YAML δεν αποτελεί απόδειξη ίδιων εγγυήσεων σε δύο παρόχους.

Τι αναλαμβάνεις εσύ

Αποκτάς περισσότερα μέρη που πρέπει να καταλαβαίνεις όταν κάτι αποτύχει. Ένα Pod μπορεί να μη βρίσκει θέση. Ένα image μπορεί να μην κατεβαίνει. Μια εφαρμογή μπορεί να είναι υγιής αλλά να έχει λανθασμένο readiness. Το Service μπορεί να έχει σωστό όνομα και μηδέν χρήσιμα endpoints.

Η ομάδα χρειάζεται τρόπο να αναβαθμίζει το cluster, να παρατηρεί πόρους, να ορίζει δικαιώματα και να διατηρεί δεδομένα και logs. Σε managed cluster κάποια από αυτά τα αναλαμβάνει ο πάροχος. Δεν αναλαμβάνει τη συμβατότητα των migrations σου ή τη λογική του Sidekiq job σου.

Σχήμα 1.3 · Τρεις διαφορετικές ευθύνες. Η λειτουργία του cluster, η λειτουργία της εφαρμογής και η ανθεκτικότητα των δεδομένων χρειάζονται χωριστές αποφάσεις.

Το κόστος δεν είναι μόνο ο λογαριασμός των nodes. Είναι και ο χρόνος διάγνωσης, οι αλλαγές στα pipelines και οι άνθρωποι που πρέπει να καταλαβαίνουν το σύστημα. Δεν θα συγκρίνουμε υποθετικά ποσά. Θα συγκρίνουμε ευθύνες που υπάρχουν στη δική σου εφαρμογή.

> **Από το Kamal στο Kubernetes**
> Η σύνδεση SSH σε συγκεκριμένους servers παραχωρεί μέρος της θέσης της σε αιτήματα προς ένα API. Το δικό σου deploy εξακολουθεί να χρειάζεται σειρά, έλεγχο επιτυχίας και δυνατότητα επαναφοράς. Ο controller δεν γνωρίζει αν η νέα business logic είναι σωστή.

Πότε δεν θα το επέλεγα

Αν μια μικρή ομάδα λειτουργεί λίγες εφαρμογές σε σταθερούς servers και τα deploy είναι προβλέψιμα, θα κρατούσα την υπάρχουσα λύση ώσπου να εμφανιστεί συγκεκριμένη ανάγκη. Αν το βασικό πρόβλημα είναι ένα αργό SQL query, θα κοίταζα πρώτα το query. Αν δεν υπάρχει δοκιμασμένη επαναφορά backup, θα τη διόρθωνα πριν μετακινήσω τη βάση.

Το ίδιο ισχύει όταν κανείς δεν μπορεί να αναλάβει τη λειτουργία του cluster ή όταν η εφαρμογή εξαρτάται από έναν τοπικό δίσκο χωρίς σχέδιο μεταφοράς. Το Kubernetes μπορεί να περιγράψει έναν περιορισμό. Δεν μπορεί να τον αναιρέσει επειδή έγραψες περισσότερα replicas.

Η πρακτική άσκηση αυτού του κεφαλαίου δεν χρειάζεται terminal. Γράψε μία πρόταση για μια βλάβη που σήμερα διορθώνεις με το χέρι. Έπειτα γράψε ποιο στοιχείο θα έπρεπε να παρατηρεί ένα σύστημα και ποια ενέργεια θα του επέτρεπες. Αν δεν προκύπτει σαφής βλάβη και σαφής ενέργεια, η αιτιολόγηση της μετάβασης είναι ακόμη αδύναμη.

Μια ενδεικτική λύση. «Χάνω web αντίγραφο όταν σταματά ο server. Θέλω να παρατηρείται η διαθεσιμότητα του node και να επιτρέπεται νέο stateless αντίγραφο αλλού, αφού επιβεβαιωθεί ότι υπάρχει χώρος». Η λέξη stateless και η συνθήκη του χώρου είναι μέρος της λύσης.

Το βιβλίο ξεκινά από αυτή τη διάκριση. Θα μάθεις τι αναλαμβάνει το cluster και πού σταματά. Στο επόμενο κεφάλαιο θα δώσουμε ονόματα στα αντικείμενα που θα συναντάς συνέχεια.

Οι βασικές έννοιες τεκμηριώνονται στο Overview του Kubernetes και στους Controllers.

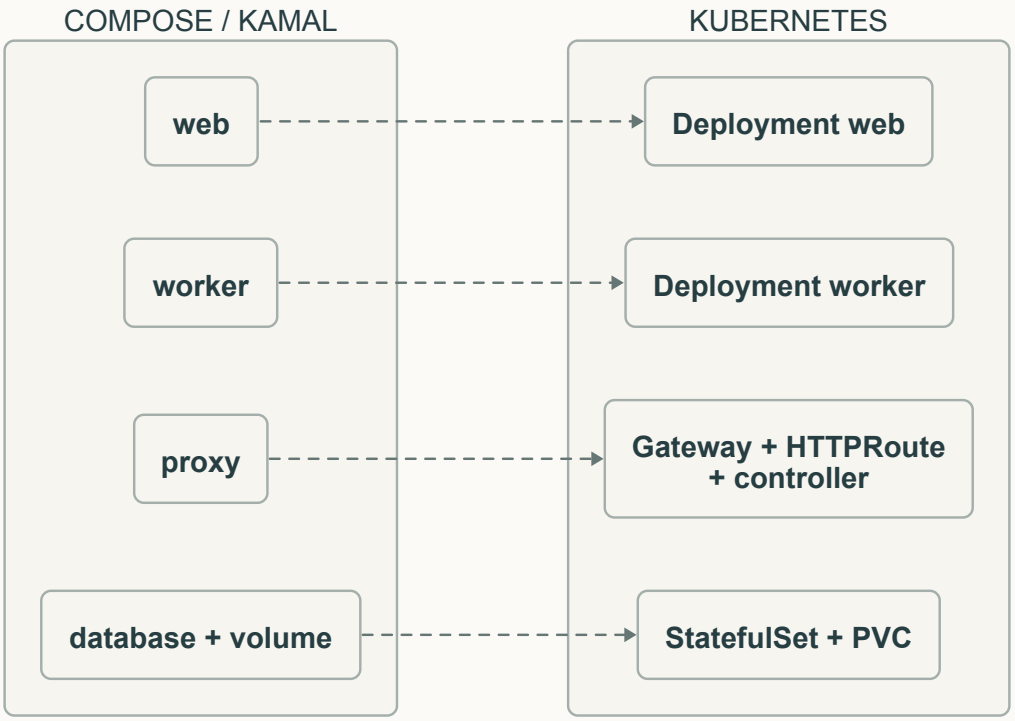
ΚΕΦΑΛΑΙΟ 2

Το λεξικό της μετάβασης

Η ίδια εφαρμογή, με άλλες μονάδες ευθύνης.

Ας κρατήσουμε γνωστή την εφαρμογή. Ένα web process δέχεται HTTP, ένας worker εκτελεί jobs, η Postgres αποθηκεύει εγγραφές και το Redis κρατά την ουρά. Υπάρχει proxy μπροστά και μερικές ρυθμίσεις που αλλάζουν ανά περιβάλλον. Τα ονόματα δεν αλλάζουν την ανάγκη τους.

Στο compose τα περιγράφεις συνήθως ως υπηρεσίες, δίκτυα και volumes. Στο Kubernetes θα χρησιμοποιήσεις περισσότερα αντικείμενα. Αυτό δεν είναι γραφειοκρατία μόνο για να γράφεις YAML. Ένα Deployment διαχειρίζεται τον κύκλο των Pods. Ένα Service διατηρεί την πρόσβαση προς αυτά. Ένα PVC ζητά αποθήκευση. Έχουν διαφορετικές διάρκειες ζωής.



Σχήμα 2.1 · Η ίδια εφαρμογή στις δύο πλευρές. Οι γραμμές είναι αντιστοιχίσεις εννοιών. Δεν δείχνουν ότι ένα αρχείο compose μετατρέπεται μηχανικά σε ισοδύναμη υποδομή.

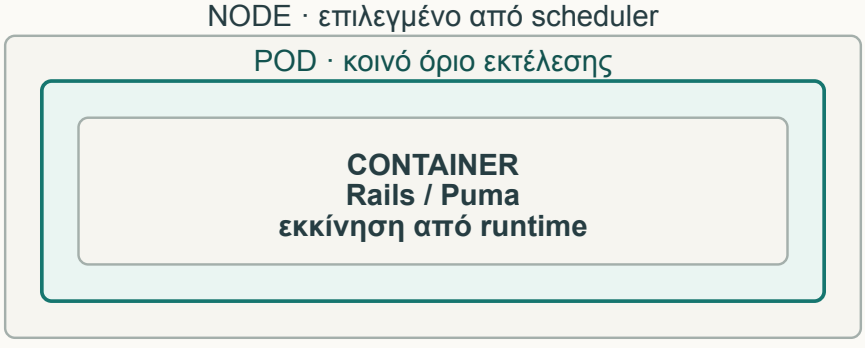
Το web και ο worker μπορούν να χρησιμοποιούν το ίδιο image με διαφορετική εντολή εκκίνησης. Θα έχουν χωριστά Deployments, ώστε να αυξάνεις τους workers χωρίς να αυξάνεις τα web Pods και να ελέγχεις ξεχωριστά το shutdown τους. Η κοινή καταγωγή από το ίδιο repository δεν τα κάνει ίδια λειτουργική μονάδα.

Η αντιστοίχιση έχει όριο

Ο πίνακας είναι χάρτης εισόδου. Η τελευταία στήλη είναι εξίσου χρήσιμη με τη μεσαία. Εκεί βρίσκονται οι περιπτώσεις όπου η προηγούμενη γνώση σε οδηγεί σε λάθος πρόβλεψη.

Αυτό που ξέρεις	Το αντικείμενο που θα δεις	Πού σταματά η αναλογία
web υπηρεσία	Deployment	Δεν είναι ένα συγκεκριμένο container. Διαχειρίζεται ReplicaSets και Pods.
worker υπηρεσία	Χωριστό Deployment	Η ολοκλήρωση ή επανάληψη ενός job παραμένει ευθύνη της εφαρμογής.
δημοσιευμένο port	Service και πρόσβαση από έξω	Το ClusterIP δεν είναι από μόνο του προσβάσιμο από το Mac.
nginx ή Traefik	Gateway, HTTPRoute και controller	Τα αντικείμενα δηλώνουν κανόνες. Η υλοποίηση περνά την κίνηση.
αρχείο env	ConfigMap και Secret	Οι μεταβλητές ενός ήδη εκκινημένου process δεν αλλάζουν με το αντικείμενο.
named volume	PVC και παροχή αποθήκευσης	Ο δίσκος δεν μετακινείται αυτόματα σε οποιονδήποτε node.
healthcheck	startup, readiness, liveness	Κάθε probe έχει διαφορετική συνέπεια όταν αποτύχει.

Θα δεις συχνά τη λέξη Pod δίπλα στο container. Το Pod είναι αντικείμενο που περιγράφει ένα ή περισσότερα containers με κοινό περιβάλλον εκτέλεσης. Στο βιβλίο αρχίζουμε με ένα container ανά Pod. Δεν χρειάζεται να προσθέσεις sidecar για να είναι το παράδειγμα «κανονικό» Kubernetes.



Σχήμα 2.2 · Το Pod είναι το όριο που βλέπει ο scheduler. Το container είναι η μονάδα που ξεκινά το runtime στο επιλεγμένο node.

Τα ρήματα του deploy

Στο Kamal διαβάζεις ένα σύνολο εντολών ως διαδικασία. Στο Kubernetes θα χρησιμοποιείς ρήματα προς αντικείμενα που παραμένουν. Το `apply` ενημερώνει δηλώσεις. Το `get` παρατηρεί. Το `rollout status` περιμένει μια συνθήκη προόδου. Το `delete` αφαιρεί ένα συγκεκριμένο αντικείμενο.

Αυτό που έκανες	Η πρώτη αντίστοιχη ερώτηση
`kamal deploy`	Εφαρμόστηκε το Pod template και ολοκληρώθηκε το rollout;
`kamal rollback`	Ποιο προηγούμενο Pod template επαναφέρω και είναι συμβατό με τη σημερινή βάση;
`kamal app logs`	Ποιο Pod, ποιο container και ποια εκτέλεσή του θέλω να διαβάσω;
`kamal app exec`	Σε ποιο ακριβώς Pod θα εκτελεστεί η εντολή;
`docker ps`	Ποια Pods βλέπω στο συγκεκριμένο namespace;
restart από systemd	Έγινε restart container ή δημιουργήθηκε άλλο Pod;

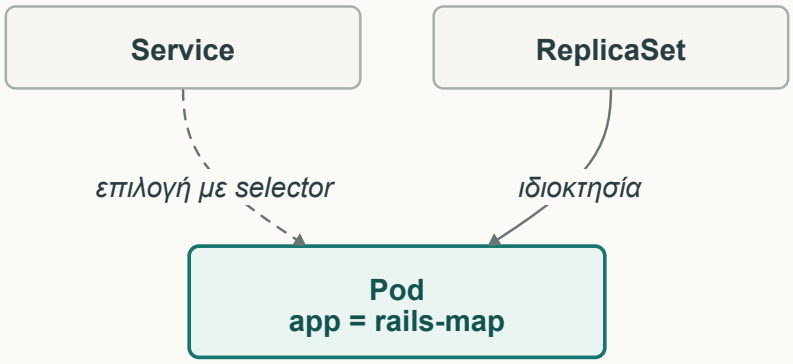
Το `rollout undo` δεν είναι μια γενική μηχανή του χρόνου. Επαναφέρει το Pod template ενός Deployment. Δεν αναιρεί ένα migration ούτε επιστρέφει ένα Secret στην προηγούμενη τιμή. Αν άλλαξαν τέσσερα αντικείμενα, πρέπει να ξέρεις ποιο από αυτά επαναφέρεις.

Η αλλαγή νοοτροπίας φαίνεται και στα logs. Ένα όνομα που υπήρχε το πρωί μπορεί να έχει φύγει το απόγευμα. Η φράση «μπες στον web server» χρειάζεται τώρα ακριβέστερο στόχο. Θα χρησιμοποιούμε labels για να βρίσκουμε τα μέλη μιας εφαρμογής και UIDs για να ξεχωρίζουμε αντικείμενα.

Όνομα, επιλογή και ιδιοκτησία

Τρεις συνδέσεις εμφανίζονται συνέχεια. Το όνομα σού επιτρέπει να αναφερθείς σε ένα αντικείμενο. Ο selector επιλέγει ένα σύνολο με βάση labels. Το owner reference καταγράφει σχέση ιδιοκτησίας.

Ένα Service μπορεί να επιλέγει Pods μέσω labels χωρίς να τα κατέχει. Αν διαγράψεις το Service, δεν διαγράφεις τα Pods του Deployment. Αντίθετα, η διαγραφή ενός ιδιοκτήτη μπορεί να ενεργοποιήσει τη συλλογή των εξαρτώμενων αντικειμένων, ανάλογα με την πολιτική διαγραφής.



Σχήμα 2.3 · Το ReplicaSet κατέχει Pods. Το Service τα επιλέγει. Το ίδιο σύνολο Pods συμμετέχει σε δύο διαφορετικές σχέσεις.

> **Από το Kamal στο Kubernetes**
> Το σταθερό όνομα μιας υπηρεσίας σε βοηθά να βρίσκεις την εφαρμογή. Δεν σου λέει ποιο process θα απαντήσει. Στο Kubernetes θα χωρίσουμε τη σταθερή πρόσβαση από τα Pods που αλλάζουν από κάτω.

Άσκηση. Θέλεις να αυξήσεις μόνο τους workers, να αλλάξεις τον HTTP host και να δώσεις άλλη δημόσια τιμή ρύθμισης στο web. Γράψε ποιο είδος αντικειμένου αφορά κάθε αλλαγή. Έπειτα σκέψου ποια αλλαγή χρειάζεται νέα Pods.

Λύση. Αλλάζεις replicas στο worker Deployment, hostname στο HTTPRoute και δεδομένα στο ConfigMap. Η νέα τιμή ConfigMap δεν ανανεώνει μια ήδη ληφθείσα μεταβλητή περιβάλλοντος. Για αυτή τη χρήση χρειάζεσαι νέα εκκίνηση Pods. Η αλλαγή host συνήθως αφορά τη ρύθμιση του proxy και δεν απαιτεί να ξαναξεκινήσει η Rails εφαρμογή.

Δεν χρειάζεται να απομνημονεύσεις ακόμη τη σύνταξη. Χρειάζεται να ξεχωρίζεις τις ευθύνες. Στο επόμενο κεφάλαιο θα αποκτήσεις ένα μικρό cluster στο οποίο θα μπορείς να ελέγχεις κάθε αντιστοίχιση.

Για τους ακριβείς ορισμούς δες τα Objects, Labels και selectors και Owners and dependents.

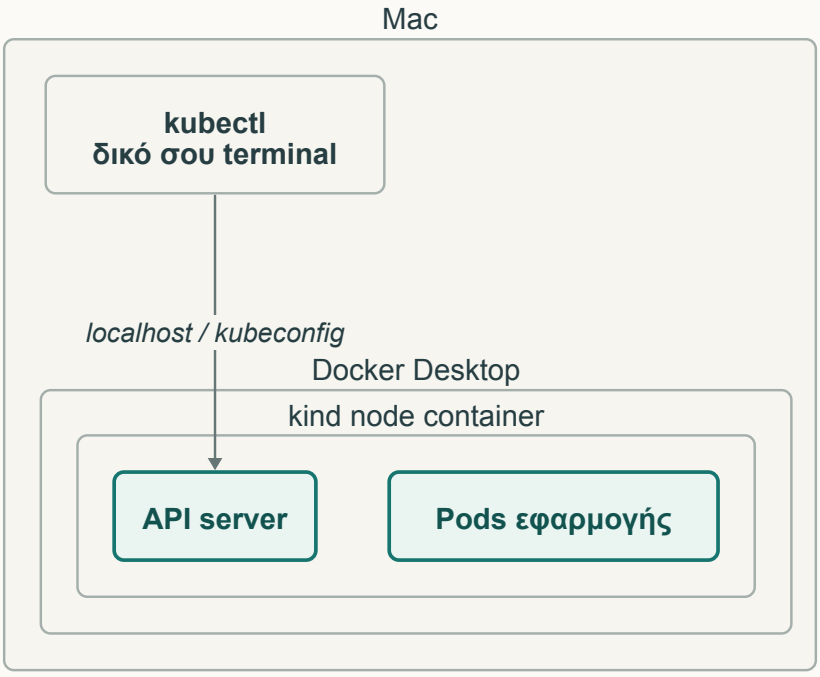
ΚΕΦΑΛΑΙΟ 3

kind, kubectl και k9s στο Mac

Ένα cluster που ξέρεις πού βρίσκεται.

Θα δουλέψουμε τοπικά με kind. Τα nodes του είναι Docker containers, οπότε μπορείς να τα δεις με εργαλεία που ήδη χρησιμοποιείς. Αυτό κάνει την αρχιτεκτονική λιγότερο αφηρημένη και επιτρέπει να επαναλαμβάνουμε πειράματα χωρίς λογαριασμό cloud.

Το minikube είναι επίσης χρήσιμο τοπικό περιβάλλον. Εδώ επιλέγουμε kind επειδή μας αρκούν αρχεία διαμόρφωσης για διαφορετικά προφίλ nodes και επειδή στο κεφάλαιο 17 θέλουμε να σταματήσουμε ένα συγκεκριμένο worker container. Η επιλογή υπηρετεί αυτό το εργαστήριο.



Σχήμα 3.1 · Το kind δημιουργεί nodes μέσα στο Docker. Το kubeconfig συνδέει το kubectl με το API του συγκεκριμένου cluster.

Χρειάζεσαι Docker Desktop σε λειτουργία, Homebrew, kind, kubectl, Python 3 και make. Το περιβάλλον αναφοράς χρησιμοποιεί Kubernetes 1.34.11 και kubectl 1.34. Η έκδοση του node είναι κλειδωμένη με digest στο script. Αν εγκαθιστάς αλλού το εργαστήριο, έλεγξε και την έκδοση του client. Η υποστηριζόμενη απόσταση του kubectl από τον API server είναι μία minor έκδοση προς τα πάνω ή προς τα κάτω.

```
TERMINAL

brew install kind
docker --context desktop-linux info
kind version
kubectl version --client
```

Για εγκατάσταση του σωστού kubectl ακολούθησε την επίσημη οδηγία για macOS, με έλεγχο checksum. Δεν αλλάζουμε έναν υπάρχοντα client χωρίς να εξετάσουμε ποια άλλα clusters διαχειρίζεται.

Η πρώτη εκκίνηση

Άνοιξε terminal στον φάκελο `k8s/` του βιβλίου. Η πρώτη εντολή ετοιμάζει το cluster και τη μικρή web εφαρμογή. Η δεύτερη επιλέγει το kubeconfig του βιβλίου μόνο για αυτό το terminal.

```
TERMINAL

make cluster
export KUBECONFIG="$PWD/build/kubeconfig"
kubectl config current-context
kubectl get nodes
kubectl get pods
```

Το context πρέπει να γράφει `kind-k8s-book`. Το node πρέπει να φτάσει σε `Ready`. Τα Pods της εφαρμογής πρέπει να γίνουν `1/1`. Το setup περιμένει με timeout. Αν δεν φτάσει σε αυτή την κατάσταση, η επόμενη δουλειά είναι η διάγνωση του setup, όχι η συνέχιση στα επόμενα manifests.

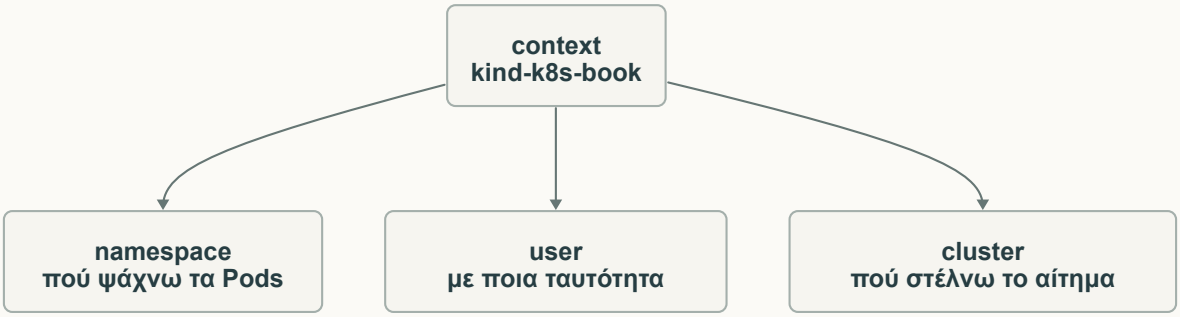
```
YAML

kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
  - role: control-plane
```

Αυτό είναι το μικρό προφίλ `lab/kind.yaml`. Ένας node φιλοξενεί το control plane και τα Pods του παραδείγματος. Το ότι λέγεται control-plane δεν απαγορεύει στα workloads να τρέχουν πάνω του σε αυτό το τοπικό setup.

Το script δημιουργεί namespace `k8s-book` και το επιλέγει στο δικό του kubeconfig. Δεν στηρίζεται στο τρέχον context κάποιου άλλου project. Όταν ξανατρέχεις το setup, ελέγχει το όνομα και το σημάδι ιδιοκτησίας του τοπικού cluster πριν το χρησιμοποιήσει.

Context και namespace

Το context είναι ένας συνδυασμός cluster, ταυτότητας και προεπιλεγμένου namespace. Η αλλαγή namespace δεν αλλάζει cluster. Η αλλαγή context μπορεί να αλλάξει και τα τρία. Αυτή η διάκριση πρέπει να προηγείται κάθε εντολής που γράφει στο API.



Σχήμα 3.2 · Το context επιλέγει cluster και ταυτότητα. Το namespace περιορίζει τον συνηθισμένο κατάλογο αντικειμένων μέσα στο επιλεγμένο cluster. Δεν είναι δεύτερο cluster.

TERMINAL

```
kubectl config current-context
kubectl config view --minify \
  -o jsonpath='{.contexts[0].context.namespace}{"\n"}'
kubectl get pods -n k8s-book
kubectl get pods -A
```

Η τελευταία εντολή εμφανίζει Pods από όλα τα namespaces. Θα δεις και στοιχεία του συστήματος. Δεν σημαίνει ότι το `-A` σου δίνει περισσότερα δικαιώματα. Ζητά ευρύτερη λίστα και ο API server εξακολουθεί να ελέγχει την ταυτότητά σου.

Για ένα χειροκίνητο βήμα που θέλεις να είναι απολύτως σαφές, μπορείς να γράψεις και τα δύο επάνω στην ίδια εντολή.

TERMINAL

```
kubectl --context kind-k8s-book -n k8s-book get pods
```

> **Από το Kamal στο Kubernetes**
> Ο server στόχος στο deploy έχει αντίστοιχη σημασία με το cluster στόχο εδώ. Το namespace βοηθά στην οργάνωση μέσα στο cluster. Τα δικαιώματα θα τα ορίσουμε χωριστά με RBAC στο κεφάλαιο 19.

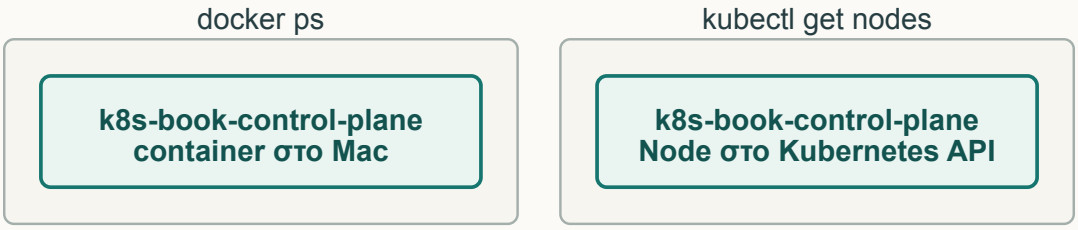
Βρες το node στο Docker

Το όνομα του Kubernetes node και το όνομα του container του kind συνδέονται στο τοπικό περιβάλλον. Δες και τις δύο πλευρές.

TERMINAL

```
kubectl get nodes -o wide
docker --context desktop-linux ps \
  --filter label=io.x-k8s.kind.cluster=k8s-book \
  --format '{{.Names}}'
```

Το φίλτρο είναι μέρος της εντολής. Σε ένα Docker Desktop μπορεί να υπάρχουν και άλλα projects. Η ύπαρξη του Docker δεν σημαίνει ότι όλα τα containers του ανήκουν στο εργαστήριο.



Σχήμα 3.3 · Δύο ονόματα για την ίδια τοπική μονάδα. Το kubectl βλέπει node, το Docker βλέπει container του kind. Τα Pods βρίσκονται ένα επίπεδο βαθύτερα.

Το προαιρετικό k9s προσφέρει μια οθόνη παρατήρησης στο terminal. Μπορείς να το ανοίξεις με το kubeconfig του βιβλίου και να κινηθείς ανάμεσα σε Pods, logs και describe. Οι εντολές του βιβλίου παραμένουν σε kubectl, ώστε κάθε βήμα να έχει σαφή, αντιγράψιμη μορφή.

TERMINAL

```
k9s --kubeconfig "$PWD/build/kubeconfig" \
  --context kind-k8s-book --namespace k8s-book
```

Η εντολή προϋποθέτει ότι έχεις εγκαταστήσει το k9s. Δεν χρειάζεται για τα πειράματα. Το UI δεν αλλάζει το μοντέλο δικαιωμάτων του API.

Επαναφορά και μικρή άσκηση

Το cluster του βιβλίου είναι αναλώσιμο. Η αναδημιουργία του είναι διαφορετική πράξη από τη διαγραφή ενός Pod. Αφαιρεί ολόκληρο το τοπικό περιβάλλον και επηρεάζει τα δεδομένα που ζουν μέσα του. Κάν’ την στην αρχή, πριν προσθέσουμε βάσεις και εγγραφές.

TERMINAL

```
bash scripts/cluster-down.sh
make cluster
export KUBECONFIG="$PWD/build/kubeconfig"
kubectl get nodes
kubectl get pods
```

Πριν αναδημιουργήσεις το cluster, κράτησε όσα δεδομένα χρειάζεσαι έξω από τα nodes του. Η διαγραφή του εργαστηρίου δεν είναι δοκιμή της διατήρησης ενός PVC μετά από αντικατάσταση Pod.

Άσκηση. Άνοιξε δεύτερο terminal. Επίλεξε το kubeconfig του βιβλίου και βρες το namespace χωρίς να βασιστείς στο prompt του shell. Σύγκρινε τη λίστα Pods με και χωρίς `-A`.

Λύση. Χρησιμοποίησε το ίδιο `export`, το `config current-context` και το `config view --minify` της προηγούμενης ενότητας. Τα δύο web Pods εμφανίζονται και στις δύο λίστες. Τα Pods του συστήματος εμφανίζονται μόνο στην ευρύτερη αναζήτηση, εφόσον επιτρέπεται η πρόσβαση.

Το kind quick start και η πολιτική version skew είναι οι πηγές για το setup και τη συμβατότητα των εργαλείων.

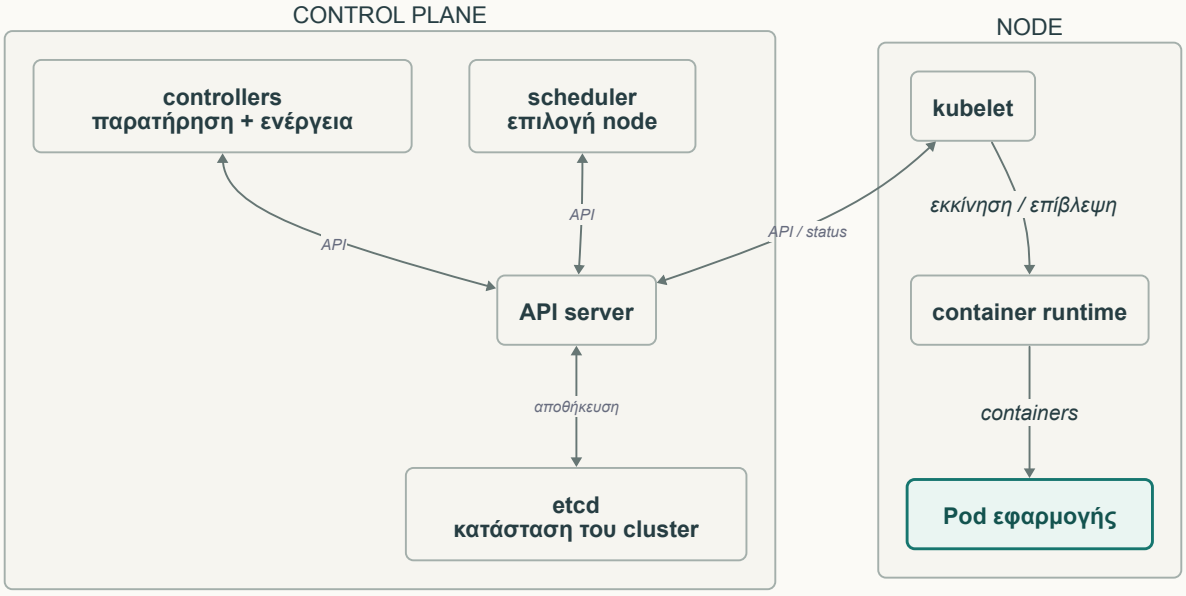
ΚΕΦΑΛΑΙΟ 4

Η ανατομία του cluster

Το apply δεν ταξιδεύει κατευθείαν στο container.

Όταν στέλνεις μια δήλωση, το kubectl μιλά με τον API server. Δεν συνδέεται με SSH στο node και δεν τρέχει `docker run`. Ο API server ελέγχει το αίτημα και αποθηκεύει την αποδεκτή αλλαγή. Άλλες συνιστώσες παρατηρούν όσα αποθηκεύτηκαν και κάνουν τη δική τους δουλειά.

Αυτό το μοίρασμα ευθυνών εξηγεί πολλές καθυστερήσεις που αρχικά φαίνονται παράξενες. Μια δήλωση μπορεί να είναι αποδεκτή, αλλά το Pod να περιμένει θέση. Το Pod μπορεί να έχει θέση, αλλά το image να μην κατεβαίνει. Ο server μπορεί να ξεκίνησε, αλλά το readiness να μην επιτρέπει ακόμη κίνηση.



Σχήμα 4.1 · Ο χάρτης αναφοράς του control plane και ενός node. Τα βέλη διαχείρισης περνούν από το API. Η κίνηση της εφαρμογής θα προστεθεί σε χωριστή διαδρομή.

Στο kind αυτά μπορεί να φιλοξενούνται στο ίδιο container node. Οι λογικές ευθύνες παραμένουν διαφορετικές. Σε μεγαλύτερο cluster μπορεί να βρίσκονται σε διαφορετικά μηχανήματα και να έχουν περισσότερα αντίγραφα. Το σχήμα δεν υπόσχεται μία διεργασία ανά φυσικό server.

Τα πέντε σημεία που χρειάζεσαι τώρα

Ο **API server** είναι η είσοδος για αιτήματα διαχείρισης. Εκεί γίνονται έλεγχοι ταυτότητας, δικαιωμάτων και αποδοχής αντικειμένων. Ένα επιτυχημένο HTTP αίτημα προς αυτόν δεν είναι αίτημα προς τη Rails εφαρμογή.

Το **etcd** κρατά τα δεδομένα του cluster που χρησιμοποιεί το API. Δεν είναι η βάση της Rails εφαρμογής. Ένα backup του etcd αφορά την κατάσταση του control plane. Δεν αποτελεί αυτόματα backup των εγγραφών που έχει γράψει η εφαρμογή στην Postgres.

Ο **scheduler** βρίσκει θέση για Pods που δεν έχουν ανατεθεί σε node. Ελέγχει περιορισμούς και διαθέσιμους πόρους, βαθμολογεί κατάλληλες θέσεις και καταγράφει την επιλογή. Δεν κάνει το ίδιο το pull του image.

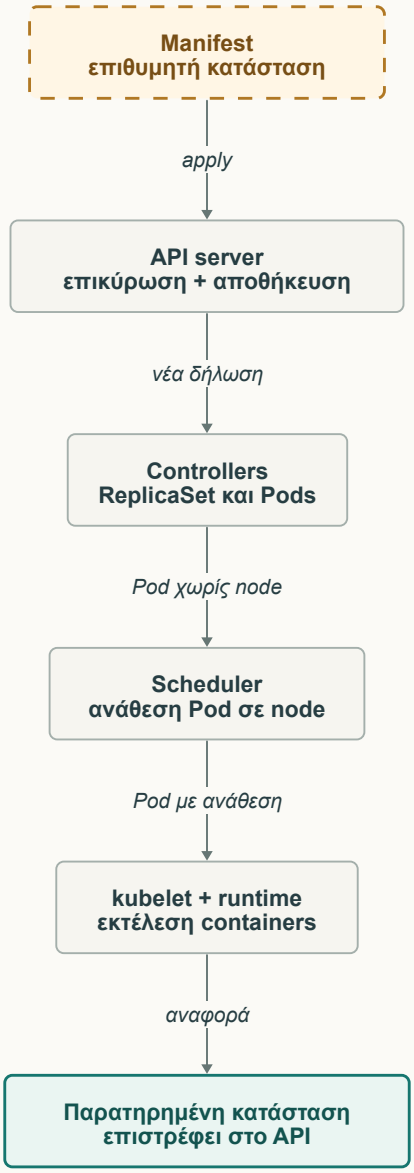
Ο **controller manager** φιλοξενεί πολλούς controllers. Δεν υπάρχει ένας αόριστος «διορθωτής». Ο controller ενός Deployment έχει άλλη δουλειά από εκείνον ενός ReplicaSet ή ενός Job.

Το **kubelet** λειτουργεί σε κάθε node. Παρατηρεί τα Pods που του έχουν ανατεθεί και ζητά από το container runtime να εκτελέσει τα containers. Εκτελεί τους ελέγχους υγείας και αναφέρει κατάσταση. Το containerd του kind δεν είναι το Docker CLI που τρέχεις στο Mac.

Ερώτηση	Πρώτος υπεύθυνος που εξετάζεις
Έγινε δεκτή η δήλωση;	API server
Υπάρχουν τα ζητούμενα εξαρτώμενα αντικείμενα;	Ο αντίστοιχος controller
Βρέθηκε node για το Pod;	Scheduler
Ξεκίνησε το container στο node;	kubelet και runtime
Απάντησε σωστά το HTTP;	Εφαρμογή και διαδρομή κίνησης

Το ταξίδι μιας δήλωσης

Σκέψου ένα Deployment με δύο replicas. Ο API server αποδέχεται το αντικείμενο. Ο Deployment controller διαχειρίζεται ένα ReplicaSet. Ο ReplicaSet controller δημιουργεί τα απαιτούμενα Pods. Ο scheduler τους αναθέτει node. Το kubelet αναλαμβάνει την εκκίνηση των containers.



Σχήμα 4.2 · Η σειρά είναι λογική ακολουθία ευθυνών. Οι συνιστώσες παρατηρούν και ενημερώνουν το API ασύγχρονα. Δεν πρόκειται για μία συγχρονισμένη κλήση από κουτί σε κουτί.

Το `status` ενημερώνεται καθώς οι υπεύθυνοι αναφέρουν τι παρατήρησαν. Μπορεί να διαβάσεις μια παλαιότερη εικόνα για μικρό διάστημα. Για αυτό οι έλεγχοι δεν τελειώνουν στην απάντηση «created» ή «configured». Περιμένουν τη συνθήκη που έχει σημασία και έπειτα ελέγχουν την εφαρμογή.

Αν δεν υπάρχει κατάλληλο node, ο controller μπορεί να έχει κάνει τη δουλειά του και το Pod να παραμένει `Pending`. Αν το image λείπει, μπορεί να έχει κάνει τη δουλειά του και ο scheduler. Η βλάβη αποκτά συγκεκριμένη θέση στον χάρτη.

Δες τα μέρη στο δικό σου cluster

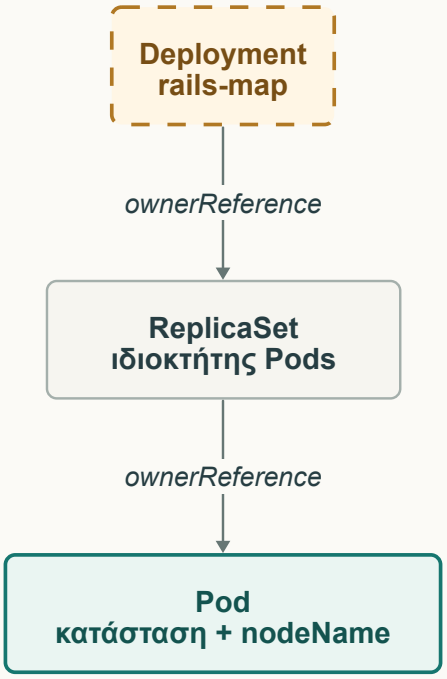
Με το kubeconfig του βιβλίου επιλεγμένο, δες τα Pods του συστήματος και τα βασικά workloads της εφαρμογής.

TERMINAL

```
kubectl -n kube-system get pods
kubectl get nodes
kubectl -n k8s-book get deployment,replicaset,pods
kubectl -n k8s-book describe deployment rails-map
```

Στο kube-system θα βρεις τα αντίστοιχα ονόματα του control plane για το συγκεκριμένο kind node. Το kubelet δεν εμφανίζεται ως ένα ακόμη συνηθισμένο Pod στη λίστα. Είναι διεργασία του node. Το ότι μια συνιστώσα δεν βρίσκεται στο `get pods` δεν σημαίνει ότι δεν λειτουργεί.

Στο `describe deployment` αναζήτησε το ζητούμενο πλήθος, την αναφορά στο ReplicaSet και τα conditions. Στη λίστα Pods δες σε ποιο στάδιο βρίσκεται η εκτέλεση. Συνδέεις αντικείμενα διαφορετικών επιπέδων.



Σχήμα 4.3 · Η ανάγνωση από το API δίνει αντικείμενα και αναφορές. Ο μεγάλος χάρτης σε βοηθά να καταλάβεις σε ποιο επίπεδο ανήκει κάθε γραμμή.

> **Από το Kamal στο Kubernetes**
> Η αποτυχία μιας εντολής deploy συνήθως έχει θέση μέσα στη διαδικασία. Εδώ μπορεί να έχει αποθηκευτεί σωστά η δήλωση και να αποτυγχάνει κάποιο επόμενο στάδιο. Χρειάζεσαι και το αποτέλεσμα του API και την κατάσταση των αντικειμένων που δημιουργήθηκαν.

Μια διαδρομή διάγνωσης πριν από τις βλάβες

Άσκηση. Η εντολή apply επέστρεψε επιτυχία. Το Pod έχει `spec.nodeName` κενό. Ποιο στάδιο θα εξετάσεις πρώτο; Αν έχει node αλλά το container περιμένει με `ImagePullBackOff`, αλλάζει η απάντηση;

Λύση. Στην πρώτη περίπτωση κοιτάς την τοποθέτηση και τα scheduling events. Δεν υπάρχει ακόμη node στο οποίο θα μπορούσες να ψάξεις για την εκκίνηση του συγκεκριμένου container. Στη δεύτερη έχει ολοκληρωθεί η ανάθεση. Εξετάζεις το όνομα του image, το registry, τα δικαιώματα pull και τα events από το kubelet.

Δεν χρειάζεται ακόμη να διαχειριστείς ένα control plane. Χρειάζεται να χρησιμοποιείς τα ονόματα για να εντοπίζεις τη φάση της αποτυχίας. Στο επόμενο κεφάλαιο θα πάρουμε ένα μόνο κομμάτι αυτού του χάρτη, τη διατήρηση των replicas, και θα το δοκιμάσουμε με δύο διαγραφές.

Δες τα Kubernetes components, το Kubernetes API και τον Scheduler για τους επιμέρους ρόλους.

ΚΕΦΑΛΑΙΟ 5

Ο βρόχος συμφιλίωσης

Ζητάς δύο. Σβήνεις ένα.

Έχεις δύο αντίγραφα της Rails εφαρμογής σου. Σβήνεις το ένα Pod με το `kubectl`. Λίγο μετά βλέπεις πάλι δύο. Το καινούριο έχει άλλο όνομα. Δεν εκτέλεσες δεύτερο deploy.

Η εξήγηση βρίσκεται σε κάτι που δεν έσβησες. Το Deployment εξακολουθεί να ζητά δύο αντίγραφα. Ο controller του ReplicaSet βλέπει ότι λείπει ένα από τα Pods που διαχειρίζεται και δημιουργεί άλλο.

Σχήμα 5.1 · Ο στόχος παραμένει δύο, ενώ ο controller παρατηρεί ένα ενεργό Pod. Η διαφορά προκαλεί νέα ενέργεια.

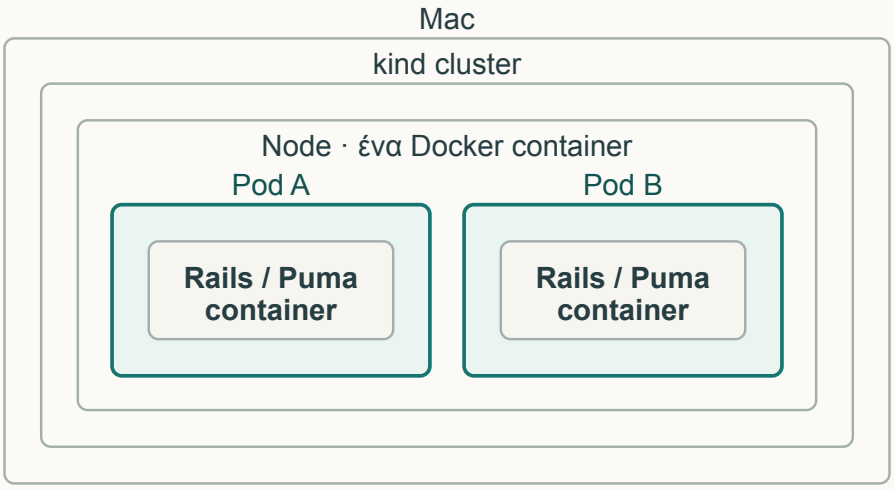
Αυτό είναι το πείραμα του κεφαλαίου. Θα το επαναλάβεις με ένα ανεξάρτητο Pod, χωρίς controller που να διατηρεί αντίγραφα. Εκεί η διαγραφή θα έχει άλλο αποτέλεσμα. Έπειτα θα τερματίσεις μόνο το container ενός Pod και θα ξεχωρίσεις την επανεκκίνηση από την αναπλήρωση.

> **Από το Kamal στο Kubernetes**
> Το `kamal deploy` είναι μια εκτέλεση με αρχή και τέλος. Εδώ το αίτημα για δύο αντίγραφα μένει αποθηκευμένο και ένας controller συνεχίζει να δουλεύει πάνω του. Η αναλογία σταματά στο deploy. Η πολιτική restart του Docker αφορά containers που ήδη υπάρχουν και δεν είναι controller που διατηρεί τον αριθμό των Pods.

Στο τέλος θα μπορείς να πεις **ποιος αποκατέστησε τι**. Αυτή η πρόταση είναι πιο χρήσιμη από το «το Kubernetes το ξανασήκωσε».

Πρώτα ο χάρτης

Το εργαστήριο τρέχει μέσα στο Mac σου. Το kind χρησιμοποιεί Docker containers ως nodes. Μέσα σε ένα τέτοιο node θα τρέχουν τα δύο Pods της εφαρμογής. Κάθε Pod του παραδείγματος έχει ένα container με Rails και Puma.



Σχήμα 5.2 · Το κομμάτι του χάρτη που μας ενδιαφέρει. Το ένα node φιλοξενεί και το control plane, που εδώ παραλείπεται για να φανούν τα όρια Pod και container.

Το **κεχριμπαρένιο διακεκομμένο περίγραμμα** δείχνει το επιθυμητό. Το **πετρόλ συνεχές περίγραμμα** δείχνει το παρατηρημένο. Οι ετικέτες και το είδος της γραμμής κρατούν τη διάκριση και σε ασπρόμαυρη εκτύπωση. Τα γκρίζα πλαίσια δείχνουν όρια και δομή.

Στα αντικείμενα που διαθέτουν `spec` και `status`, το πρώτο περιγράφει τι ζητάς και το δεύτερο την τελευταία αναφερόμενη κατάσταση. Το `status` μπορεί να καθυστερεί. Δεν είναι ζωντανή φωτογραφία όλων όσων συμβαίνουν στο cluster.

Στα σχήματα τα Pods λέγονται A, B και Γ για να διαβάζονται εύκολα. Στο terminal θα έχουν πραγματικά ονόματα και UIDs. Το UID είναι η ταυτότητα του συγκεκριμένου αντικειμένου στο API.

Ένα εργαστήριο που μπορείς να επαναλάβεις

Δούλεψε από τον φάκελο `k8s/` του βιβλίου, με Docker Desktop σε λειτουργία. Χρειάζεσαι `docker`, `kind`, `kubectl`, Python 3 και `make`. Οι εκδόσεις και η κατάσταση των ελέγχων καταγράφονται στο τέλος.

TERMINAL

```
make cluster
export KUBECONFIG="$PWD/build/kubeconfig"
kubectl config current-context
kubectl get nodes
kubectl get pods
```

Το `make cluster` δημιουργεί το τοπικό cluster `k8s-book`, χτίζει τη μικρή εφαρμογή Rails, φορτώνει το image στα nodes και περιμένει να ξεκινήσουν δύο Pods. Η πρώτη εκτέλεση κατεβάζει και τα απαραίτητα images. Οι χρόνοι εξαρτώνται από το δίκτυο και το cache.

Το `export` επιλέγει το ξεχωριστό kubeconfig του βιβλίου για αυτό το terminal. Το context πρέπει να είναι **`kind-k8s-book`**. Στο αρχείο αυτό έχει οριστεί και το namespace `k8s-book`, όπου εκτελούνται όλες οι εντολές του κεφαλαίου.

Πριν συνεχίσεις, το node πρέπει να γράφει `Ready` και τα δύο Pods να έχουν `1/1` στη στήλη `READY`. Αν το context διαφέρει, επανάλαβε το `export` από τον σωστό φάκελο.

Το image περιέχει μόνο το web κομμάτι. Δεν χρειάζεται Postgres, Redis ή asset pipeline. Το `/` επιστρέφει το όνομα της εφαρμογής και του Pod που απάντησε. Η εφαρμογή μένει μικρή ώστε το πείραμα να αφορά το cluster.

> **Αν δεν φτάσεις σε δύο έτοιμα Pods**
> Δες `kubectl get pods` και έπειτα `kubectl describe pod ONOMA`. Το `ImagePullBackOff` συνήθως σημαίνει ότι δεν φορτώθηκε το image του βιβλίου στο kind ή ότι άλλαξε το όνομά του. Επανάλαβε `make cluster`. Για container που ξεκινά και πέφτει, δες `kubectl logs ONOMA`.

Για να σταματήσεις το εργαστήριο όταν τελειώσεις, τρέξε `bash scripts/cluster-down.sh`. Η επόμενη εκτέλεση του `make cluster` το δημιουργεί ξανά.

Η δήλωση που μένει

Το παρακάτω είναι ολόκληρο το αρχείο `manifests/ch05/deployment.yaml`. Θα το διαβάσουμε αναλυτικά στο κεφάλαιο 9. Εδώ κράτα τρία σημεία. Το όνομα είναι `rails-map`, το `replicas` ζητά **δύο** αντίγραφα και το `image` είναι αυτό που έχτισε το εργαστήριο.

manifests/ch05/deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: rails-map
  namespace: k8s-book
spec:
  replicas: 2
  selector:
    matchLabels:
      app: rails-map
  template:
    metadata:
      labels:
        app: rails-map
    spec:
      containers:
        - name: web
          image: k8s-book/rails-map:ch05
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 3000
```

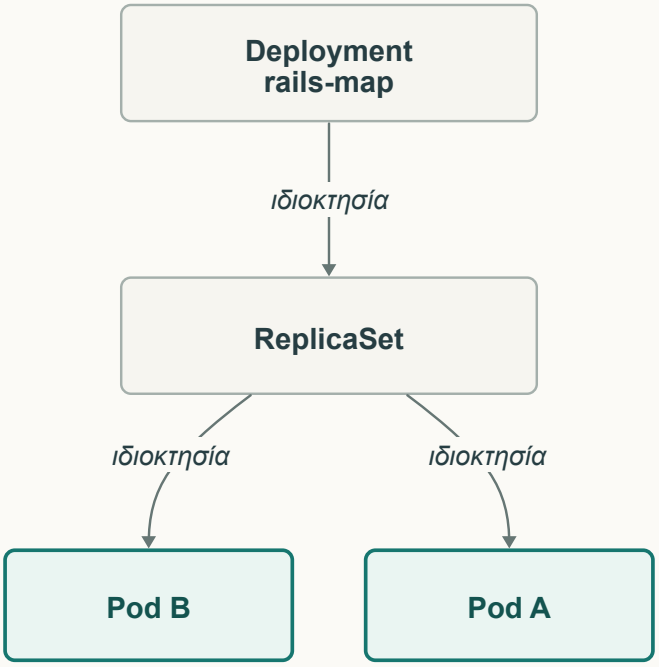
Το `kubectl apply` στέλνει τη δήλωση στο API. Η αποδοχή της δεν σημαίνει ότι τα containers έχουν ήδη ξεκινήσει. Γι’ αυτό περιμένουμε ξεχωριστά το rollout και μετά κοιτάμε τα Pods.

```
TERMINAL

kubectl apply -f manifests/ch05/deployment.yaml
kubectl rollout status deployment/rails-map --timeout=120s
kubectl get deployment rails-map
kubectl get pods -l app=rails-map
```

Το αρχείο στο Mac είναι η πηγή που επεξεργάζεσαι. Οι controllers διαβάζουν την αποθηκευμένη δήλωση από το API. Δεν παρακολουθούν τον φάκελό σου για αλλαγές.

Ο αριθμός έχει υπεύθυνο

Το Deployment δεν εκτελεί Ruby. Είναι αντικείμενο στο API. Ο Deployment controller διαχειρίζεται ReplicaSets. Ο ReplicaSet controller διατηρεί τον ζητούμενο αριθμό των Pods που ανήκουν στο ReplicaSet.



Σχήμα 5.3 · Η αλυσίδα ιδιοκτησίας. Τα συνεχή βέλη διαβάζονται από τον ιδιοκτήτη προς το αντικείμενο που του ανήκει. Δεν είναι κίνηση HTTP.

Στο σταθερό στιγμιότυπο του πειράματος έχουμε ένα Deployment, ένα ReplicaSet και δύο Pods. Το ReplicaSet έχει αντίγραφο του ζητούμενου αριθμού στο δικό του `spec.replicas`.

Διάλεξε ένα από τα Pods και κράτησε την ταυτότητά του. Οι επόμενες εντολές εκτελούνται στο ίδιο terminal, ώστε να διατηρηθεί η μεταβλητή `POD`.

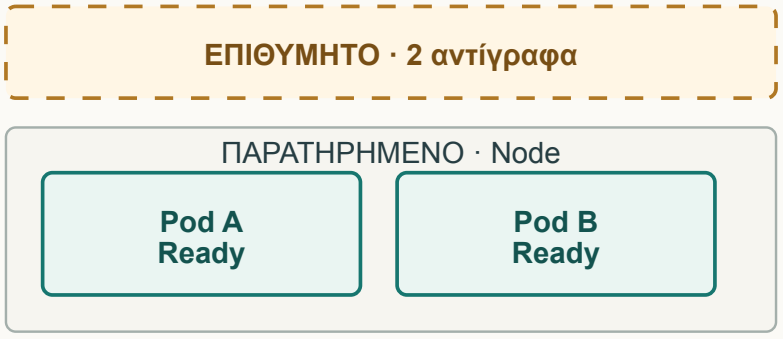
TERMINAL

```
POD=$(kubectl get pods -l app=rails-map \
  -o jsonpath='{.items[0].metadata.name}')
kubectl get pod "$POD" \
  -o custom-columns='NAME:.metadata.name,UID:.metadata.uid'
kubectl get pod "$POD" \
  -o jsonpath='{.metadata.ownerReferences[0].kind}{"\n"}'
```

Η τελευταία εντολή πρέπει να γράψει `ReplicaSet`. Διαβάζει το `ownerReferences`, τη σύνδεση του Pod με τον ιδιοκτήτη του. Τα labels βοηθούν το ReplicaSet να επιλέγει Pods. Το `ownerReferences` δείχνει σε ποιο αντικείμενο ανήκει το συγκεκριμένο Pod.

Σβήσε το Pod

Πριν τη διαγραφή, ο στόχος και το παρατηρημένο πλήθος συμφωνούν. Στο σχήμα το Pod που διάλεξες λέγεται A.



Σχήμα 5.4 · Πριν τη διαγραφή. Το ReplicaSet ζητά δύο αντίγραφα και τα δύο Pods είναι έτοιμα.

TERMINAL

```
kubectl delete pod "$POD" --wait=false
kubectl get pods -l app=rails-map --watch
```

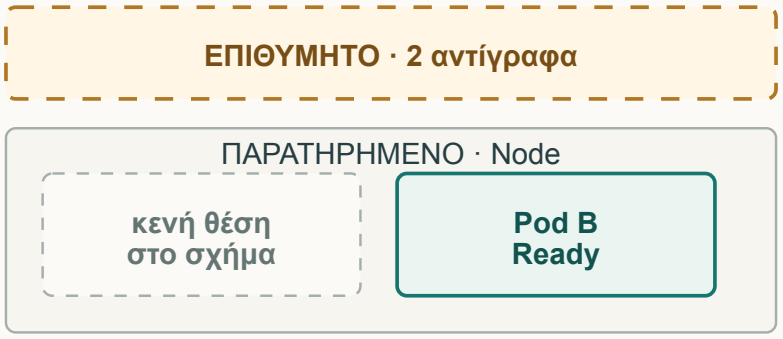
Το `--wait=false` επιστρέφει μετά την αποδοχή της διαγραφής από το API. Δεν περιμένει να ολοκληρωθεί ο τερματισμός. Το `--watch` δείχνει αλλαγές καθώς αναφέρονται. Σταμάτησέ το με **Ctrl+C** όταν δεις δύο έτοιμα Pods.

Μπορεί να προλάβεις το παλιό Pod σε `Terminating` και το νέο σε `Pending` ή `ContainerCreating`. Μπορεί να εμφανιστεί το νέο πριν φύγει η γραμμή του παλιού. Το ReplicaSet δεν μετρά ένα Pod που τερματίζεται ως ενεργό αντίγραφο.

Οι ενδιάμεσες γραμμές και η διάρκειά τους δεν είναι το κριτήριο επιτυχίας. Το watch μπορεί να ξεκινήσει όταν μέρος της αλλαγής έχει ήδη συμβεί. Το κρίσιμο εύρημα είναι ότι τελικά υπάρχουν δύο έτοιμα Pods και ένα από αυτά έχει **νέο UID**.

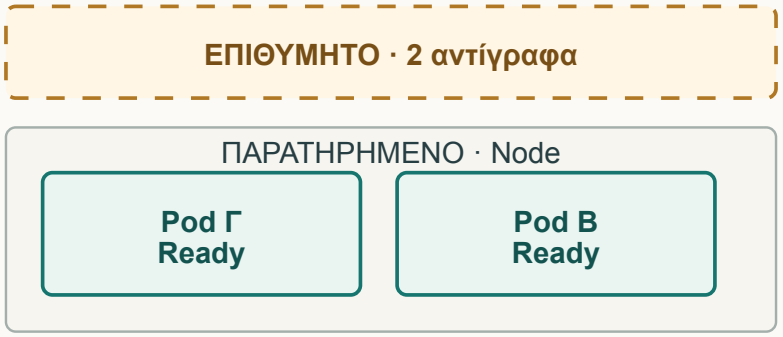
Η διαφορά κλείνει

Δεν άλλαξες το `replicas`. Ο controller εξακολουθεί να έχει στόχο δύο, ακόμα κι όταν το Pod Α έχει πάψει να μετρά ως ενεργό.



Σχήμα 5.5 · Μετά τη διαγραφή του Α. Η κενή θέση είναι βοήθημα του σχήματος, όχι αντικείμενο του Kubernetes. Το παλιό Pod μπορεί ακόμη να τερματίζεται.

Ο ReplicaSet controller δημιουργεί ένα νέο Pod από το πρότυπο που έχει αποθηκευμένο. Ο scheduler το τοποθετεί σε node και το kubelet φροντίζει να ξεκινήσει το container μέσω του runtime.

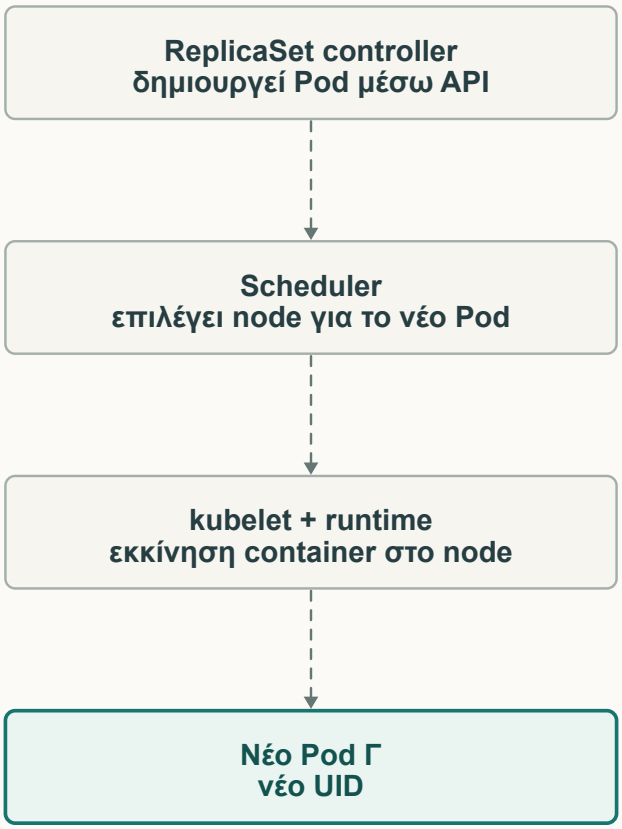


Σχήμα 5.6 · Μετά την αναπλήρωση. Το Γ έχει νέο UID. Το Β έμεινε στη θέση του. Ο αριθμός επέστρεψε στον στόχο.

Η ίδια θέση στο χαρτί βοηθά να δεις την αλλαγή. Δεν υπόσχεται την ίδια IP ούτε ένα μόνιμο «κουτάκι» μέσα στο cluster. Σε cluster με περισσότερα nodes, το νέο Pod μπορεί να τοποθετηθεί αλλού.

Από το νέο αντικείμενο στο container

Η αναπλήρωση χρειάζεται περισσότερους από έναν υπεύθυνους. Το νέο αντικείμενο Pod δεν είναι ακόμα ένα Rails process που δέχεται αιτήματα.



Σχήμα 5.7 · Σειρά ευθυνών κατά την αναπλήρωση. Τα διακεκομμένα γκρίζα βέλη δείχνουν τη διαδοχή ενεργειών. Οι συνιστώσες συντονίζονται μέσω του API, δεν καλούν απευθείας η μία την επόμενη.

Μετά το Ctrl+C, έλεγξε το αποτέλεσμα.

TERMINAL

```
kubectl rollout status deployment/rails-map --timeout=120s
kubectl get pods -l app=rails-map \
  -o custom-columns='NAME:.metadata.name,UID:.metadata.uid'
kubectl get pod "$POD" --ignore-not-found
```

Σύγκρινε τα UIDs με αυτά που είδες πριν. Το UID του διαγραμμένου Pod πρέπει να λείπει. Η τελευταία εντολή δεν πρέπει να εμφανίζει τίποτα όταν ολοκληρωθεί η διαγραφή. Αν το παλιό Pod τερματίζεται ακόμη, περίμενε και επανάλαβέ την.

Εδώ δεν έχουμε readiness probe. Το `Ready` του Pod δεν αποδεικνύει ότι η Rails εφαρμογή μπορεί να εξυπηρετήσει κάθε πραγματικό αίτημα. Αυτό το όριο θα το διορθώσουμε με probes στο κεφάλαιο 9.

Το ίδιο σβήσιμο χωρίς controller

Τώρα δημιούργησε ένα Pod απευθείας. Το container είναι το ίδιο. Αυτό που αλλάζει είναι η ιδιοκτησία.

manifests/ch05/independent-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: independent
  namespace: k8s-book
  labels:
    app: independent
spec:
  containers:
    - name: web
      image: k8s-book/rails-map:ch05
      imagePullPolicy: IfNotPresent
      ports:
        - containerPort: 3000
```

Πρόσεξε το label `app: independent`. Είναι σκόπιμα διαφορετικό από το `app: rails-map`. Αν ένα ανεξάρτητο Pod ταιριάζει στον selector ενός ReplicaSet, το ReplicaSet μπορεί να το υιοθετήσει. Θα είχες αλλάξει το ίδιο το πείραμα.

TERMINAL

```
kubectl apply -f manifests/ch05/independent-pod.yaml
kubectl wait --for=condition=Ready pod/independent --timeout=120s
kubectl get pod independent \
  -o jsonpath='{.metadata.ownerReferences}{"\n"}'
kubectl delete pod independent
kubectl get pods -l app=independent
```

Πριν τη διαγραφή, η εντολή για το `ownerReferences` πρέπει να επιστρέψει κενή γραμμή. Μετά τη διαγραφή, η τελευταία εντολή πρέπει να αναφέρει ότι δεν βρέθηκαν Pods. Επανάλαβέ την λίγο αργότερα. Τα δύο Pods του Deployment συνεχίζουν να υπάρχουν.

Το ανεξάρτητο Pod είχε δική του δήλωση για το container του. Όταν διέγραψες το Pod, διέγραψες και αυτή τη δήλωση. Δεν έμεινε από πάνω του άλλο αντικείμενο που να ζητά την αναπλήρωσή του.

Αυτό που απέδειξαν οι δύο διαγραφές

Σχήμα 5.8 · Μετά τις δύο διαγραφές. Αριστερά υπάρχει ακόμη δήλωση πλήθους στο ReplicaSet. Δεξιά δεν υπάρχει controller που να ζητά νέο independent Pod.

Στην πρώτη περίπτωση έφυγε ένα εξαρτώμενο αντικείμενο. Ο στόχος του ιδιοκτήτη του παρέμεινε. Στη δεύτερη έφυγε το ίδιο το αντικείμενο που περιέγραφε τι πρέπει να τρέξει.

Αυτός είναι ο λόγος που η φράση «το Kubernetes ξαναφτιάχνει ό,τι σβήσω» δεν αρκεί για να προβλέψεις συμπεριφορά. Χρειάζεται να βρεις τον controller, τη δήλωση που ακολουθεί και τα αντικείμενα που διαχειρίζεται.

Ο βρόχος επίσης δεν εγγυάται ότι η αποκατάσταση θα πετύχει. Ένα λανθασμένο image μπορεί να δώσει νέο Pod που δεν ξεκινά. Έλλειψη πόρων μπορεί να αφήσει το Pod σε `Pending`. Ο controller μπορεί να έχει δημιουργήσει το ζητούμενο αντικείμενο και η εφαρμογή να παραμένει μη διαθέσιμη.

> **Το όριο αυτού του πειράματος**
> Διέγραψες ένα Pod σε λειτουργικό node. Δεν έχασες μηχάνημα. Εφόσον εδώ υπάρχει μόνο ένα node, τα δύο αντίγραφα δεν προσφέρουν αντοχή σε απώλειά του. Το πείραμα απώλειας worker, με αναπλήρωση σε άλλα nodes και μέτρηση των καθυστερήσεων, ανήκει στο κεφάλαιο 17.

Σπάσε μόνο το container

Κράτησε το Pod και τερμάτισε μόνο το Puma μέσα του. Στο image του βιβλίου τρέχει ως PID 1. Η εντολή στέλνει `SIGTERM` μέσα στο επιλεγμένο container.

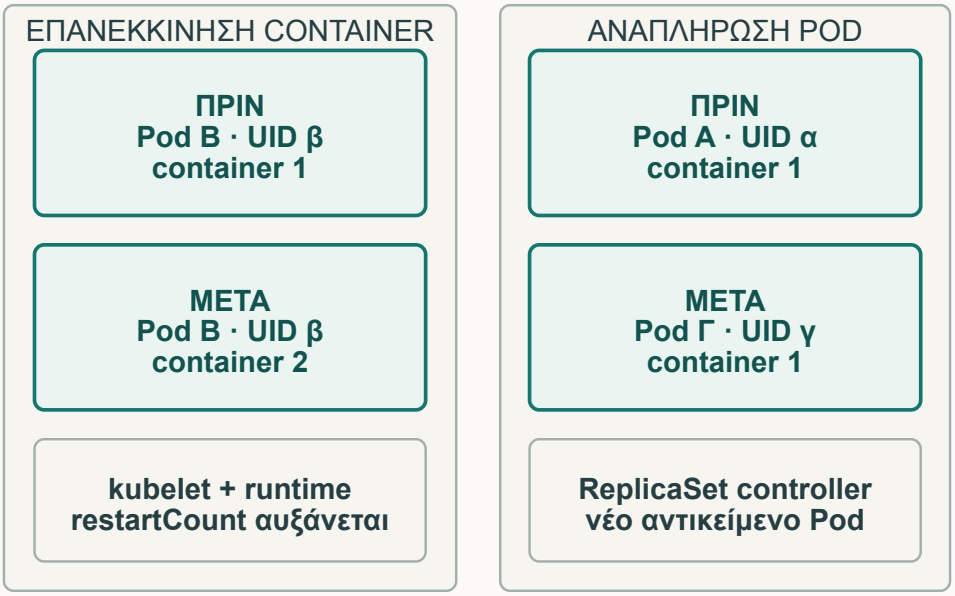
TERMINAL

```
POD=$(kubectl get pods -l app=rails-map \
  -o jsonpath='{.items[0].metadata.name}')
kubectl get pod "$POD" -o jsonpath='{.metadata.uid}{"\n"}'
kubectl exec "$POD" -- ruby -e 'Process.kill("TERM", 1)'
kubectl get pod "$POD" --watch
```

Το exec μπορεί να κλείσει με σφάλμα καθώς τερματίζεται το container. Όταν αυξηθεί το `RESTARTS` και το Pod γίνει πάλι `1/1`, σταμάτησε το watch με Ctrl+C και έλεγξε την ταυτότητα.

TERMINAL

```
kubectl get pod "$POD" -o jsonpath='{.metadata.uid}{"\n"}'
kubectl get pod "$POD" \
  -o jsonpath='{.status.containerStatuses[0].restartCount}{"\n"}'
```



Σχήμα 5.9 · Αριστερά αλλάζει το container μέσα στο ίδιο Pod. Δεξιά δημιουργείται άλλο Pod. Τα ελληνικά γράμματα παριστάνουν διαφορετικά UIDs, όχι πραγματικές τιμές.

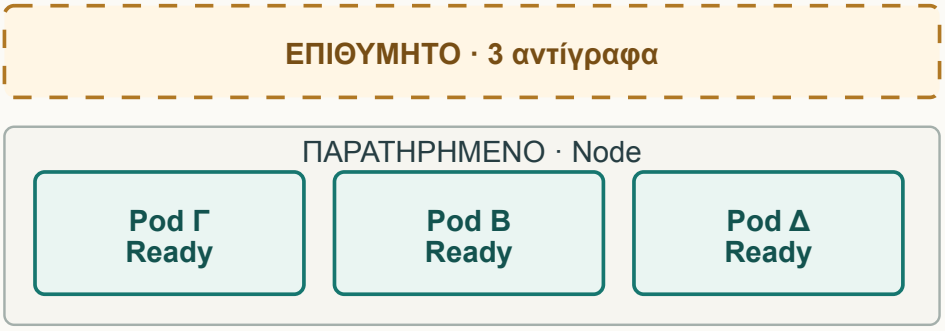
Το UID έμεινε ίδιο, ενώ ο μετρητής επανεκκινήσεων αυξήθηκε. Η προεπιλεγμένη πολιτική `restartPolicy: Always` επιτρέπει στο kubelet να επανεκκινήσει το container. Αυτό θα ίσχυε και μέσα σε ανεξάρτητο Pod που **δεν έχει διαγραφεί**. Δεν χρειάζεται ReplicaSet για αυτή την τοπική επανεκκίνηση.

Άσκηση με τρία αντίγραφα

Άλλαξε προσωρινά τον στόχο σε τρία αντίγραφα και περίμενε να ξεκινήσουν.

TERMINAL

```
kubectl scale deployment rails-map --replicas=3
kubectl rollout status deployment/rails-map --timeout=120s
kubectl get pods -l app=rails-map
```



Σχήμα 5.10 · Ο νέος στόχος είναι τρία. Η αλλαγή αφορά τη δήλωση που ακολουθεί ο controller.

Κάνε τις ακόλουθες προβλέψεις πριν τρέξεις άλλες εντολές.

1. Αν διαγράψεις ένα από τα τρία Pods, σε ποιο πλήθος θα επιστρέψει το Deployment;
2. Αν το τοπικό YAML εξακολουθεί να γράφει δύο, αρκεί αυτό από μόνο του για να μειωθεί το πλήθος;
3. Ποια εντολή επαναφέρει τον στόχο στα δύο, χρησιμοποιώντας το αρχείο του βιβλίου;

Διέγραψε ένα Pod με την ίδια διαδικασία του πρώτου πειράματος. Περίμενε το rollout και μέτρησε τα Pods. Έπειτα γύρισε στα δύο. Η άσκηση τελειώνει όταν η δήλωση και το παρατηρημένο πλήθος συμφωνούν ξανά σε δύο.

Το στοιχείο που χρειάζεσαι είναι το σημείο όπου αποθηκεύτηκε η αλλαγή του `kubectl scale`. Η εντολή άλλαξε το Deployment στο API. Δεν επεξεργάστηκε το αρχείο σου.

Λύση και επαναφορά

Μετά τη διαγραφή, ο στόχος παραμένει τρία. Ο ReplicaSet controller δημιουργεί άλλο Pod και το πλήθος επιστρέφει στα τρία, εφόσον το νέο Pod μπορεί να ξεκινήσει.

Το YAML στο Mac δεν ξαναδιαβάζεται μόνο του. Για να γίνει πάλι ο αποθηκευμένος στόχος δύο, εφάρμοσε την αρχική δήλωση.

TERMINAL

```
kubectl apply -f manifests/ch05/deployment.yaml
kubectl rollout status deployment/rails-map --timeout=120s
kubectl get deployment rails-map
kubectl get pods -l app=rails-map
```

Το `READY` του Deployment πρέπει να γίνει `2/2`. Όταν ολοκληρωθεί και ο τερματισμός του επιπλέον Pod, η λίστα πρέπει να έχει δύο έτοιμα Pods. Ο controller μπορεί να επιλέξει ποιο θα αφαιρέσει. Τα σύντομα ονόματα των εικόνων δεν υπόσχονται ότι θα κρατηθεί ένα συγκεκριμένο Pod.

Τι άλλαξες	Τι παραμένει	Ποιος δρα	Τι ελέγχεις
Διέγραψες managed Pod	Ο στόχος του ReplicaSet	ReplicaSet controller	Νέο UID, ίδιο πλήθος

Τι άλλαξες	Τι παραμένει	Ποιος δρα	Τι ελέγχεις
Διέγραψες independent Pod	Δεν υπάρχει στόχος πλήθους	Κανένας controller αναπλήρωσης	Το Pod λείπει
Τερμάτισες container	Το ίδιο Pod και η restart policy	kubelet και runtime	Ίδιο UID, περισσότερα restarts
Άλλαξες το replicas	Ο νέος στόχος στο API	Deployment και ReplicaSet controllers	Το πλήθος ακολουθεί τον νέο στόχο

> **Η ερώτηση που παίρνεις μαζί σου**
> Όταν κάτι επανέρχεται, ποια δήλωση εξακολουθεί να το ζητά και ποιος controller την ακολουθεί;

Στο επόμενο κεφάλαιο θα ακολουθήσουμε τη δική σου εικόνα από το Docker build μέχρι το container μέσα στο Pod.

Σημειώσεις του εργαστηρίου

Η κατάσταση των ελέγχων βρίσκεται στο `verification/ch05.md`. Το `make verify` επαναλαμβάνει τους ελέγχους στο τοπικό cluster του βιβλίου. Τα ονόματα και τα UIDs αλλάζουν σε κάθε εκτέλεση.

Το πείραμα εκτελέστηκε σε Kubernetes. Στις 7 Σεπτεμβρίου 2026 ελέγχθηκαν η αναπλήρωση Pod με νέο UID, η απουσία αναπλήρωσης του ανεξάρτητου Pod, το restart container με ίδιο UID και η άσκηση των τριών replicas. Στο τέλος επανήλθαν δύο έτοιμα αντίγραφα.

Η εκτέλεση χρησιμοποίησε kind 0.33.0 και Kubernetes 1.34.11 σε arm64. Η αναλυτική καταγραφή βρίσκεται στο `verification/ch05.md`. Οι χρόνοι αφορούν το συγκεκριμένο τοπικό εργαστήριο.

Οι πηγές πίσω από το μοντέλο

Οι Controllers περιγράφουν τον βρόχο παρατήρησης και ενέργειας. Η τεκμηρίωση του ReplicaSet εξηγεί τον στόχο πλήθους, την ιδιοκτησία και την υιοθέτηση Pods που ταιριάζουν στον selector.

Το Pod lifecycle ξεχωρίζει την ταυτότητα του Pod από την επανεκκίνηση των containers του. Η τεκμηρίωση για Deployments καλύπτει τη διαχείριση των ReplicaSets και το rollout.

Το kind quick start εξηγεί το τοπικό cluster και τη φόρτωση images. Οι συγκεκριμένες εκδόσεις και τα digests του εργαστηρίου κρατιούνται στα αρχεία του project, ώστε το παράδειγμα να μπορεί να επαναληφθεί.

Έλεγχος κατανόησης

Πριν προχωρήσεις, εξήγησε τη διαφορά επιθυμητού και παρατηρημένου και βρες ποιος controller διαχειρίζεται κάθε βήμα. Το έτοιμο YAML θα διαβαστεί αναλυτικά στο κεφάλαιο 9. Το πείραμα πρέπει ήδη να βγάζει νόημα από τον στόχο replicas και την ιδιοκτησία των Pods.

ΚΕΦΑΛΑΙΟ 6

Η δική σου εικόνα στα Pods

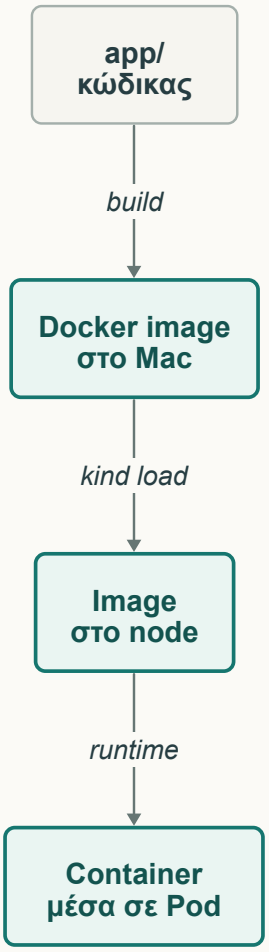
Το Docker build τελειώνει στο Mac. Το container ξεκινά σε ένα node. Ανάμεσα στα δύο υπάρχει μία μεταφορά που πρέπει να γίνει.

Στο προηγούμενο κεφάλαιο το setup είχε ήδη ετοιμάσει την εικόνα. Τώρα παίρνουμε αυτή τη δουλειά στα χέρια μας. Το ζητούμενο είναι απλό. Αλλάζεις μια εφαρμογή, χτίζεις μια συγκεκριμένη έκδοση και μπορείς να εξηγήσεις ποιος κώδικας τρέχει μέσα σε κάθε Pod.

Η εφαρμογή βρίσκεται στο `app/`. Απαντά με JSON που περιέχει το όνομα της εφαρμογής, την έκδοση και το hostname του Pod. Δεν έχει asset pipeline. Στο αρχικό web δεν απαιτεί σύνδεση με Postgres ή Redis. Έτσι μια βλάβη εικόνας δεν μπερδεύεται με μια βλάβη βάσης.

> **Από το Kamal στο Kubernetes**
> Στο Kamal η διαδρομή build και μεταφοράς κρύβει μέρος της μηχανικής. Εδώ ξεχωρίζεις την εικόνα του Docker στο Mac από την εικόνα που διαθέτει το runtime του node. Το ίδιο tag σε δύο αποθήκες δεν αποδεικνύει ίδιο περιεχόμενο.

Τέσσερα διαφορετικά σημεία



Σχήμα 6.1 · Ο πηγαίος κώδικας γίνεται εικόνα στο Docker του Mac. Το kind τη φορτώνει στο runtime κάθε node. Από εκεί ξεκινά το container.

Το Docker Desktop και το container runtime μέσα στο kind node δεν είναι η ίδια αποθήκη εικόνων. Το ότι βλέπεις ένα tag στο `docker images` δεν αποδεικνύει ότι το node μπορεί να το χρησιμοποιήσει. Σε ένα απομακρυσμένο cluster συνήθως θα έκανες push σε registry. Εδώ χρησιμοποιούμε τη σύντομη τοπική διαδρομή.

TERMINAL

```
bash scripts/build-images.sh
docker --context desktop-linux image inspect \
  k8s-book/rails-map:v1 --format '{{.Id}}'
```

Το script χτίζει τις εκδόσεις `v1` και `v2` και τις φορτώνει στο cluster του βιβλίου. Οι ενέργειες για μία έκδοση, απομονωμένες από το script, είναι οι εξής.

```
TERMINAL

docker --context desktop-linux build \
  --build-arg APP_VERSION=v1 \
  -t k8s-book/rails-map:v1 app
kind load docker-image k8s-book/rails-map:v1 \
  --name k8s-book
```

Το `v1` είναι ένα συγκεκριμένο όνομα έκδοσης του εργαστηρίου. Σε κανονικό deploy προτίμησε μοναδικό tag από το commit ή digest. Αν ξαναχτίσεις άλλο περιεχόμενο με το ίδιο tag, το όνομα παύει να αρκεί για την αναγνώριση του κώδικα. Ένα node μπορεί ακόμη να έχει την προηγούμενη εικόνα.

Ποια εικόνα ζητά το Pod

Στο Deployment του προηγούμενου κεφαλαίου ο container λέγεται `web`. Αλλάζουμε την εικόνα του και περιμένουμε να ολοκληρωθεί η αντικατάσταση των Pods.

```
TERMINAL

kubectl set image deployment/rails-map \
  web=k8s-book/rails-map:v1
kubectl rollout status deployment/rails-map --timeout=180s
kubectl get pods -l app=rails-map -o wide
```

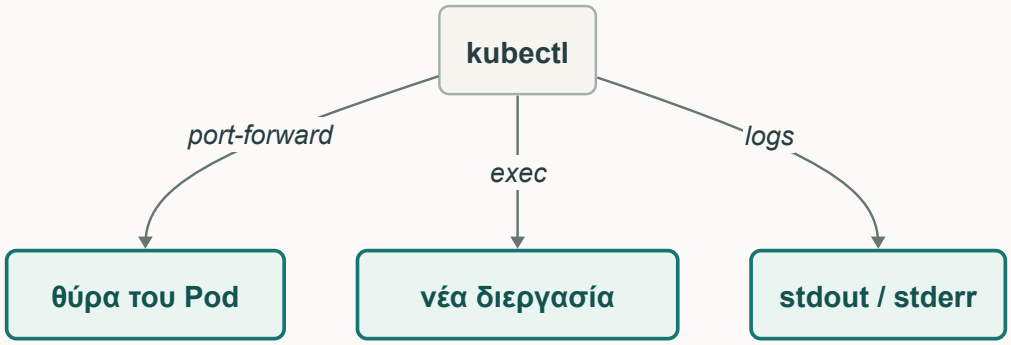
Το αντίστοιχο τμήμα του manifest είναι μικρό.

```
YAML

containers:
  - name: web
    image: k8s-book/rails-map:v1
    imagePullPolicy: IfNotPresent
```

Το `IfNotPresent` επιτρέπει στο kubelet να χρησιμοποιήσει την εικόνα που υπάρχει ήδη στο node. Αν δεν υπάρχει, θα επιχειρήσει pull. Δεν σημαίνει «απαγορεύεται το δίκτυο». Η πολιτική `Never` θα απαιτούσε να υπάρχει τοπικά. Η `Always` θα έκανε έλεγχο στο registry ακόμη και όταν μπορεί να επαναχρησιμοποιήσει ήδη κατεβασμένα layers.

Το tag `latest` έχει διαφορετική προεπιλογή πολιτικής όταν δημιουργείται το αντικείμενο. Δεν το χρησιμοποιούμε εδώ. Η ρητή πολιτική και το συγκεκριμένο tag αφαιρούν μια άσχετη πηγή σύγχυσης από το πείραμα. Οι κανόνες περιγράφονται στην επίσημη σελίδα Container images.

Τρία παράθυρα προς το ίδιο Pod



Σχήμα 6.2 · Τα logs διαβάζουν την έξοδο του container. Το exec ξεκινά άλλη διεργασία μέσα του. Το port forward μεταφέρει bytes σε συγκεκριμένο Pod μέσω του API.

```
TERMINAL

kubectl logs deployment/rails-map -c web --tail=30
kubectl exec deployment/rails-map -c web -- \
  ruby -e 'puts RUBY_VERSION'
kubectl port-forward deployment/rails-map 3000:3000
```

Η τελευταία εντολή παραμένει ανοιχτή. Σε άλλο terminal στέλνεις ένα αίτημα.

```
TERMINAL

curl -fsS http://localhost:3000/version
```

Το αποτέλεσμα έχει `version` ίσο με `v1` και το όνομα ενός Pod. Το port forward προς Deployment επιλέγει ένα Pod. Δεν αποδεικνύει ότι όλα τα replicas απαντούν και δεν είναι μηχανισμός κανονικής δημοσίευσης της εφαρμογής. Αν το συγκεκριμένο Pod αντικατασταθεί, μπορεί να χρειαστεί να ξεκινήσεις ξανά την εντολή. Το Service θα λύσει διαφορετικό πρόβλημα στο κεφάλαιο 10.

Το `exec` δεν είναι SSH στο node. Η νέα διεργασία τρέχει στο περιβάλλον του επιλεγμένου container. Αν η εικόνα δεν περιέχει shell ή `curl`, αυτά τα εργαλεία δεν εμφανίζονται επειδή χρησιμοποιείς Kubernetes. Στην εφαρμογή μας υπάρχει Ruby και την αξιοποιούμε για τους ελέγχους.

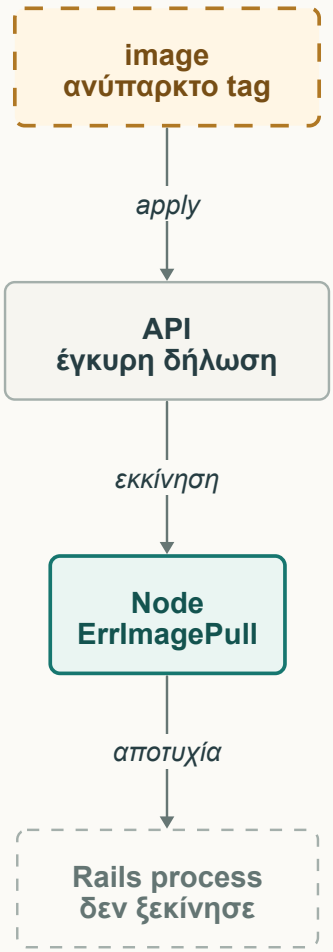
Όταν υπάρχουν πολλά containers στο ίδιο Pod, το `-c web` αποφεύγει την αμφιβολία. Όταν υπάρχουν πολλά Pods, μια εντολή προς Deployment δεν αποτελεί συλλογή από όλα. Για συγκεντρωτικά logs μπορείς να χρησιμοποιήσεις selector και `--prefix=true`.

Σπάσ’ το επίτηδες

Ζήτησε μια εικόνα που δεν υπάρχει. Η βλάβη είναι τοπική στο Deployment του βιβλίου και αντιστρέφεται με την τελευταία εντολή.

TERMINAL

```
kubectl set image deployment/rails-map \
  web=k8s-book/does-not-exist:missing
kubectl get pods -l app=rails-map
kubectl describe pods -l app=rails-map
kubectl get events --sort-by=.metadata.creationTimestamp
```



Σχήμα 6.3 · Η δήλωση έγινε δεκτή από το API. Η εκκίνηση σταμάτησε όταν το node δεν βρήκε την εικόνα. Η εφαρμογή δεν έχει εκτελεστεί ακόμη.

Αρχικά μπορεί να δεις `ErrImagePull` και αργότερα `ImagePullBackOff`. Το δεύτερο περιγράφει την καθυστέρηση ανάμεσα σε επαναλαμβανόμενες προσπάθειες. Η χρήσιμη εξήγηση βρίσκεται στα events. Λάθος όνομα, ανύπαρκτο tag και έλλειψη πρόσβασης σε ιδιωτικό registry μπορεί να καταλήξουν στην ίδια σύντομη ένδειξη.

Μην ξεκινήσεις από τα Rails logs. Δεν υπάρχει Rails process σε αυτόν τον container για να γράψει την αιτία. Έλεγξε το `Image` και το μήνυμα του runtime στο `describe`. Έπειτα επανάφερε την εικόνα και περίμενε το rollout.

TERMINAL

```
kubectl set image deployment/rails-map \
  web=k8s-book/rails-map:v1
kubectl rollout status deployment/rails-map --timeout=180s
```

Η μικρή άσκηση

Άνοιξε το `/version`, κράτησε το όνομα του Pod και βρες το ίδιο αντικείμενο με `kubectl get pod`. Διάβασε το `image` και το `imageID` από το YAML του. Το πρώτο είναι η αναφορά που ζήτησες. Το δεύτερο καταγράφει την εικόνα που χρησιμοποίησε το runtime.

Μετά κλείσε το port forward με Ctrl-C. Η εφαρμογή συνεχίζει να τρέχει. Έκλεισες τη δική σου σήραγγα προς αυτή, όχι το Deployment. Αν αυτή η διάκριση είναι καθαρή, μπορείς πλέον να ακολουθήσεις μια αλλαγή από τον φάκελο `app/` μέχρι ένα συγκεκριμένο container.

Οι λεπτομέρειες των εντολών βρίσκονται στις επίσημες οδηγίες για logs και debugging και port forwarding.

Λύση της άσκησης. Στο YAML του Pod σύγκρινε `spec.containers[].image` με `status.containerStatuses[].imageID` και κράτησε το `metadata.uid`. Το όνομα του tag είναι η ζητούμενη αναφορά. Το runtime αναφέρει την εικόνα που χρησιμοποίησε στο imageID. Το κλείσιμο του port forward δεν αλλάζει αυτά τα πεδία.

ΚΕΦΑΛΑΙΟ 7

YAML και το API

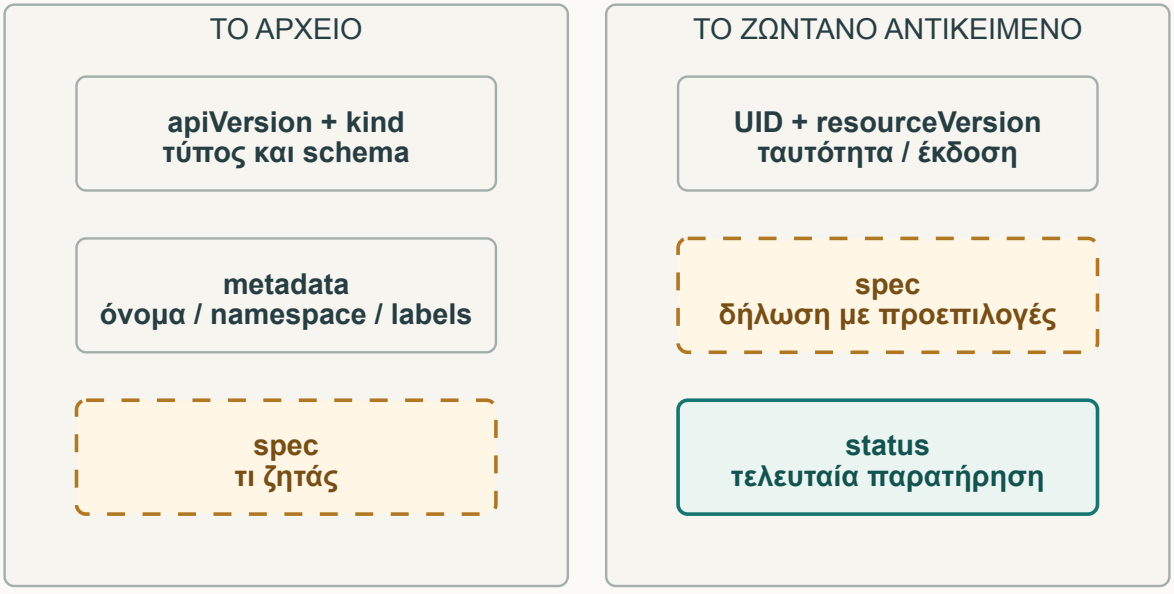
Το YAML είναι ο τρόπος που γράφεις μια δήλωση. Το API είναι αυτό που ορίζει τι σημαίνει.

Μέχρι εδώ χρησιμοποιήσαμε ένα έτοιμο Deployment και μερικές εντολές που το άλλαζαν. Τώρα θα το διαβάσουμε ως αντικείμενο. Δεν χρειάζεται να απομνημονεύσεις εκατοντάδες πεδία. Χρειάζεται να βρίσκεις τον τύπο, την ταυτότητα και το σημείο όπου δηλώνεται η πρόθεση.

Ένα YAML αρχείο μπορεί να είναι συντακτικά σωστό και να περιγράφει αντικείμενο που το cluster δεν γνωρίζει. Μπορεί επίσης να είναι έγκυρο για το API και να δηλώνει κάτι που δεν μπορεί να εκτελεστεί. Η ανύπαρκτη εικόνα του προηγούμενου κεφαλαίου είναι ακριβώς τέτοιο παράδειγμα.

> **Από το Kamal στο Kubernetes**
> Ένα αρχείο ρυθμίσεων deploy περιγράφει την είσοδο ενός εργαλείου. Το Kubernetes object έχει επιπλέον δική του ταυτότητα και πεδία που ενημερώνουν διαφορετικοί writers. Για να εξηγήσεις μια ζωντανή τιμή χρειάζεσαι και τη δήλωση και τη διαχείρισή της.

Διάβασε πρώτα την ταυτότητα



Σχήμα 7.1 · Ο τύπος επιλέγει το schema. Το metadata δίνει ταυτότητα και labels. Το spec δηλώνει επιθυμητή κατάσταση. Το status γράφεται από τα αρμόδια μέρη του cluster.

YAML

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: rails-map
  namespace: k8s-book
spec:
  replicas: 2
```

Αυτό είναι απόσπασμα, όχι πλήρες Deployment. Το `apiVersion` επιλέγει ομάδα και έκδοση του API. Το `kind` επιλέγει τύπο. Το `metadata.name` ονομάζει το αντικείμενο μέσα στο namespace. Το `spec.replicas` ζητά δύο αντίγραφα του Pod template που λείπει από το απόσπασμα.

Το `namespace` δεν είναι μέρος του ονόματος του cluster. Είναι ο χώρος ονομάτων για το συγκεκριμένο αντικείμενο. Δύο Deployments μπορούν να λέγονται `rails-map` σε διαφορετικά namespaces. Ένα Node, αντίθετα, δεν ανήκει σε namespace.

Κοίτα το πλήρες ζωντανό αντικείμενο.

TERMINAL

```
kubectl get deployment rails-map -o yaml
kubectl explain deployment
kubectl explain deployment.spec.replicas
kubectl explain deployment.spec.template.spec.containers
```

Το `explain` διαβάζει το schema που εκθέτει το API του cluster σου. Χρησιμοποίησέ το για τη θέση, τον τύπο και την περιγραφή ενός πεδίου. Για συμπεριφορές όπως το rollout διάβασε και την τεκμηρίωση του controller. Ένα schema δεν εξηγεί μόνο του τη χρονική ακολουθία.

Δεν έχουν όλα spec

Στο ζωντανό Deployment υπάρχουν πεδία που δεν έγραψες. Προεπιλογές, UID, resource version, timestamps και `status` προστίθενται από το σύστημα. Δεν χρειάζεται να τα αντιγράψεις πίσω στο αρχείο που συντηρείς.

Το `status` περιγράφει την τελευταία κατάσταση που αναφέρθηκε. Δεν είναι δεύτερη δήλωση του χρήστη και δεν εγγυάται ότι κανένα γεγονός δεν συνέβη μετά την παρατήρηση. Το `generation` βοηθά να ξεχωρίσεις μια νέα δήλωση από εκείνη που έχει ήδη παρατηρήσει ο controller.

Ένα ConfigMap έχει διαφορετικό σχήμα.

manifests/ch07/field-owner-a.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: field-ownership
  namespace: k8s-book
data:
  message: first
```

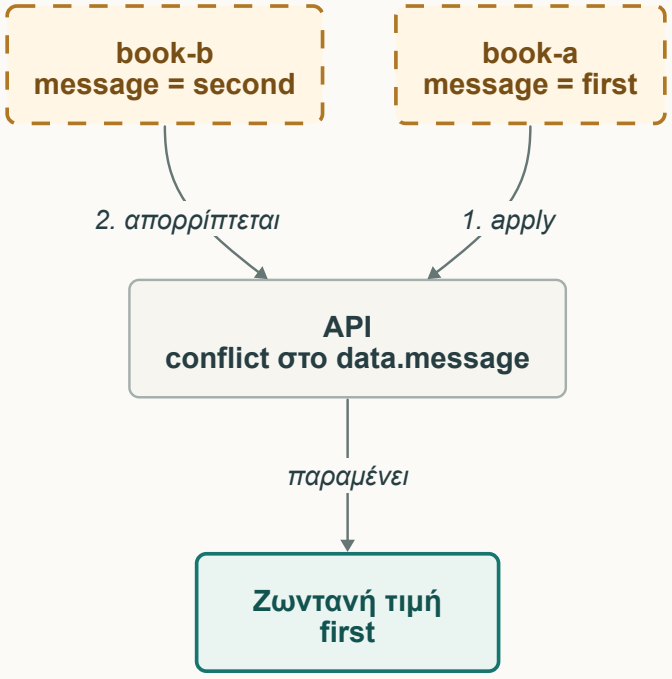
Εδώ τα δεδομένα βρίσκονται στο `data`. Δεν υπάρχει υποχρεωτικό γενικό `spec` που περιέχει τα πάντα. Το ίδιο λάθος εμφανίζεται όταν κάποιος θεωρεί ότι κάθε αντικείμενο έχει Pods. Ένα ConfigMap αποθηκεύει ρυθμίσεις. Δεν εκτελεί διεργασία και δεν χρειάζεται readiness.

Οι βασικές έννοιες ορίζονται στα Kubernetes objects και η συγκεκριμένη δομή στα ConfigMaps.

Ποιος κατέχει το πεδίο

Το `create` επιχειρεί να δημιουργήσει ένα νέο αντικείμενο. Αν υπάρχει ήδη με το ίδιο όνομα, αποτυγχάνει. Το συνηθισμένο client-side `apply` συγκρίνει την τωρινή δήλωση, την προηγούμενη δήλωση που κράτησε σε annotation και το ζωντανό αντικείμενο. Το server-side apply στέλνει τη δήλωση στο API μαζί με την ταυτότητα ενός field manager.

Ο field manager δεν είναι ιδιοκτήτης ολόκληρου του αντικειμένου. Η ιδιοκτησία καταγράφεται ανά πεδίο ή υποδομή πεδίων. Ένα εργαλείο μπορεί να διαχειρίζεται το Pod template και ένας HPA τον αριθμό replicas. Όταν δύο managers προσπαθούν να αλλάξουν ασύμβατα το ίδιο πεδίο με server-side apply, το API μπορεί να απορρίψει την αλλαγή ως conflict.



Σχήμα 7.2 · Δύο writers ζητούν διαφορετική τιμή για το ίδιο data.message. Το API σταματά τη δεύτερη αλλαγή και εμφανίζει τον manager που ήδη διαχειρίζεται το πεδίο.

TERMINAL

```
kubectl apply --server-side --field-manager=book-a \
  -f manifests/ch07/field-owner-a.yaml
kubectl apply --server-side --field-manager=book-b \
  -f manifests/ch07/field-owner-b.yaml
```

Η δεύτερη εντολή αναμένεται να αποτύχει. Το δεύτερο αρχείο αλλάζει το `message` από `first` σε `second`. Διάβασε την αναφορά του conflict και επιβεβαίωσε ότι η τιμή παρέμεινε ίδια.

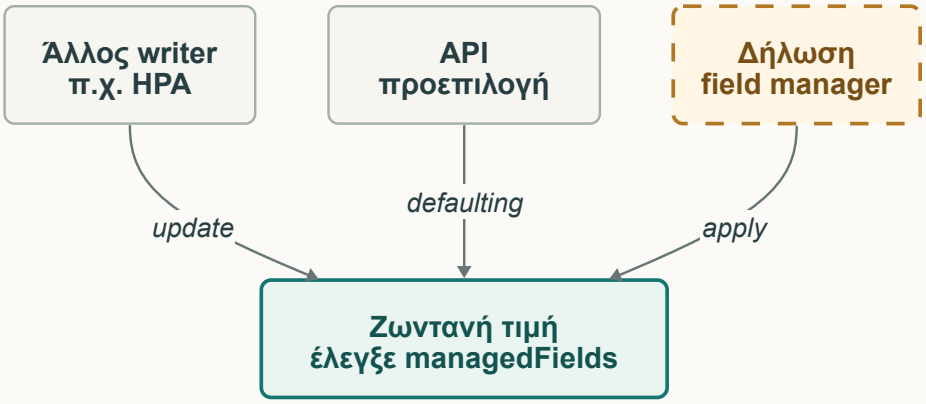
TERMINAL

```
kubectl get configmap field-ownership \
  -o jsonpath='{.data.message}{"\n"}'
kubectl get configmap field-ownership \
  -o yaml --show-managed-fields
```

Το `--force-conflicts` υπάρχει, αλλά σημαίνει συνειδητή ανάληψη διαχείρισης των συγκρουόμενων πεδίων. Δεν είναι κανονικό επόμενο βήμα για να εξαφανιστεί ένα ενοχλητικό μήνυμα. Πρώτα αποφασίζεις ποιος πρέπει να γράφει την τιμή. Η επίσημη περιγραφή του Server-Side Apply εξηγεί και την κοινή ιδιοκτησία όταν οι managers συμφωνούν στην τιμή.

Γιατί επανήλθε αυτό που έσβησα

Η αφαίρεση μιας γραμμής από το αρχείο δεν είναι γενική εντολή «κάνε αυτό το πεδίο ανύπαρκτο». Εξαρτάται από τον τρόπο διαχείρισης, την προηγούμενη ιδιοκτησία και τις προεπιλογές του API. Αν άλλος manager διαχειρίζεται το πεδίο, η δική σου παράλειψη μπορεί να μην το αφαιρέσει. Αν υπάρχει προεπιλογή, η απουσία μπορεί να οδηγήσει σε αυτή. Αν ένας controller το υπολογίζει, μπορεί να το ξαναγράψει.



Σχήμα 7.3 · Το αρχείο είναι μία είσοδος. Οι προεπιλογές του API και οι εξουσιοδοτημένοι writers μπορούν επίσης να επηρεάζουν τη ζωντανή τιμή.

Γι' αυτό ένα debugging session χρειάζεται και το αρχείο και το ζωντανό αντικείμενο. Το `kubectl diff -f ...` δείχνει την αλλαγή που προκύπτει από τη συγκεκριμένη δήλωση. Δεν προβλέπει κάθε μελλοντική ενέργεια ενός controller. Επίσης επιστρέφει διαφορετικό exit code όταν υπάρχει diff, οπότε ένα script δεν πρέπει να μεταφράζει κάθε μη μηδενικό αποτέλεσμα ως χαλασμένο cluster.

Πριν εφαρμόσεις νέο αρχείο, μπορείς να ζητήσεις επικύρωση από τον server χωρίς αποθήκευση.

```
TERMINAL

kubectl apply --dry-run=server -f manifests/ch05/deployment.yaml
```

Ο επιτυχής έλεγχος αποδεικνύει ότι η δήλωση έγινε δεκτή από το API και τους ελέγχους εισαγωγής. Δεν χτίζει την εικόνα, δεν δεσμεύει RAM και δεν εκτελεί το Rails boot.

Το πρώτο σου cluster

Εδώ κλείνει ο πρώτος κύκλος. Χωρίς να κοιτάξεις τις προηγούμενες σελίδες, φόρτωσε την εικόνα, εφάρμοσε το Deployment και άνοιξε την εφαρμογή με port forward. Άλλαξε προσωρινά το όνομα εικόνας, βρες την αιτία στα events και διόρθωσέ την.

Έπειτα εξήγησε τι θα γίνει αν διαγράψεις ένα Pod του ReplicaSet και τι θα γίνει αν διαγράψεις το ανεξάρτητο Pod του κεφαλαίου 5. Στην πρώτη απάντηση πρέπει να υπάρχει controller, επιθυμητός αριθμός και νέο UID. Στη δεύτερη πρέπει να εξηγήσεις ποια δήλωση λείπει. Η φράση «το Kubernetes τα ξανασηκώνει» δεν αρκεί πλέον.

Αν μπορείς να κάνεις αυτά τα βήματα και να βρεις το σωστό context πριν τα εκτελέσεις, έχεις ένα χρήσιμο πρώτο cluster. Τα επόμενα κεφάλαια θα συνδέσουν όσα τώρα τρέχουν με όσα πρέπει να δέχονται κίνηση, να κρατούν δεδομένα και να ολοκληρώνουν δουλειές.

ΚΕΦΑΛΑΙΟ 8

Labels και selectors

Δύο αντικείμενα δεν συνδέονται επειδή μοιάζουν τα ονόματά τους. Συνδέονται με συγκεκριμένα πεδία.

Έχεις δύο web Pods και έναν ανεξάρτητο client. Θέλεις να βλέπεις μόνο τα web. Αργότερα θέλεις ένα Service να στέλνει κίνηση στα web και ένας κανόνας διασποράς να τα μετρά ως σύνολο. Η κοινή γλώσσα για αυτές τις επιλογές είναι τα labels.

Ένα label είναι ζεύγος κλειδιού και τιμής στο metadata. Δεν αλλάζει από μόνο του τον κώδικα ή τον ρόλο ενός container. Το `tier=web` είναι περιγραφή που αποκτά συνέπειες όταν κάποια αναζήτηση ή κάποιος controller τη χρησιμοποιήσει.

> **Από το Kamal στο Kubernetes**
> Ένας ρόλος web σε βοηθά να ομαδοποιείς υπηρεσίες. Τα labels επιτρέπουν περισσότερες ταυτόχρονες ομάδες πάνω στα ίδια Pods. Ο selector περιγράφει ποια βρίσκεις, όχι ποια κατέχεις.

Δες το σύνολο πριν το αλλάξεις

TERMINAL

```
kubectl get pods --show-labels
kubectl get pods -l app=rails-map
kubectl get pods -l app=rail-map
```

Η τελευταία εντολή είναι συντακτικά σωστή. Το API δεν ξέρει ότι ξέχασες ένα `s`. Επιστρέφει το σύνολο που αντιστοιχεί στον selector, το οποίο εδώ είναι κενό. Η απουσία αποτελεσμάτων δεν σημαίνει ότι τα Pods εξαφανίστηκαν.

Σχήμα 8.1 · Τα ίδια Pods, δύο ερωτήματα. Ο σωστός selector βρίσκει δύο. Το μικρό τυπογραφικό λάθος βρίσκει μηδέν.

Το επόμενο βήμα είναι να συγκρίνεις τον selector με τις πραγματικές ετικέτες. Μην αλλάξεις τυχαία τα Pods για να ταιριάξουν με το λάθος ερώτημα. Βρες ποια δήλωση είναι η πηγή της αλήθειας για το συγκεκριμένο σύνολο.

Οι selectors υποστηρίζουν και περισσότερες συνθήκες. Σε μια λίστα χωρισμένη με κόμμα οι συνθήκες πρέπει να ισχύουν μαζί.

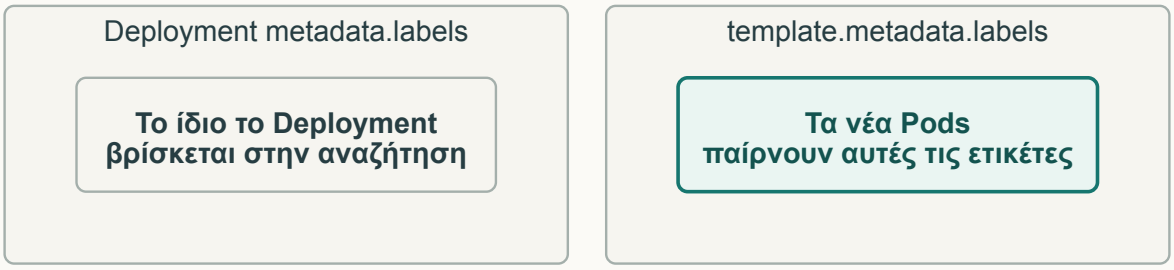
TERMINAL

```
kubectl get pods -l 'app=rails-map,tier=web'
kubectl get pods -l 'app in (rails-map,book-client)'
kubectl get pods -l 'tier'
```

Η πρώτη εντολή ταιριάζει όταν εφαρμόσουμε το manifest του κεφαλαίου 9, που προσθέτει το `tier=web`. Η τρίτη ζητά παρουσία του κλειδιού, ανεξάρτητα από την τιμή του. Τα εισαγωγικά προστατεύουν τους ειδικούς χαρακτήρες από το shell. Οι μορφές και οι περιορισμοί ορίζονται στα Labels and Selectors.

Το label στο σωστό επίπεδο

Ένα Deployment έχει δικό του metadata. Το Pod template μέσα του έχει επίσης metadata. Το πρώτο χαρακτηρίζει το Deployment. Το δεύτερο δίνει labels στα Pods που θα δημιουργηθούν. Η προσθήκη label μόνο στο εξωτερικό metadata δεν το αντιγράφει αυτόματα στα Pods.



Σχήμα 8.2 · Η θέση μέσα στο YAML αλλάζει τον στόχο της ετικέτας. Το Deployment και τα Pods του είναι διαφορετικά αντικείμενα.

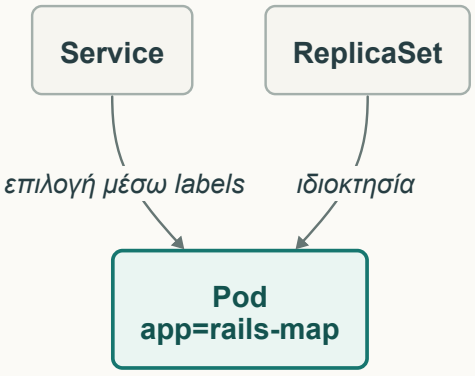
YAML

```
metadata:
  labels:
    component: application
spec:
  selector:
    matchLabels:
      app: rails-map
  template:
    metadata:
      labels:
        app: rails-map
        tier: web
```

Αυτό είναι απόσπασμα για να φανεί η θέση. Ο selector του Deployment πρέπει να ταιριάζει στα labels του template. Στο `apps/v1` δεν μπορείς να αλλάξεις ελεύθερα αυτόν τον selector σε ένα υπάρχον Deployment. Σχεδίασέ τον γύρω από σταθερή ταυτότητα της εφαρμογής, όχι γύρω από την έκδοση που αλλάζει σε κάθε deploy.

Η έκδοση μπορεί να υπάρχει ως πρόσθετο label για αναζήτηση. Αν όμως ο σταθερός selector απαιτεί `version=v1`, ένα template με μόνο `version=v2` δεν περιγράφει πια το ίδιο επιλεγμένο σύνολο. Αυτό δεν είναι ο μηχανισμός ενός συνηθισμένου rolling update.

Επιλογή δεν σημαίνει ιδιοκτησία



Σχήμα 8.3 · Το ReplicaSet διαχειρίζεται τον κύκλο ζωής των Pods. Το Service τα επιλέγει για δρομολόγηση. Το δεύτερο βέλος δεν είναι ownerReference.

Στο κεφάλαιο 5 κοιτάξαμε το `ownerReferences`. Αυτό είναι το πεδίο που κατέγραφε τη σχέση Pod και ReplicaSet. Ένα Service μπορεί να επιλέγει τα ίδια Pods χωρίς να τα κατέχει. Η διαγραφή του Service δεν αποτελεί εντολή διαγραφής των Pods και η δημιουργία του δεν δημιουργεί replicas.

Ένας ReplicaSet έχει επίσης selector. Γι' αυτό το ανεξάρτητο Pod του πειράματος είχε διαφορετικά labels. Ένα ορφανό Pod που ταιριάζει σε selector ReplicaSet μπορεί να υιοθετηθεί από αυτόν. Η απουσία ownerReference τη στιγμή που γράφεις ένα manifest δεν αρκεί αν το βάλεις μέσα στο σύνολο που διαχειρίζεται ήδη ένας controller.

Αυτή η διάκριση βοηθά και στη διάγνωση. Για την ερώτηση «ποιος θα αναπληρώσει το Pod» κοιτάς ιδιοκτησία και controllers. Για την ερώτηση «γιατί το Service δεν το βλέπει» συγκρίνεις selector, namespace και labels. Δεν ψάχνεις μια σχέση ιδιοκτησίας που δεν υπάρχει.

Η μικρή άσκηση

Γράψε πρώτα σε χαρτί πόσα Pods περιμένεις από κάθε selector. Έπειτα εκτέλεσέ τον. Αν το αποτέλεσμα είναι κενό, γύρισε στο `--show-labels` και εξήγησε ακριβώς ποια συνθήκη δεν ισχύει.

Πρόσθεσε προσωρινά ένα αθώο label στο Deployment, χωρίς να πειράξεις το template.

TERMINAL

```
kubectl label deployment rails-map book-note=exercise
kubectl get deployment -l book-note=exercise
kubectl get pods -l book-note=exercise
kubectl label deployment rails-map book-note-
```

Το Deployment εμφανίζεται, τα Pods όχι. Το τελικό - αφαιρεί το συγκεκριμένο label. Ο αριθμός replicas παραμένει ίδιος. Το πείραμα κλείνει όταν μπορείς να εξηγήσεις τη διαφορά από τη θέση του metadata, χωρίς να επικαλεστείς κάποια καθυστέρηση του cluster.

ΚΕΦΑΛΑΙΟ 9

Deployment και rolling update

Το deploy είναι μια περίοδος όπου η παλιά και η νέα έκδοση συνυπάρχουν. Η στρατηγική ορίζει πόσο χώρο και πόση διαθεσιμότητα μπορείς να χρησιμοποιήσεις στη μετάβαση.

Στο κεφάλαιο 5 μάθαμε γιατί επανέρχονται τα Pods. Τώρα θα δούμε πώς αλλάζουν. Το Deployment διαχειρίζεται ReplicaSets με διαφορετικά Pod templates. Μια αλλαγή εικόνας ή env στο template μπορεί να δημιουργήσει νέα αναθεώρηση. Μια αλλαγή μόνο στο πλήθος replicas δεν είναι νέο template εφαρμογής.

Η εφαρμογή μας διαθέτει `v1` και `v2`. Χτίσε τις εικόνες με το script του κεφαλαίου 6 αν δεν υπάρχουν και εφάρμοσε την αρχική κατάσταση.

TERMINAL

```
kubectl apply -f manifests/ch09/web.yaml
kubectl rollout status deployment/rails-map --timeout=180s
```

> **Από το Kamal στο Kubernetes**
> Το deploy αλλάζει έκδοση και το rollback επιστρέφει κώδικα. Εδώ το Deployment αποθηκεύει Pod templates και ελέγχει τη σταδιακή αντικατάσταση. Η συμβατότητα της βάσης παραμένει χωριστό μέρος της διαδικασίας.

Δύο έτοιμα και μία επιπλέον θέση

Το manifest κρατά δύο replicas και προσθέτει μια συγκεκριμένη στρατηγική.

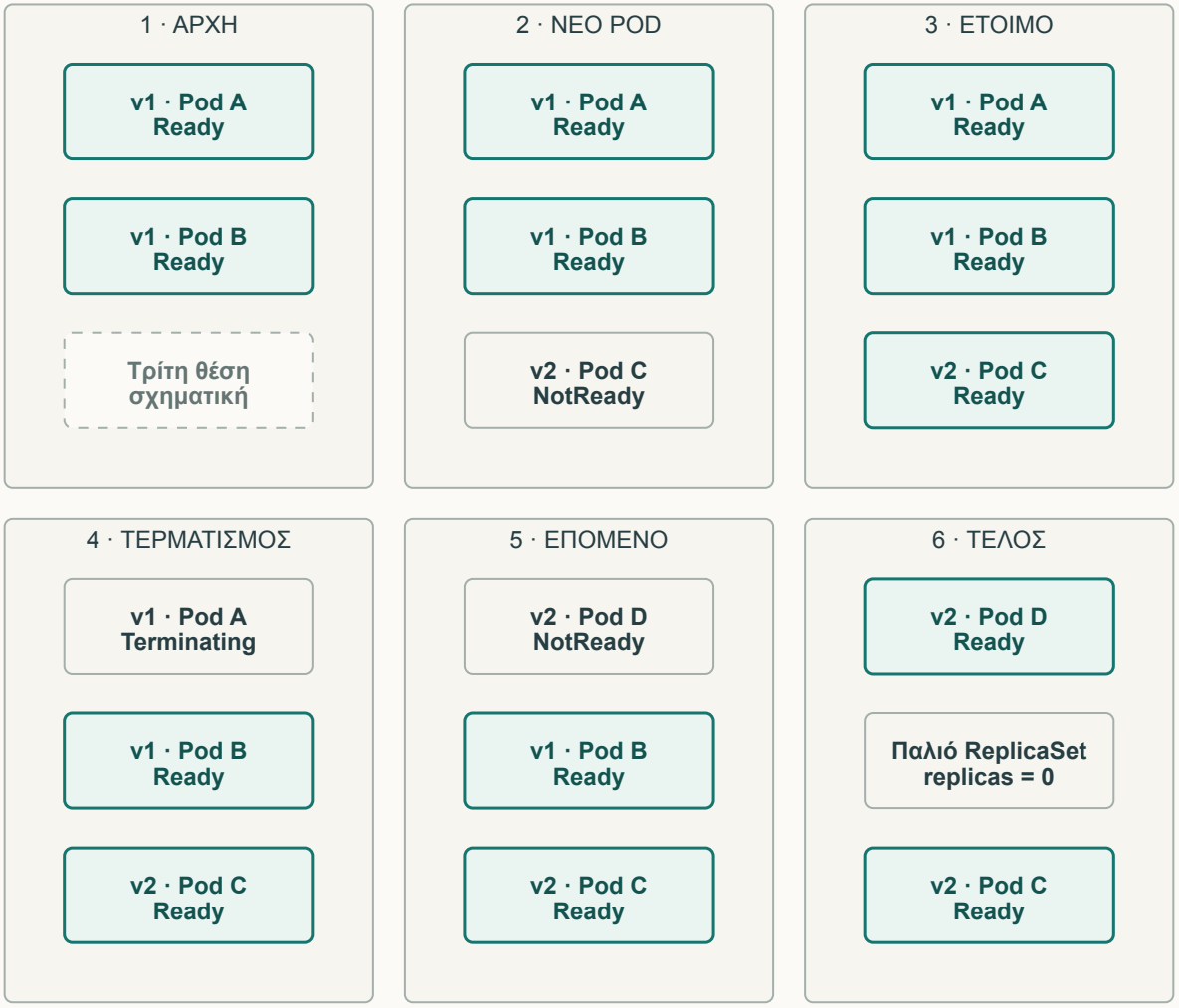
YAML

```
replicas: 2
progressDeadlineSeconds: 90
strategy:
  type: RollingUpdate
  rollingUpdate:
    maxSurge: 1
    maxUnavailable: 0
```

Το `maxSurge` επιτρέπει ένα επιπλέον replica πάνω από τον στόχο κατά τη μετάβαση. Το `maxUnavailable` δεν επιτρέπει στο Deployment να μειώσει τα διαθέσιμα replicas κάτω από τον επιθυμητό αριθμό όσο πραγματοποιεί το rollout. Δεν προστατεύει από κάθε εξωτερική βλάβη. Αν

χαθεί node ή χαλάσουν και τα δύο παλιά Pods, η ρύθμιση δεν δημιουργεί διαθεσιμότητα από το μηδέν.

Στα επόμενα καρέ κρατάμε σταθερές θέσεις για να διαβάζεται η αλλαγή. Οι θέσεις είναι σχηματικές. Δεν είναι υποδοχές του API και δεν αντιστοιχούν υποχρεωτικά σε διαφορετικά nodes. Τα Pods που τερματίζονται μπορεί προσωρινά να κάνουν τον συνολικό αριθμό ορατών αντικειμένων μεγαλύτερο από τον απλό στόχο μαζί με το surge.



Σχήμα 9.1 · Ένα επιτρεπτό rolling update σε έξι καρέ. Η νέα έκδοση γίνεται έτοιμη πριν αρχίσει να αποσύρεται η παλιά. Η πραγματική σειρά μικρών γεγονότων μπορεί να διαφέρει.

Running και Ready απαντούν αλλού

Το `Running` είναι φάση του Pod. Δεν αποδεικνύει ότι το Rails έχει ολοκληρώσει το boot ή ότι μπορεί να εξυπηρετήσει το συγκεκριμένο αίτημα. Στο manifest προσθέσαμε readiness probe.

```yaml
readinessProbe:
  httpGet:
    path: /ready
    port: http
  periodSeconds: 2
  failureThreshold: 1
```

Το kubelet καλεί το endpoint. Η εφαρμογή απαντά επιτυχώς όταν επιτρέπει να λάβει κίνηση. Μια αποτυχία readiness δεν είναι εντολή restart. Αλλάζει την ετοιμότητα του Pod και επηρεάζει τη συνήθη επιλογή endpoints από το Service που θα προσθέσουμε αμέσως μετά.

Στην αρχική έκδοση το `/ready` ελέγχει έναν απλό διακόπτη του εργαστηρίου. Όταν ενεργοποιήσουμε τη βάση θα ελέγχει και τη σύνδεση. Η επιλογή εξαρτήσεων σε readiness χρειάζεται κρίση. Αν μια κοινή εξάρτηση αποτύχει και όλα τα web βγουν από το Service, η εφαρμογή μπορεί να χάσει και endpoints που θα μπορούσαν ακόμη να απαντήσουν. Το probe πρέπει να εκφράζει συγκεκριμένη δυνατότητα εξυπηρέτησης.

```terminal
kubectl set image deployment/rails-map \
  web=k8s-book/rails-map:v2
kubectl get pods -l app=rails-map -w
```

Σε δεύτερο terminal παρακολούθησε τον controller.

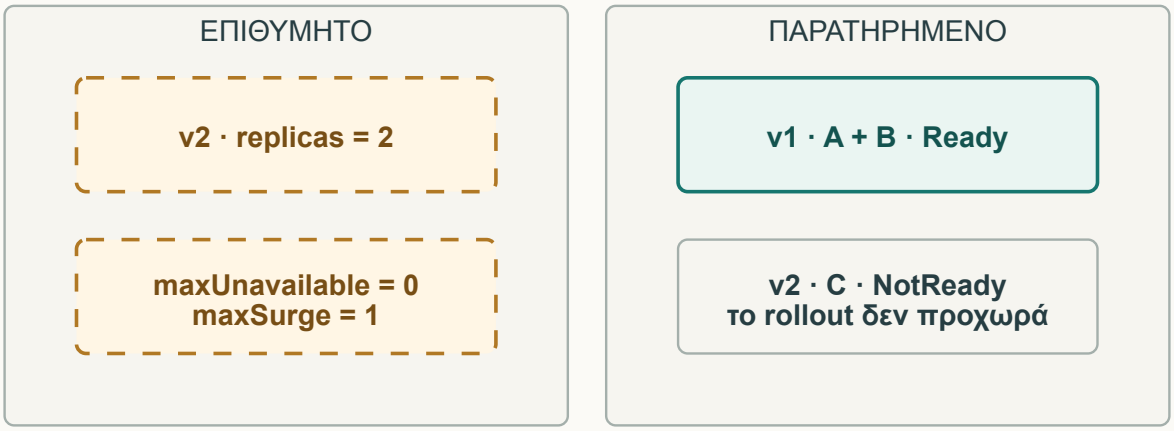
```terminal
kubectl rollout status deployment/rails-map --timeout=180s
kubectl get replicasets -l app=rails-map
kubectl rollout history deployment/rails-map
```

Το παλιό ReplicaSet μπορεί να μείνει με μηδέν replicas για το ιστορικό. Δεν σημαίνει ότι η παλιά εφαρμογή εξακολουθεί να δέχεται κίνηση. Το μοντέλο των αναθεωρήσεων και οι προϋποθέσεις προόδου περιγράφονται στα Deployments.

Σπάσ’ το επίτηδες

Ζήτησε νέα αναθεώρηση που δεν γίνεται έτοιμη.

```terminal
kubectl set env deployment/rails-map READINESS_FAIL=1
kubectl rollout status deployment/rails-map --timeout=120s
kubectl get pods -l app=rails-map
kubectl describe deployment rails-map
```



Σχήμα 9.2 · Το νέο Pod υπάρχει και τρέχει, αλλά δεν είναι έτοιμο. Με maxUnavailable μηδέν, τα παλιά έτοιμα replicas παραμένουν.

Το `rollout status` δεν πρέπει να αναφέρει επιτυχία. Το timeout της εντολής και το `progressDeadlineSeconds` του Deployment είναι διαφορετικά ρολόγια. Το δεύτερο επιτρέπει στον controller να αναφέρει ότι δεν υπήρξε η απαιτούμενη πρόοδος. Δεν εκτελεί αυτόματα rollback.

Κοίτα το νέο Pod και τα probe events. Θα δεις αποτυχία HTTP στο `/ready`, ενώ μπορεί να υπάρχει κανονικό Rails process. Επανέφερε το προηγούμενο template.

TERMINAL

```
kubectl rollout undo deployment/rails-map
kubectl rollout status deployment/rails-map --timeout=180s
kubectl apply -f manifests/ch09/web.yaml
kubectl rollout status deployment/rails-map --timeout=180s
```

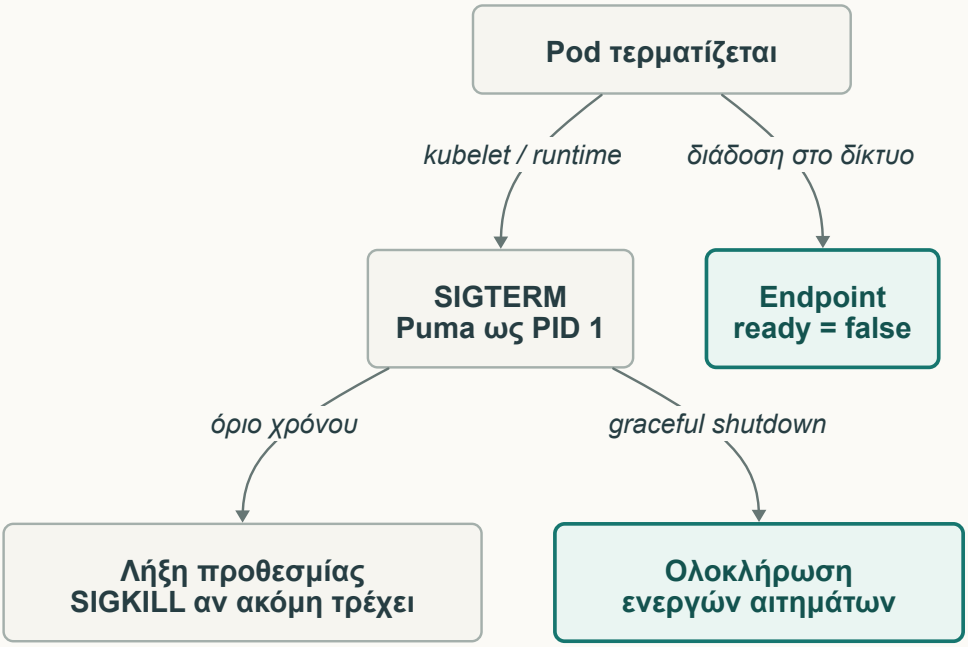
Η πρώτη επαναφορά γυρίζει στην αμέσως προηγούμενη αναθεώρηση. Το τελευταίο apply επαναφέρει τη δηλωμένη αρχή του κεφαλαίου, με εικόνα `v1`, ώστε η επόμενη άσκηση να ξεκινά από γνωστό σημείο.

Πώς φεύγει ένα Pod

Το να ξεκινήσει το νέο web δεν αρκεί. Το παλιό πρέπει να ολοκληρώσει τα αιτήματα που έχει ήδη αναλάβει. Η εικόνα χρησιμοποιεί exec μορφή για την εντολή εκκίνησης του Puma. Έτσι το σήμα τερματισμού φτάνει στη διεργασία που πρέπει να το χειριστεί.

YAML

```
terminationGracePeriodSeconds: 30
```



Σχήμα 9.3 · Η διάδοση της αλλαγής στα endpoints και η παράδοση SIGTERM είναι παράλληλες εργασίες. Το διάγραμμα δεν υπόσχεται ατομική σειρά μεταξύ τους.

Το runtime στέλνει SIGTERM στο πλαίσιο του graceful shutdown. Ο server χρειάζεται χρόνο για τις ενεργές συνδέσεις. Αν παραμένει σε λειτουργία όταν εξαντληθεί η προθεσμία, ακολουθεί αναγκαστικός τερματισμός. Ένα `preStop` hook καταναλώνει μέρος της ίδιας προθεσμίας. Ένα αυθαίρετο sleep δεν αποτελεί απόδειξη ότι το δίκτυο έχει ενημερωθεί παντού.

Οι αλλαγές στα EndpointSlices δεν έχουν μηδενική καθυστέρηση. Γι' αυτό η καλή στρατηγική deploy συνδυάζει ετοιμότητα, επαρκή χωρητικότητα, σωστό χειρισμό σημάτων και μέτρηση πραγματικών αιτημάτων. Το επόμενο κεφάλαιο κάνει αυτή τη μέτρηση μέσα από Service. Η ακολουθία τερματισμού τεκμηριώνεται στο Pod lifecycle.

Τι γυρίζει πίσω

Σχήμα 9.4 · Το undo αφορά το Pod template του Deployment. Ρυθμίσεις και δεδομένα χρειάζονται δικό τους σχέδιο επαναφοράς.

Το `rollout undo` δεν επαναφέρει την τιμή ενός ConfigMap που άλλαξες χωριστά. Δεν αναιρεί ένα migration στη βάση. Δεν ακυρώνει ένα email που έστειλε ο worker. Επαναφέρει προηγούμενο Pod template και προκαλεί άλλο rollout.

Η άσκηση είναι να προβλέψεις τη συνύπαρξη των εκδόσεων. Αν η `v2` γράφει δεδομένα που η `v1` δεν καταλαβαίνει, το πρόβλημα υπάρχει πριν ακόμη χρειαστεί rollback. Υπάρχει ήδη στα καρέ όπου οι δύο εκδόσεις εξυπηρετούν μαζί. Θα επιστρέψουμε σε αυτή τη χρονική επικάλυψη όταν συντονίσουμε migrations και workers.

ΚΕΦΑΛΑΙΟ 10

Service και εσωτερικό DNS

Τα Pods αλλάζουν. Ο client χρειάζεται ένα όνομα που να παραμένει.

Αν κρατήσεις την IP ενός Pod σε μια ρύθμιση, το επόμενο rollout μπορεί να την καταστήσει άχρηστη. Το νέο Pod έχει νέα ταυτότητα και μπορεί να έχει άλλη διεύθυνση. Δεν θέλεις κάθε client να παρακολουθεί μόνος του τον κύκλο ζωής των web replicas.

Ένα συνηθισμένο Service τύπου ClusterIP προσφέρει σταθερή εσωτερική διεύθυνση και όνομα. Ο selector του περιγράφει ποια Pods ανήκουν στο σύνολο των πιθανών προορισμών. Τα EndpointSlices κρατούν τις διευθύνσεις και τις συνθήκες που χρησιμοποιούν οι μηχανισμοί δρομολόγησης.

> **Από το Kamal στο Kubernetes**
> Το σταθερό όνομα μιας υπηρεσίας παραμένει χρήσιμο όταν αλλάζουν οι διεργασίες. Το Service το χωρίζει από τις Pod IPs. Δεν είναι ο controller που δημιουργεί τα αντίγραφα και δεν δίνει από μόνο του δημόσια HTTP είσοδο.

Το πρώτο Service

Ξεκίνα από το έτοιμο web του προηγούμενου κεφαλαίου.

TERMINAL

```
kubectl apply -f manifests/ch09/web.yaml
kubectl rollout status deployment/rails-map --timeout=180s
kubectl apply -f manifests/ch10/service.yaml
kubectl apply -f manifests/ch10/client.yaml
kubectl wait pod/book-client --for=condition=Ready --timeout=120s
```

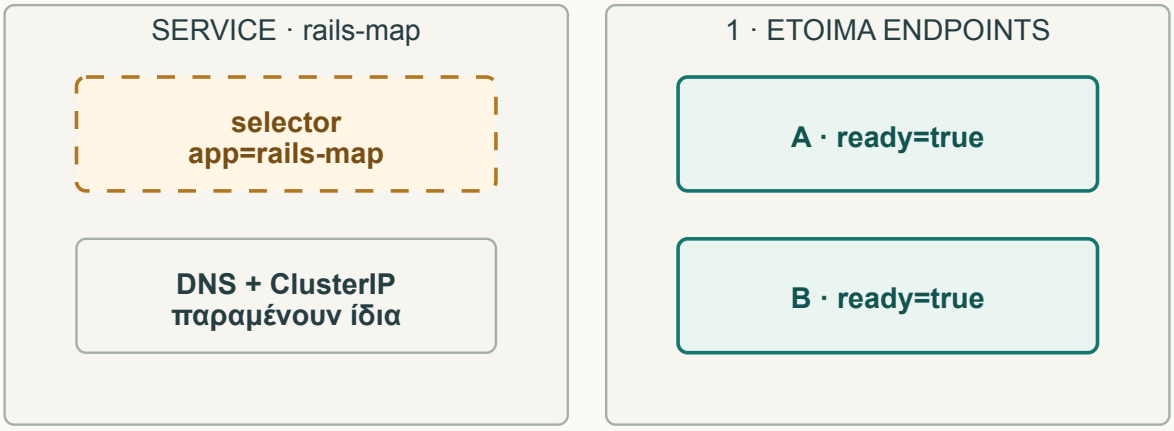
manifests/ch10/service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: rails-map
  namespace: k8s-book
spec:
  selector:
    app: rails-map
  ports:
    - name: http
      port: 80
      targetPort: http
  type: ClusterIP
```

Το `port` είναι η θύρα του Service. Το `targetPort` είναι η θύρα στον επιλεγμένο Pod. Εδώ το όνομα `http` αντιστοιχεί στη θύρα 3000 που δηλώνει ο container. Η δήλωση `containerPort` δεν ξεκινά listener. Ο Puma πρέπει πράγματι να ακούει εκεί.

Το Service δεν είναι ένα επιπλέον container που δημιουργήθηκε μέσα στο namespace. Η προώθηση υλοποιείται από τα στοιχεία δικτύωσης του cluster. Στο εργαστήριο θα δεις kube-proxy, αλλά το API επιτρέπει και άλλες υλοποιήσεις. Μη μετατρέψεις τον εννοιολογικό κόμβο «Service» στο διάγραμμα σε φανταστική διεργασία.

Τρία καρέ που μοιάζουν στο terminal

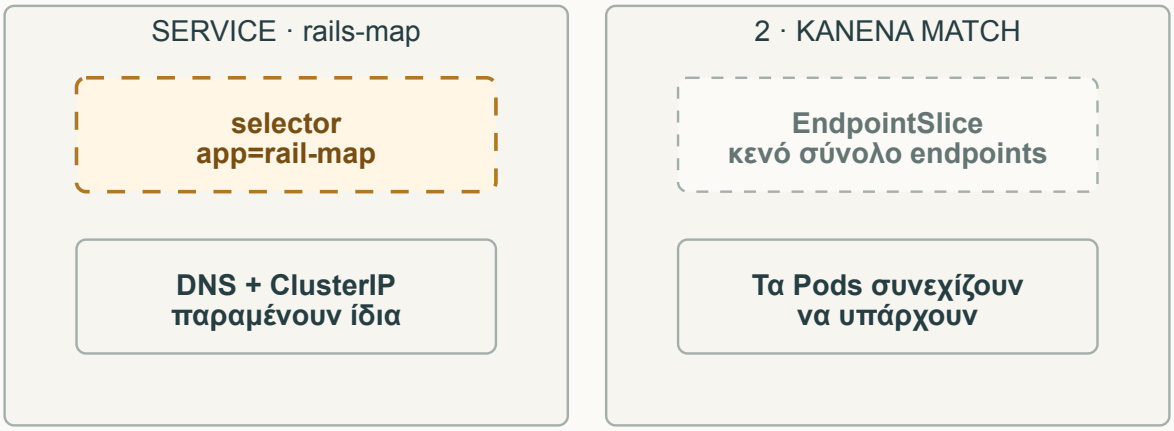


Σχήμα 10.1 · Πρώτη κατάσταση. Ο selector βρίσκει δύο Pods και τα endpoints τους είναι έτοιμα. Το Service έχει χρήσιμους προορισμούς.

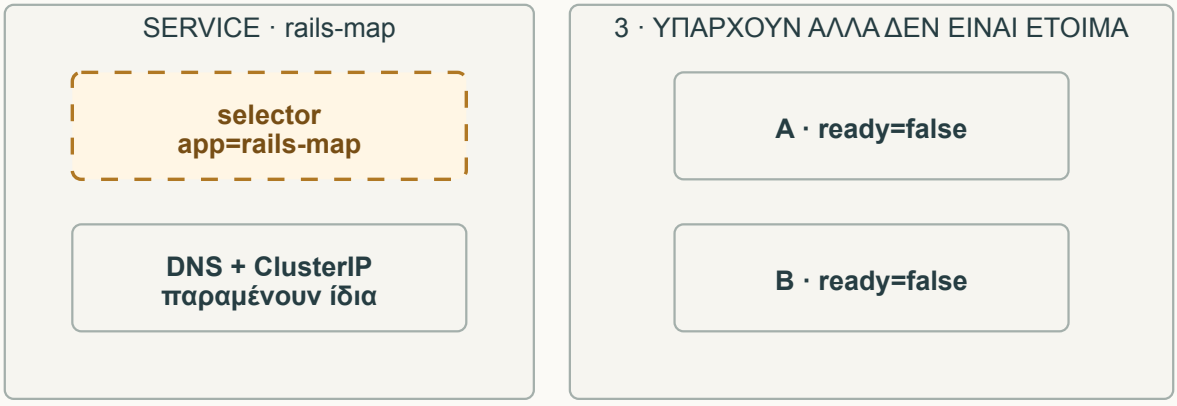
TERMINAL

```
kubectl get service rails-map
kubectl get endpointslices \
  -l kubernetes.io/service-name=rails-map -o yaml
```

Στο YAML κοίτα κάθε `addresses`, το `targetRef` και το `conditions.ready`. Ένα απλό `get service` δεν εμφανίζει όλη αυτή την πληροφορία. Μπορεί να δείχνει απολύτως φυσιολογική ClusterIP ενώ η εφαρμογή είναι μη προσβάσιμη.



Σχήμα 10.2 · Δεύτερη κατάσταση. Ο selector δεν ταιριάζει με κανένα Pod. Η διεύθυνση του Service υπάρχει, αλλά το σύνολο endpoints είναι κενό.



Σχήμα 10.3 · Τρίτη κατάσταση. Ο selector βρίσκει Pods. Τα EndpointSlices περιέχουν τις διευθύνσεις τους, όμως το readiness είναι false.

Στις δύο τελευταίες εικόνες ένας client μπορεί να βλέπει την ίδια γενική αποτυχία σύνδεσης. Η αιτία και η διόρθωση είναι διαφορετικές. Στη δεύτερη συγκρίνεις labels. Στην τρίτη εξετάζεις την εφαρμογή και το probe. Η αναζήτηση συνεχίζεται στο επίπεδο όπου βρέθηκε η πρώτη διαφορά από την αναμενόμενη κατάσταση.

Στο βιβλίο χρησιμοποιούμε Service με selector, χωρίς `publishNotReadyAddresses`. Τα EndpointSlices έχουν επίσης `serving` και `terminating`, που επιτρέπουν λεπτομερέστερο χειρισμό τερματιζόμενων endpoints. Η εξίσωση «κάθε διεύθυνση μέσα σε slice δέχεται νέα κίνηση» είναι λάθος. Η επίσημη σελίδα EndpointSlices περιγράφει τις συνθήκες.

Το όνομα λύνει μέσα στο cluster

Ο `book-client` είναι ένα ανεξάρτητο Pod με Ruby που περιμένει. Δεν έχει το label των web, οπότε δεν γίνεται κατά λάθος backend. Από μέσα του καλούμε το Service.

TERMINAL

```
kubectl exec book-client -- ruby -rnet/http -e \
  'puts Net::HTTP.get(URI("http://rails-map/version"))'
```

Στο ίδιο namespace το σύντομο `rails-map` αρκεί. Από άλλο namespace χρησιμοποιείς `rails-map.k8s-book` ή το πλήρες `rails-map.k8s-book.svc.cluster.local` στο cluster του εργαστηρίου. Η κατάληξη του cluster domain μπορεί να διαφέρει σε άλλα περιβάλλοντα.

Το Mac δεν χρησιμοποιεί αυτόματα το εσωτερικό DNS του cluster. Το να γράψεις `curl http://rails-map` στο δικό σου terminal δεν είναι ισοδύναμο με την προηγούμενη εντολή. Για προσωρινή τοπική πρόσβαση υπάρχει port forward. Για HTTP με host και TLS θα βάλουμε Gateway στο επόμενο κεφάλαιο.

Τύπος Service	Τι προσθέτει	Τι χρειάζεται γύρω του
ClusterIP	Εσωτερική διεύθυνση	Πρόσβαση από το δίκτυο του cluster
NodePort	Θύρα πάνω στα nodes	Προσβάσιμη διαδρομή προς τις IP των nodes
LoadBalancer	Αίτημα για εξωτερικό load balancer	Υλοποίηση που θα τον παρέχει

Στο kind οι IP των node containers βρίσκονται μέσα στο Docker. Ένα NodePort δεν αποτελεί αυτόματα θύρα του Mac. Ένα LoadBalancer που μένει με διεύθυνση `Pending` δεν σημαίνει ότι χάλασε το Rails. Μπορεί απλώς να λείπει η υλοποίηση που εκχωρεί εξωτερική διεύθυνση. Δες τα Services και το DNS για Services.

Rollout με αιτήματα

Κράτησε το Service και άλλαξε τα Pods του. Σε πρώτο terminal ξεκίνησε τον client από το πραγματικό αρχείο Ruby του εργαστηρίου.

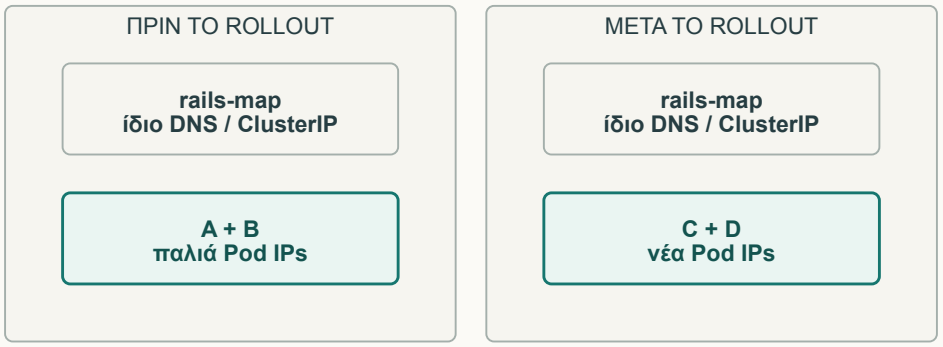
TERMINAL

```
kubectl exec -i book-client -- \
  env BOOK_REQUESTS=300 BOOK_INTERVAL=0.2 ruby \
  < scripts/service-client.rb \
  > build/service-rollout.jsonl
```

Σε δεύτερο terminal άλλαξε έκδοση.

TERMINAL

```
kubectl set image deployment/rails-map \
  web=k8s-book/rails-map:v2
kubectl rollout status deployment/rails-map --timeout=180s
```



Σχήμα 10.4 · Η διεύθυνση και το DNS μένουν σταθερά. Το σύνολο των Pod IPs αλλάζει μέσω των EndpointSlices.

Ο client ανοίγει νέα σύνδεση σε κάθε αίτημα και καταγράφει HTTP code, σώμα ή σφάλμα. Μην περιμένεις αυστηρή εναλλαγή A, B, A, B. Η κατανομή συνδέσεων δεν είναι υπόσχεση κυκλικής σειράς και μια μακρόβια σύνδεση δεν ανακατανέμεται σαν να ήταν νέο αίτημα.

Μέτρησε επιτυχίες και αποτυχίες. Αν δεν εμφανιστεί καμία αποτυχία, αυτό είναι αποτέλεσμα της συγκεκριμένης εκτέλεσης. Δεν είναι απόδειξη ότι κάθε μελλοντικό rollout με διαφορετικό φορτίο, διάρκεια αιτήματος ή topology θα έχει το ίδιο αποτέλεσμα.

Σπάσ’ το και στις δύο πλευρές

Η πρώτη βλάβη αλλάζει μόνο τον selector του Service.

TERMINAL

```
kubectl patch service rails-map --type=merge \
  -p '{"spec":{"selector":{"app":"rail-map"}}}'
kubectl get pods -l app=rails-map
kubectl get endpointslices \
  -l kubernetes.io/service-name=rails-map -o yaml
kubectl apply -f manifests/ch10/service.yaml
```

Η δεύτερη κρατά τα ίδια Pods και τους ίδιους selectors, αλλά κλείνει τον διακόπτη readiness μέσα σε κάθε web. Το αρχείο είναι ειδικό εργαλείο του παραδείγματος.

TERMINAL

```
for book_pod in $(kubectl get pods -l app=rails-map -o name); do
  kubectl exec "$book_pod" -- ruby -e \
    'File.write("/tmp/book-readiness", "0")'
done
kubectl get endpointslices \
  -l kubernetes.io/service-name=rails-map -o yaml
```

Περίμενε τις νέες συνθήκες. Για επαναφορά εκτέλεσε τον ίδιο βρόχο με τιμή `1`. Τα ίδια UIDs γίνονται ξανά ready, χωρίς διαγραφή Pod. Ο έλεγχος ολοκληρώνεται όταν υπάρχουν δύο ready endpoints και το ίδιο αίτημα από τον client πετυχαίνει.

ΚΕΦΑΛΑΙΟ 11

HTTP, TLS και πρόσβαση από έξω

Το Service βρίσκει εφαρμογές μέσα στο cluster. Το Gateway περιγράφει πώς φτάνει ένα HTTP αίτημα από έξω σε αυτές.

Μέχρι εδώ ο client ζούσε στο ίδιο namespace με τα web. Τώρα το αίτημα ξεκινά από το Mac και έχει hostname. Θέλουμε το `rails.book.test` να φτάνει στην εφαρμογή, πρώτα με HTTP και μετά με TLS. Το εσωτερικό Service παραμένει ίδιο.

Η εγκατάσταση γίνεται από το εργαστήριο. Δεν χρειάζεται να διαβάσεις πρώτα εκατοντάδες γραμμές CRDs και δικαιωμάτων του controller για να δεις την πρώτη απάντηση.

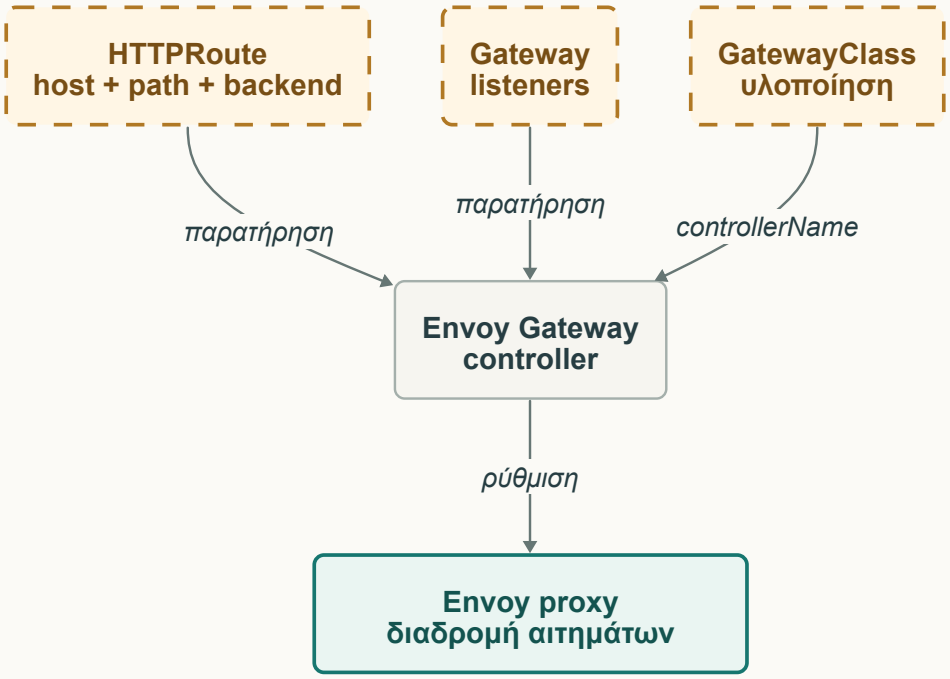
TERMINAL

```
bash scripts/cluster-up.sh --with-gateway
```

Η έκδοση αναφοράς χρησιμοποιεί Envoy Gateway 1.9.1 και τα Gateway API CRDs που περιλαμβάνει το επίσημο chart. Η συμβατότητα με Kubernetes 1.34 ελέγχθηκε στον πίνακα εκδόσεων του Envoy Gateway. Το chart και η έκδοσή του βρίσκονται στο `vendor/envoy/`, ώστε η επανάληψη να μη σημαίνει σιωπηλή εγκατάσταση του σημερινού latest.

> **Από το Kamal στο Kubernetes**
> Τον host και το TLS που έδινες στον proxy θα τα συναντήσεις σε αντικείμενα Gateway API. Οι δηλώσεις χρειάζονται controller και υλοποίηση proxy. Η ύπαρξη του YAML δεν αποδεικνύει ότι περνά ήδη κίνηση.

Τρεις δηλώσεις και ένας controller



Σχήμα 11.1 · Το GatewayClass επιλέγει υλοποίηση. Το Gateway δηλώνει listeners. Το HTTPRoute συνδέει host και path με Service. Ο controller μεταφράζει αυτές τις δηλώσεις σε ρύθμιση του proxy.

Το **GatewayClass** είναι αντικείμενο επιπέδου cluster. Στο παράδειγμα το `controllerName` δείχνει τον Envoy Gateway controller. Το **Gateway** ανήκει στο namespace της εφαρμογής και δηλώνει τους HTTP και HTTPS listeners. Το **HTTPRoute** επιλέγει τον γονικό Gateway και περιγράφει ποια αιτήματα πηγαίνουν σε ποιο backend.

Τα CRDs προσθέτουν τύπους στο API. Ο controller προσθέτει τη συμπεριφορά που τους παρακολουθεί. Αν εγκαταστήσεις μόνο τα CRDs, μπορεί να αποθηκεύεις έγκυρα Gateways χωρίς κανείς να δημιουργεί proxy που τα υλοποιεί.

Ο Envoy Gateway controller και ο Envoy proxy είναι διαφορετικές διεργασίες. Ο πρώτος παρατηρεί δηλώσεις και ρυθμίζει υποδομή. Ο δεύτερος βρίσκεται στη διαδρομή των HTTP αιτημάτων. Δεν στέλνουμε τα κανονικά αιτήματα εφαρμογής στον Kubernetes API server.

Το πρώτο αίτημα από το Mac

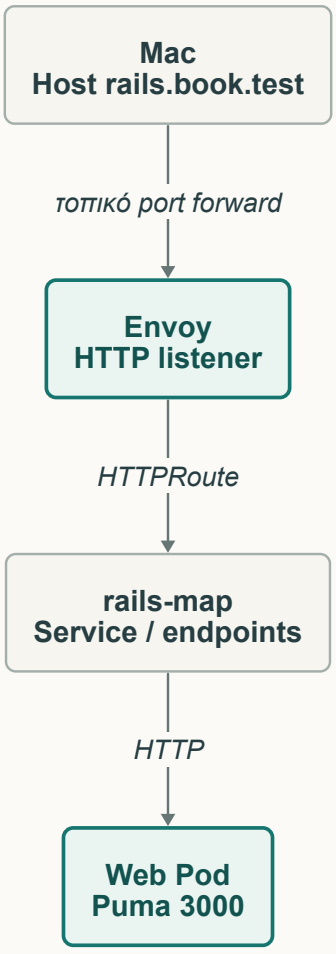
Σε δεύτερο terminal άνοιξε την τοπική πρόσβαση προς το Service του proxy.

TERMINAL

```
bash scripts/gateway-forward.sh
```

TERMINAL

```
curl -fsS -H 'Host: rails.book.test' \
  http://localhost:8088/version
```



Σχήμα 11.2 · Η ειδική διαδρομή του εργαστηρίου. Το port forward εκθέτει τον Envoy στο Mac. Από τον proxy και μετά το αίτημα ακολουθεί τη δρομολόγηση προς την εφαρμογή.

Το setup ζητά ClusterIP για το Service του Envoy και χρησιμοποιεί port forward για το Mac. Έτσι δεν εξαρτάται από εξωτερικό load balancer ή αλλαγές στο DNS του υπολογιστή. Το `Host` του HTTP αιτήματος είναι αυτό που συγκρίνεται με το route. Το `localhost` εδώ είναι μόνο το σημείο σύνδεσης της τοπικής σήραγγας.

Σε κανονικό περιβάλλον θα υπήρχε προσβάσιμη διεύθυνση για τον proxy και DNS που δείχνει σε αυτή. Η τοπική σήραγγα είναι συγκεκριμένη ευκολία του εργαστηρίου. Δεν αποτελεί αρχιτεκτονική δημοσίευσης μιας υπηρεσίας στο διαδίκτυο.

Host, path και backend

manifests/ch11/route.yaml

```
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: rails-map
  namespace: k8s-book
spec:
  parentRefs:
    - name: book
  hostnames:
    - rails.book.test
  rules:
    - matches:
        - path:
            type: PathPrefix
            value: /
      backendRefs:
        - name: rails-map
          port: 80
```

Το `parentRefs` συνδέει το route με το `book`. Το `hostnames` περιορίζει τα ονόματα στα οποία εφαρμόζεται. Το `PathPrefix` με `/` καλύπτει τις διαδρομές της εφαρμογής. Το backend είναι το Service και η θύρα 80 του Service, όχι η θύρα 3000 του Puma.

Αν ήθελες διαφορετικό Service για `/api`, θα πρόσθετες αντίστοιχο κανόνα με αυτό το prefix. Η αντιστοίχιση path δεν σημαίνει αυτόματα αφαίρεση του prefix πριν το backend. Μια εφαρμογή που περιμένει `/tasks` δεν γίνεται συμβατή με `/api/tasks` μόνο επειδή άλλαξες το route. Η επανεγγραφή URL είναι ξεχωριστή δυνατότητα που πρέπει να δηλωθεί και να υποστηρίζεται.

Τα namespaces των routes και των backends έχουν κανόνες αναφοράς. Το παράδειγμα τα κρατά στο ίδιο namespace. Μια αναφορά σε άλλο namespace μπορεί να απαιτεί `ReferenceGrant` από την πλευρά του στόχου. Η ύπαρξη ενός ονόματος δεν είναι γενική εξουσιοδότηση διασύνδεσης. Δες τον οδηγό HTTP routing του Gateway API.

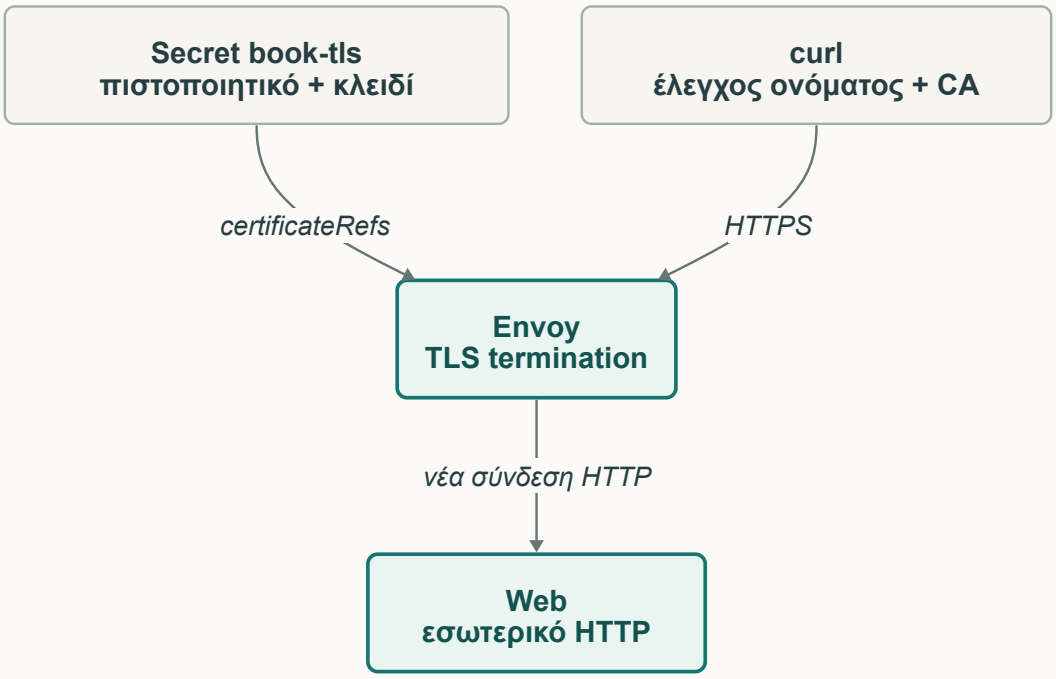
Τοπικό TLS χωρίς αλλαγή εμπιστοσύνης στο Mac

Το setup δημιουργεί προσωρινό πιστοποιητικό για το `rails.book.test` και τοποθετεί το πιστοποιητικό μαζί με το κλειδί σε Secret τύπου TLS. Ο HTTPS listener το αναφέρει στο `certificateRefs`. Ο Envoy τερματίζει το TLS και προωθεί προς το εσωτερικό Service με HTTP.

TERMINAL

```
curl -fsS --cacert build/tls/book.crt \
  --resolve rails.book.test:8443:127.0.0.1 \
  https://rails.book.test:8443/version
```

Το `--resolve` συνδέει αυτό το hostname με localhost μόνο για τη συγκεκριμένη κλήση. Το `--cacert` επιτρέπει στον client να επαληθεύσει το δοκιμαστικό πιστοποιητικό. Δεν προσθέτουμε μόνιμη αρχή εμπιστοσύνης στο macOS και δεν χρειάζεται να παρακάμψουμε τον έλεγχο με `-k`.



Σχήμα 11.3 · Το TLS ολοκληρώνεται στον proxy. Το Secret παρέχει το πιστοποιητικό στον listener. Η σύνδεση από Envoy προς Service είναι χωριστή σύνδεση.

Το πιστοποιητικό του εργαστηρίου έχει περιορισμένη διάρκεια. Όταν λήξει, χρειάζεται ανανέωση και ενημέρωση του Secret. Στην παραγωγή αυτή η διαδικασία συνήθως αυτοματοποιείται. Το ουσιώδες εδώ είναι να μπορείς να δείξεις πού τελειώνει το TLS και ποιο όνομα επαληθεύει ο client. Οι επιλογές listeners και certificate references περιγράφονται στον οδηγό TLS του Gateway API.

Σπάσ’ το επίτηδες

Πρώτα στείλε διαφορετικό host. Μετά ζήτησε ανύπαρκτο backend στο route. Είναι διαφορετικές βλάβες.

TERMINAL

```
curl -i -H 'Host: wrong.book.test' http://localhost:8088/
kubectl patch httproute rails-map --type=json -p \
  '[{"op":"replace","path":"/spec/rules/0/backendRefs/0/name",
    "value":"missing"}]'
kubectl get httproute rails-map -o yaml
kubectl get gateway book -o yaml
```

Στο route διάβασε τις συνθήκες κάτω από `status.parents`. Το `Accepted` αφορά την αποδοχή από τον γονικό Gateway. Το `ResolvedRefs` αφορά την επίλυση αναφορών όπως το backend. Ένα αντικείμενο μπορεί να έχει γίνει δεκτό και να έχει μη επιλύσιμη αναφορά. Κοίτα και το `observedGeneration`, ώστε να μη διαβάζεις απάντηση για προηγούμενη δήλωση.

TERMINAL

```
kubectl apply -f manifests/ch11/route.yaml
curl -fsS -H 'Host: rails.book.test' http://localhost:8088/
```

Η άσκηση τελειώνει όταν πετυχαίνουν ξανά HTTP και HTTPS και οι συνθήκες αντιστοιχούν στην τρέχουσα γενιά. Ο ακριβής HTTP κωδικός μόνος του δεν εντοπίζει πάντα τη βλάβη. Συνδύασέ τον με route, Service και EndpointSlices.

Το Ingress παραμένει υπαρκτό API, αλλά είναι frozen. Το ingress-nginx ήταν μία συγκεκριμένη υλοποίησή του και αποσύρθηκε τον Μάρτιο 2026. Δεν αποσύρθηκαν μαζί του όλες οι υλοποιήσεις Ingress. Για νέο υλικό επιλέγουμε Gateway API, σύμφωνα με την κατεύθυνση της τεκμηρίωσης Ingress και την ανακοίνωση απόσυρσης.

ΚΕΦΑΛΑΙΟ 12

ConfigMaps και Secrets

Μια νέα τιμή στο API δεν σημαίνει ότι η τρέχουσα διεργασία τη διαβάζει ήδη.

Μέχρι τώρα η έκδοση βρισκόταν μέσα στην εικόνα. Η ρύθμιση είναι άλλο είδος αλλαγής. Θέλουμε το ίδιο image να τρέχει με διαφορετικό μήνυμα ή διαφορετικό endpoint, χωρίς νέο build. Το ConfigMap κρατά μη ευαίσθητες τιμές. Το Secret είναι ο αντίστοιχος τύπος για δεδομένα που χρειάζονται περιορισμένη πρόσβαση.

Θα περάσουμε την ίδια τιμή με τρεις τρόπους στην ίδια εφαρμογή. Ως env var, ως αρχείο σε κανονικό volume mount και ως αρχείο μέσω `subPath`. Έπειτα θα αλλάξουμε μόνο το ConfigMap και θα δούμε ποιο από τα τρία ακολουθεί την αλλαγή.

> **Από το Kamal στο Kubernetes**
> Μια μεταβλητή περιβάλλοντος συνδέεται με την εκκίνηση της διεργασίας και στις δύο περιπτώσεις. Το ConfigMap προσθέτει αντικείμενο με χωριστή ζωή. Ο τρόπος που το διαβάζει το Pod καθορίζει πότε θα φανεί μια αλλαγή.

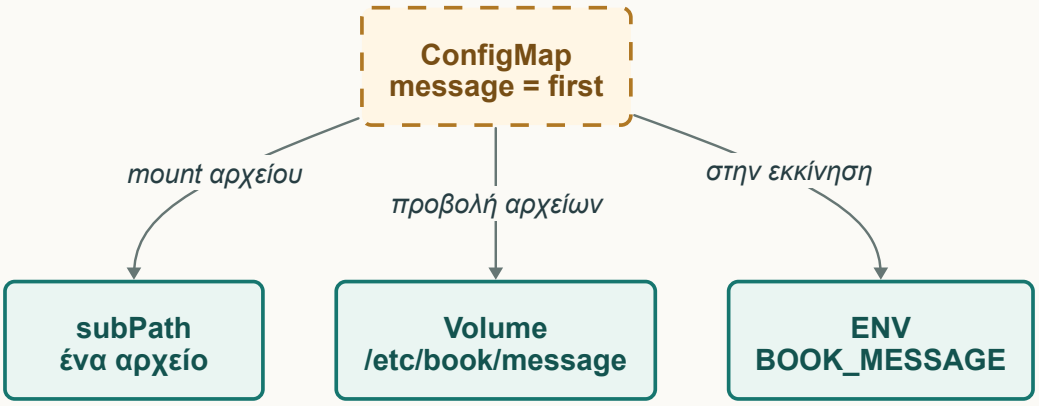
Μία πηγή, τρεις διαδρομές

manifests/ch12/configmap.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: book-settings
  namespace: k8s-book
data:
  message: first
```

TERMINAL

```
kubectl apply -f manifests/ch12/configmap.yaml
kubectl patch deployment rails-map --type=strategic \
  --patch-file manifests/ch12/config-patch.yaml
kubectl rollout status deployment/rails-map --timeout=180s
```



Σχήμα 12.1 · Η ίδια τιμή του ConfigMap φτάνει στο περιβάλλον του process, σε κανονικό mount και σε subPath mount. Η αρχική τιμή είναι ίδια, ο μηχανισμός ενημέρωσης διαφέρει.

Το patch προσθέτει `BOOK_MESSAGE` από `configMapKeyRef`, volume στο `/etc/book` και ένα δεύτερο mount του συγκεκριμένου αρχείου στο `/etc/book-subpath/message`. Η εφαρμογή επιστρέφει και τις τρεις τιμές στο `/config`. Τα αρχεία διαβάζονται σε κάθε αίτημα, ώστε να μη συγχέουμε την ενημέρωση του filesystem με caching της εφαρμογής.

TERMINAL

```
kubectl exec book-client -- ruby -rnet/http -e \
  'puts Net::HTTP.get(URI("http://rails-map/config"))'
```

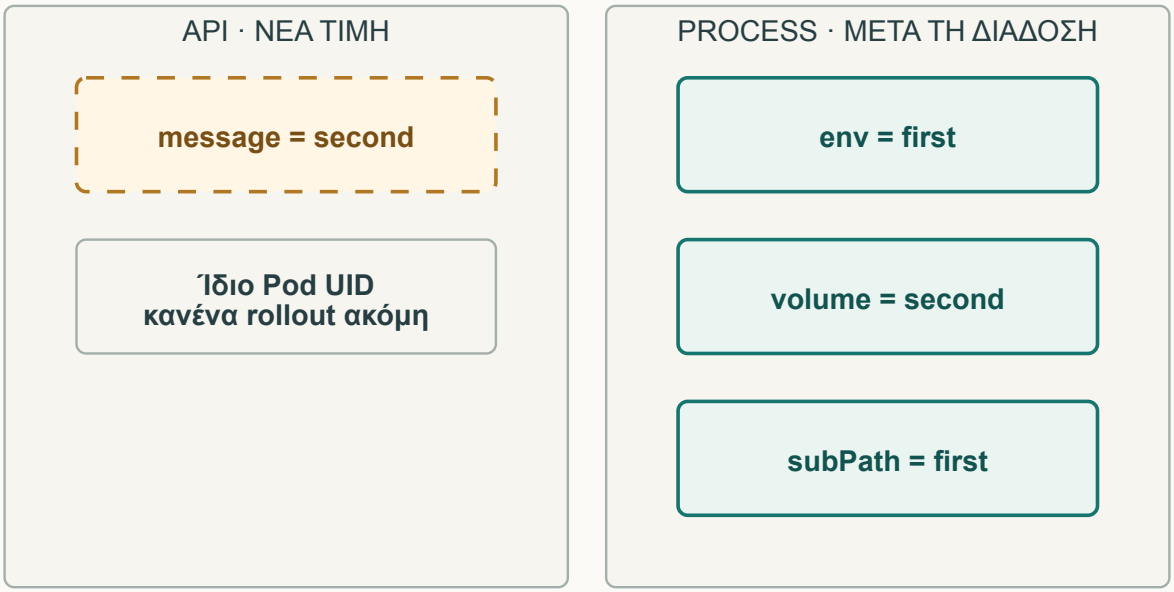
Αρχικά περιμένεις `first` και στα τρία πεδία. Κράτησε και το UID ενός web Pod. Το επόμενο βήμα δεν αλλάζει το Pod template και δεν ζητά rollout.

Άλλαξε μόνο το ConfigMap

TERMINAL

```
kubectl patch configmap book-settings --type=merge \
  -p '{"data":{"message":"second"}}'
```

Επανάλαβε το αίτημα στο `/config`. Μην θεωρήσεις αποτυχία το ότι το κανονικό volume δεν άλλαξε αμέσως. Η προβολή ενημερώνεται από το kubelet με καθυστέρηση που εξαρτάται από τον συγχρονισμό και την cache. Περίμενε και παρατήρησε χωρίς restart.



Σχήμα 12.2 · Μετά τη διάδοση της αλλαγής, το κανονικό volume δείχνει second. Το env και το subPath παραμένουν first μέσα στο ίδιο Pod.

Το env var ανήκει στο περιβάλλον της διεργασίας που ξεκίνησε. Η ενημέρωση του ConfigMap δεν ξαναγράφει το περιβάλλον της ζωντανής διεργασίας. Το κανονικό mount ενημερώνει τα προβαλλόμενα αρχεία. Το `subPath` mount δεν λαμβάνει αυτές τις ενημερώσεις.

Ακόμη και στο κανονικό mount, η εφαρμογή πρέπει να ξαναδιαβάσει το αρχείο ή να υποστηρίζει δικό της reload. Αν το διάβασε μόνο μία φορά στο boot, μπορεί το filesystem να έχει `second` και η μνήμη της εφαρμογής να κρατά `first`. Το Kubernetes δεν αλλάζει αυθαίρετα τα Ruby objects της διεργασίας σου. Οι τρεις συμπεριφορές περιγράφονται στα ConfigMaps.

Πότε χρειάζεται rollout

TERMINAL

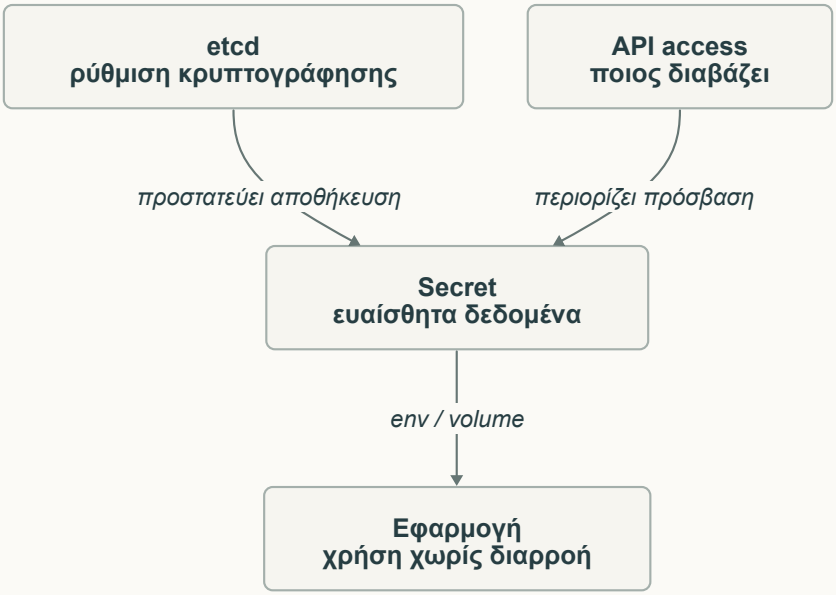
```
kubectl rollout restart deployment/rails-map
kubectl rollout status deployment/rails-map --timeout=180s
kubectl exec book-client -- ruby -rnet/http -e \
  'puts Net::HTTP.get(URI("http://rails-map/config"))'
```

Τα νέα Pods παίρνουν τη νέα τιμή στην εκκίνηση. Τώρα και τα τρία πεδία πρέπει να δείχνουν `second`. Τα UIDs αλλάζουν, επειδή το `rollout restart` προκαλεί αντικατάσταση Pods μέσω αλλαγής στο template. Δεν είναι η ίδια πράξη με restart ενός container μέσα στο ίδιο Pod.

Σε ένα πραγματικό deploy μπορείς να συνδέσεις την αλλαγή ρυθμίσεων με αλλαγή στο template, για παράδειγμα με annotation που περιέχει checksum των ρυθμίσεων. Η ουσία είναι να υπάρχει ρητός μηχανισμός που προκαλεί rollout όταν απαιτείται. Το να αποθηκεύσεις νέα τιμή σε ένα ConfigMap με το ίδιο όνομα δεν αρκεί από μόνο του.

Η επιλογή τρόπου φόρτωσης ακολουθεί τη συμπεριφορά της εφαρμογής. Για ρυθμίσεις που διαβάζονται μόνο στο boot, ένα συντονισμένο rollout είναι συχνά πιο προβλέψιμο από την ψευδαίσθηση δυναμικής ενημέρωσης. Για εφαρμογές με σωστό reload, το κανονικό volume μπορεί να είναι κατάλληλο.

Το Secret δεν είναι κρυπτογράφηση επειδή λέγεται Secret



Σχήμα 12.3 · Η προστασία εξαρτάται από πρόσβαση στο API, αποθήκευση και χρήση. Το base64 είναι αναπαράσταση του περιεχομένου, όχι όριο ασφαλείας.

Το Secret μπορεί να δοθεί ως env ή volume με παρόμοιους μηχανισμούς. Το `data` χρησιμοποιεί base64. Το `stringData` επιτρέπει να δώσεις απλό κείμενο που το API μετατρέπει. Κανένα από τα δύο δεν κάνει ασφαλή την αποθήκευση πραγματικών κωδικών σε δημόσιο repository.

Η δυνατότητα ανάγνωσης Secret από το API, η κρυπτογράφηση της αποθήκευσης, τα δικαιώματα της διεργασίας και η διαχείριση των logs είναι ξεχωριστές πλευρές. Το Kubernetes δεν εγγυάται κρυπτογράφηση στο etcd μόνο και μόνο επειδή χρησιμοποίησες τον τύπο Secret. Η σχετική ρύθμιση ανήκει στο cluster. Δες τα Secrets και τις οδηγίες χρήσης τους.

Στο εργαστήριο θα χρησιμοποιήσουμε αποκλειστικά δοκιμαστικές τιμές που δεν ανοίγουν κανέναν εξωτερικό λογαριασμό. Το `/config` επιστρέφει μόνο το δημόσιο μήνυμα. Δεν θα το μετατρέψουμε σε endpoint που εμφανίζει DATABASE_URL, κωδικούς ή ολόκληρο το περιβάλλον του process.

Η άσκηση είναι να εξηγήσεις την κατάσταση χωρίς να κοιτάξεις το διάγραμμα. Το ConfigMap γράφει `second`, το αρχείο γράφει `second`, η εφαρμογή εμφανίζει `first`. Πριν κατηγορήσεις το cluster, εντόπισε αν κοιτάς env, subPath ή τιμή που η εφαρμογή κράτησε από το boot. Αυτές είναι τρεις διαφορετικές διαδρομές προς το ίδιο σύμπτωμα.

ΚΕΦΑΛΑΙΟ 13

Postgres, Redis και επίμονα δεδομένα

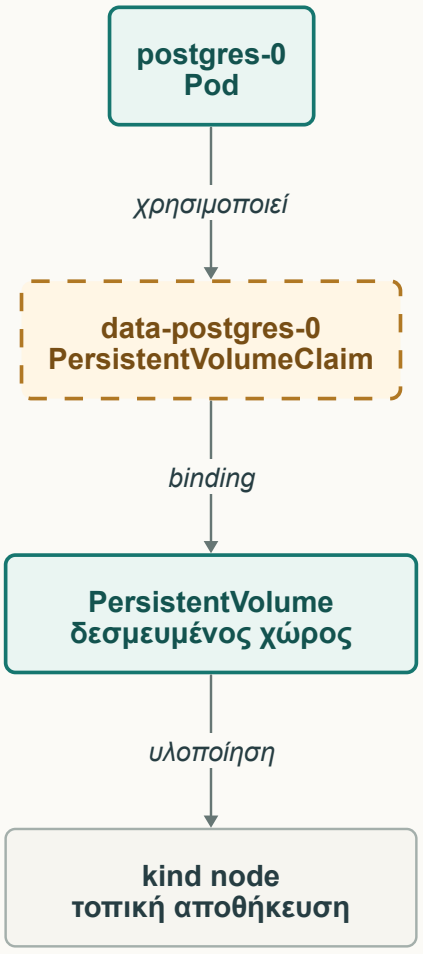
Το Pod μπορεί να αντικατασταθεί. Τα δεδομένα χρειάζονται δική τους ταυτότητα και δική τους διαδρομή ανάκαμψης.

Η εφαρμογή μας θα αποκτήσει έναν πίνακα εργασιών. Κάθε εγγραφή έχει μοναδικό token, κατάσταση και μετρητή ολοκληρώσεων. Αργότερα ο worker θα τη διαβάζει από την ουρά. Προς το παρόν θέλουμε να γράψουμε μια εγγραφή και να τη βρούμε μετά την αντικατάσταση του Pod της βάσης.

Το εργαστήριο χρησιμοποιεί ένα Postgres και ένα Redis. Η επιλογή ενός replica κρατά καθαρό το πείραμα αποθήκευσης. Δεν δημιουργεί βάση με αυτόματο failover. Ένα StatefulSet με τρία replicas Postgres θα ξεκινούσε τρεις διεργασίες. Δεν θα ρύθμιζε από μόνο του replication, εκλογή primary και προστασία από ταυτόχρονους writers.

> **Από το Kamal στο Kubernetes**
> Η βάση εξακολουθεί να χρειάζεται δίσκο και ελεγμένο backup. Το StatefulSet και το PVC περιγράφουν κομμάτια αυτής της λειτουργίας. Δεν μετατρέπουν ένα αντίγραφο Postgres σε σύστημα με failover.

Από το αίτημα χώρου στον δίσκο



Σχήμα 13.1 · Το Pod χρησιμοποιεί ένα PVC. Το PVC δεσμεύεται με PV. Η πραγματική αποθήκευση παρέχεται από την υλοποίηση του cluster.

Το **PersistentVolumeClaim** ζητά χώρο με συγκεκριμένα χαρακτηριστικά. Το **PersistentVolume** αντιπροσωπεύει τον χώρο που προσφέρεται. Το **StorageClass** επιλέγει τρόπο provision και σχετικές πολιτικές. Στο kind η κλάση `standard` χρησιμοποιεί την τοπική αποθήκευση των nodes.

TERMINAL

```
bash scripts/data-up.sh
kubectl get statefulsets
kubectl get storageclass
kubectl get pvc
kubectl get pv
```

Το script εφαρμόζει μόνο τις δοκιμαστικές ρυθμίσεις βάσης και τα StatefulSets. Περιμένει να γίνουν έτοιμα. Οι εικόνες Postgres και Redis έχουν συγκεκριμένες εκδόσεις και digests στα manifests. Ο κωδικός στο εργαστήριο είναι δημόσια δοκιμαστική τιμή, αποκλειστικά για αυτό το τοπικό περιβάλλον.

Το κρίσιμο τμήμα του StatefulSet είναι το template του claim.

YAML

```
volumeClaimTemplates:
  - metadata:
      name: data
    spec:
      accessModes:
        - ReadWriteOnce
      storageClassName: standard
      resources:
        requests:
          storage: 1Gi
```

Για το `postgres-0` δημιουργείται claim με όνομα `data-postgres-0`. Το `ReadWriteOnce` σημαίνει πρόσβαση εγγραφής από έναν node. Δεν σημαίνει γενικά «μόνο ένα Pod» όταν πολλά Pods βρίσκονται στον ίδιο node. Οι ακριβείς δυνατότητες εξαρτώνται και από το storage. Η διάκριση περιγράφεται στα Persistent Volumes.

Πρώτα schema, μετά web

Πριν ανοίξει το web με βάση, πρέπει να υπάρχει ο πίνακας. Χρησιμοποιούμε ένα Job με την εικόνα `v1` και περιμένουμε επιτυχία.

TERMINAL

```
kubectl create -f manifests/ch14/migrate-v1.yaml
kubectl wait job/book-migrate-v1 \
  --for=condition=Complete --timeout=150s
kubectl logs job/book-migrate-v1
kubectl apply -f manifests/ch13/web.yaml
kubectl rollout status deployment/rails-map --timeout=180s
```

Στην πρώτη εκτέλεση το Job δημιουργεί τον πίνακα. Αν επαναλαμβάνεις το κεφάλαιο και υπάρχει ήδη το ολοκληρωμένο Job, το `create` θα αρνηθεί δεύτερο αντικείμενο με το ίδιο όνομα. Έλεγξε την υπάρχουσα κατάστασή του. Για πραγματικές επαναλήψεις release, το επόμενο κεφάλαιο δημιουργεί νέο όνομα Job σε κάθε εκτέλεση.

Το web διαβάζει το DATABASE_URL από συγκεκριμένο κλειδί Secret. Δεν χρειάζεται να το εμφανίσεις στο terminal για να ελέγξεις αν συνδέεται. Το `/ready` και τα logs σύνδεσης μπορούν να δώσουν την απαραίτητη ένδειξη χωρίς εκτύπωση κωδικού.

Δημιούργησε μια εγγραφή από τον client.

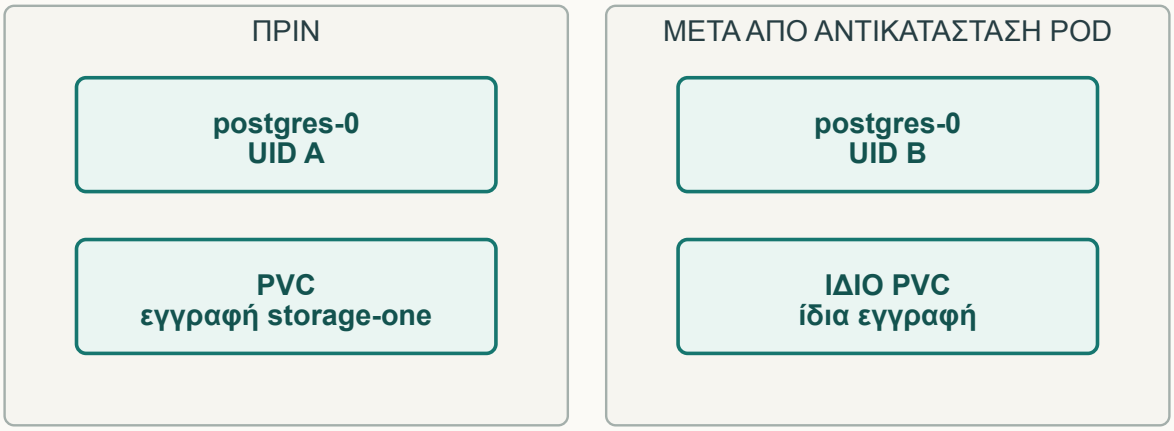
```
TERMINAL

kubectl exec book-client -- ruby -rnet/http -e \
  'puts Net::HTTP.post_form(URI("http://rails-map/tasks"),
    token: "storage-one").body'
kubectl exec book-client -- ruby -rnet/http -e \
  'puts Net::HTTP.get(URI("http://rails-map/tasks"))'
```

Σε αυτό το κεφάλαιο η εγγραφή παραμένει queued, επειδή το web δεν έχει ακόμη ενεργοποιημένη την ουρά. Η ύπαρξη της εγγραφής είναι το κριτήριο του πειράματος, όχι η ολοκλήρωση δουλειάς.

Αντικατάστησε μόνο το Pod της βάσης

Κράτησε το UID του Pod και του claim. Μετά διέγραψε μόνο το συγκεκριμένο Pod.

```
TERMINAL

kubectl get pod postgres-0 \
  -o jsonpath='{.metadata.uid}{"\n"}'
kubectl get pvc data-postgres-0 \
  -o jsonpath='{.metadata.uid}{"\n"}'
kubectl delete pod postgres-0
kubectl wait pod/postgres-0 --for=condition=Ready --timeout=180s
```

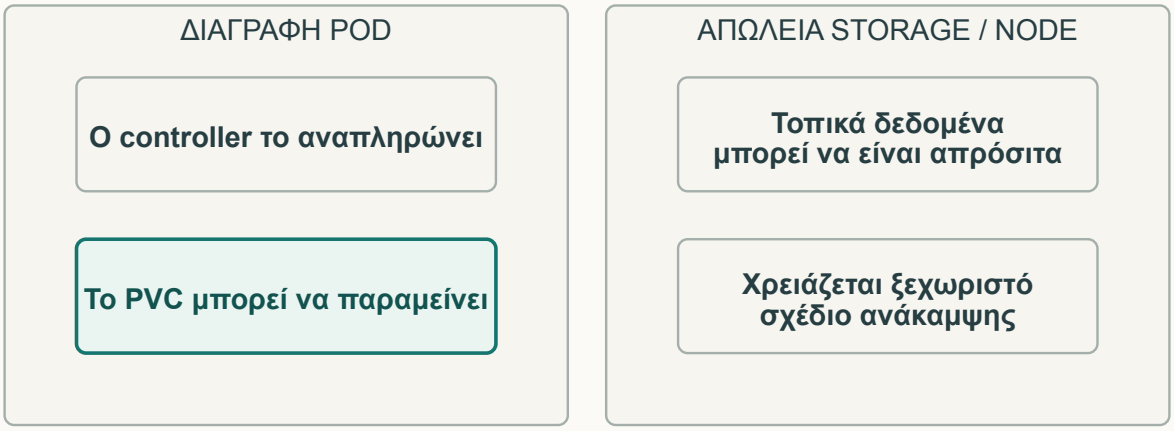


Σχήμα 13.2 · Το όνομα postgres-0 επανέρχεται, το Pod UID αλλάζει και το claim παραμένει. Η εγγραφή επιβιώνει επειδή τα δεδομένα δεν ζούσαν στο αναλώσιμο filesystem του container.

Επανάλαβε την ανάγνωση `/tasks`. Το `storage-one` πρέπει να υπάρχει. Αν δεν υπάρχει, η επιτυχία του `kubectl wait` δεν σώζει το πείραμα. Το αντικείμενο μπορεί να είναι έτοιμο με λάθος ή άδειο storage. Χρειαζόμαστε έλεγχο του ίδιου του δεδομένου.

Το StatefulSet δίνει σταθερή σειρά και ταυτότητα στα Pods του. Το όνομα `postgres-0` μπορεί να παραμένει ενώ το UID αλλάζει. Η διατήρηση claims κατά τη διαγραφή ή το scale έχει δικές της πολιτικές. Στο παράδειγμα χρησιμοποιούμε την προεπιλεγμένη διατήρηση, χωρίς πολιτική αυτόματης αφαίρεσης claims. Δες τα StatefulSets.

Τι δεν απέδειξε το πείραμα



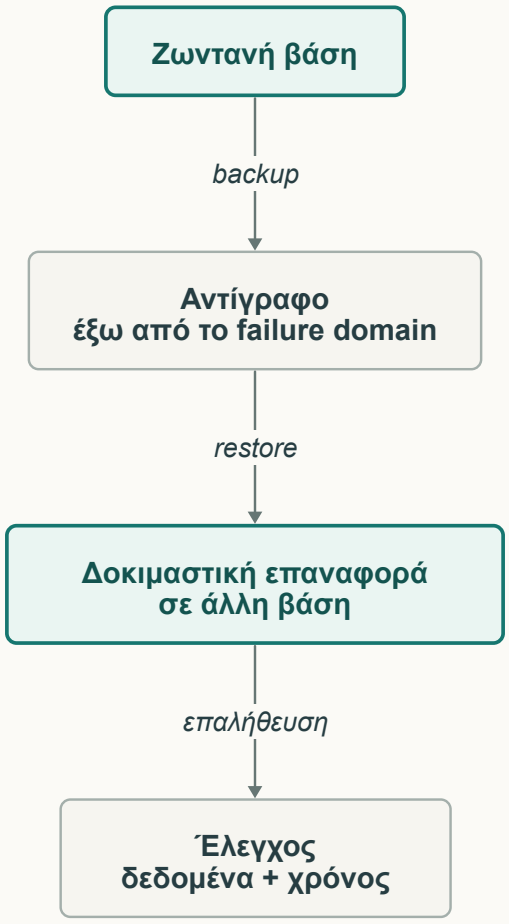
Σχήμα 13.3 · Η επιβίωση μετά τη διαγραφή Pod είναι ένα συγκεκριμένο τεστ. Η απώλεια του node ή του αποθηκευτικού μέσου είναι διαφορετικό τεστ.

Ο τοπικός δίσκος του kind node δεν μετατρέπεται σε δικτυακή αποθήκευση επειδή μπήκε πίσω από PVC. Αν αυτό το node είναι απρόσιτο, ένα Pod σε άλλο node δεν αποκτά αυτόματα πρόσβαση στα αρχεία του. Γι’ αυτό το πείραμα απώλειας node στο κεφάλαιο 17 θα χρησιμοποιήσει stateless web.

Η διαγραφή PVC είναι επίσης διαφορετική πράξη. Η reclaim policy του PV και η συμπεριφορά του provisioner καθορίζουν τι συμβαίνει στον υποκείμενο χώρο. Δεν τη χρησιμοποιούμε ως απλό reset της εφαρμογής. Θα μπορούσε να αφαιρέσει το μοναδικό αντίγραφο των δεδομένων.

Το Redis κρατά την ουρά σε δικό του PVC και χρησιμοποιεί AOF με συγχρονισμό ανά δευτερόλεπτο. Αυτό δεν συνεπάγεται μηδενική απώλεια σε κάθε απότομη βλάβη. Η πολιτική persistence του Redis και η εγγύηση παράδοσης της ουράς είναι διαφορετικές από τη διατήρηση του claim. Οι επιλογές τεκμηριώνονται στο Redis persistence.

Ένα backup πρέπει να μπορεί να επιστρέψει



Σχήμα 13.4 · Η χρήσιμη ακολουθία περιλαμβάνει λήψη, ξεχωριστή αποθήκευση, δοκιμαστική επαναφορά και έλεγχο του αποτελέσματος.

Για τη μικρή δοκιμαστική βάση μπορούμε να κρατήσουμε logical dump στο Mac.

TERMINAL

```
kubectl exec postgres-0 -- \
  pg_dump -U book -d book -Fc > build/book.dump
```

Το αρχείο έξω από το node είναι καλύτερο από ένα αντίγραφο δίπλα στα ίδια δεδομένα, αλλά παραμένει στο ίδιο Mac. Δεν είναι ανεξάρτητη προστασία από απώλεια του υπολογιστή. Η παραγωγή χρειάζεται σχέδιο φύλαξης και επαναφοράς που να αντιστοιχεί στα πραγματικά failure domains.

Δοκίμασε και την επαναφορά

Ένα dump που ολοκληρώθηκε χωρίς σφάλμα είναι η αρχή του ελέγχου. Χρειάζεται να αποδείξεις ότι μπορείς να το διαβάσεις και ότι περιέχει τα δεδομένα που περίμενες. Στο εργαστήριο κάνουμε restore σε νέα βάση μέσα στην ίδια δοκιμαστική Postgres. Η ενεργή βάση της εφαρμογής μένει στη θέση της.

TERMINAL

```
book_restore="book_restore_$(date +%s)"
kubectl exec postgres-0 -- \
  createdb -U book "$book_restore"
kubectl exec -i postgres-0 -- \
  pg_restore -U book -d "$book_restore" < build/book.dump
kubectl exec postgres-0 -- \
  psql -U book -d "$book_restore" \
  -c 'SELECT token, status FROM book_tasks;'
```

Η νέα βάση έχει διαφορετικό όνομα. Βρες το token που δημιούργησες πριν από το dump. Η παρουσία του ελέγχει συγκεκριμένο αποτέλεσμα της επαναφοράς, όχι μόνο το exit code του εργαλείου. Το αυτοματοποιημένο πείραμα του κεφαλαίου έκανε αυτή τη διαδικασία και βρήκε την αναμενόμενη εγγραφή ακριβώς μία φορά.

Η ίδια εγκατάσταση Postgres κάνει το τοπικό πείραμα μικρό. Δεν ελέγχει ανάκτηση μετά από απώλεια ολόκληρου node, διαφορετική έκδοση server ή πρόσβαση σε απομακρυσμένο αποθηκευτικό χώρο. Αυτές είναι χωριστές δοκιμές που θα χρειαστεί η κανονική στρατηγική backup.

Η βάση restore παραμένει για να μπορείς να την εξετάσεις. Για να τη χρησιμοποιήσει η εφαρμογή θα χρειαζόταν συνειδητή αλλαγή της σύνδεσης. Το restore δεν άλλαξε το `DATABASE_URL` του web και δεν έκανε τα νέα δεδομένα ενεργά από μόνο του.

Η μικρή άσκηση είναι να γράψεις δύο κριτήρια. Πόσα δεδομένα επιτρέπεται να χαθούν και σε πόσο χρόνο πρέπει να επανέλθει η υπηρεσία. Μετά εξήγησε ποια από αυτά απέδειξε η διαγραφή του Pod. Η σωστή απάντηση δεν είναι «έχουμε PVC, άρα έχουμε backup». Οι δυνατότητες του dump περιγράφονται στο pg_dump.

ΚΕΦΑΛΑΙΟ 14

Workers, Jobs και migrations

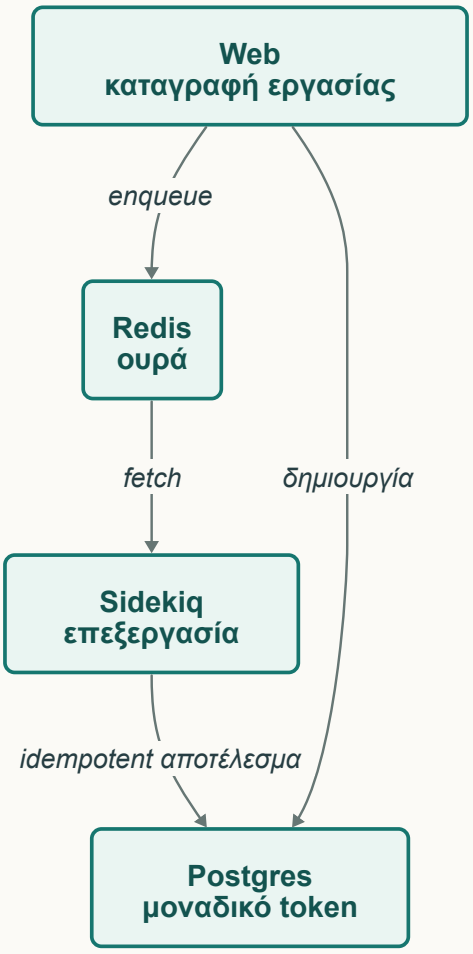
Το web απαντά σε αιτήματα. Ο worker ολοκληρώνει δουλειές. Το migration προετοιμάζει το έδαφος πριν αλλάξουν και οι δύο.

Χρησιμοποιούμε την ίδια εικόνα με διαφορετική εντολή. Στο web ξεκινά Puma. Στον worker ξεκινά Sidekiq. Ο κώδικας και οι εξαρτήσεις είναι κοινά, αλλά τα Deployments μπορούν να έχουν διαφορετικό πλήθος replicas, resources και κύκλο rollout.

Η εφαρμογή δημιουργεί εγγραφή στη βάση με μοναδικό token και βάζει το ID στην ουρά. Ο worker περιμένει μερικά δευτερόλεπτα για να έχουμε χρόνο παρατήρησης και μετά σημειώνει ολοκλήρωση. Το αποτέλεσμα στη βάση προστατεύεται από lock και έλεγχο αν έχει ήδη ολοκληρωθεί.

> **Από το Kamal στο Kubernetes**
> Ο χωριστός ρόλος worker παραμένει χωριστή μονάδα κλίμακας. Το migration αποκτά δικό του Job και ρητό σημείο αναμονής. Η σειρά του release και η ασφαλής επανάληψη εργασίας παραμένουν ευθύνες που πρέπει να σχεδιάσεις.

Η διαδρομή μιας δουλειάς



Σχήμα 14.1 · Η βάση κρατά την επιχειρησιακή κατάσταση. Η ουρά μεταφέρει την εκκρεμή εργασία στον worker. Αυτές είναι διαφορετικές εγγραφές και δεν γίνονται αυτόματα μία συναλλαγή.

TERMINAL

```
bash scripts/release.sh v1
kubectl get deployments
kubectl logs deployment/worker --tail=30
```

Η εντολή εκκίνησης του worker βρίσκεται στο manifest.

YAML

```yaml
command:
  - bundle
  - exec
  - sidekiq
  - -r
  - ./application.rb
  - -c
  - "2"
  - -t
  - "20"
```

Η concurrency είναι δύο για το μικρό εργαστήριο. Δεν προκύπτει από τον αριθμό web Pods. Ούτε τα replicas του worker υπολογίζονται από το Rails connection pool. Αυτά τα μεγέθη πρέπει να συμφωνούν με τη χωρητικότητα της βάσης και το είδος των jobs.

Στείλε μια νέα δουλειά και παρακολούθησέ την.

TERMINAL

```
kubectl exec book-client -- ruby -rnet/http -e \
  'puts Net::HTTP.post_form(URI("http://rails-map/tasks"),
    token: "worker-one").body'
kubectl logs deployment/worker -f
```

Από άλλο terminal διάβασε το `/tasks`. Η εγγραφή πρέπει να περάσει σε `done` με `completions` ίσο με ένα. Αν ξαναστείλεις το ίδιο token, δεν δημιουργείται δεύτερη επιχειρησιακή εγγραφή. Η διπλή παράδοση ενός job παραμένει πιθανή, αλλά ο τελικός μετρητής δεν πρέπει να αυξηθεί δεύτερη φορά.

Υπάρχει ένα σκόπιμα ορατό όριο. Η εγγραφή στη βάση και το enqueue στο Redis δεν αποτελούν κοινή atomic transaction. Αν αποτύχει το δεύτερο βήμα, χρειάζεται μηχανισμός επανεντοπισμού εκκρεμών εγγραφών ή σχεδίαση outbox. Δεν λύνεται αυτό το πρόβλημα με περισσότερα replicas.

Ένα migration ανά release

Μην βάλεις το `db:migrate` στην εντολή εκκίνησης κάθε web Pod. Σε rollout με surge μπορούν να ξεκινήσουν πολλά Pods κοντά χρονικά. Η εκκίνηση της εφαρμογής και η αλλαγή schema είναι διαφορετικές εργασίες με διαφορετικό κριτήριο επιτυχίας.

Σχήμα 14.2 · Το release περιμένει να ολοκληρωθεί το Job πριν ενημερώσει τα Deployments. Η αποτυχία του migration σταματά τη διαδρομή πριν από το web και τον worker.

Το `release.sh` δημιουργεί νέο όνομα Job για τη συγκεκριμένη εκτέλεση. Περιμένει `Complete` με timeout. Μόνο αν πετύχει συνεχίζει στην αλλαγή εικόνας του web και του worker. Ένα ολοκληρωμένο Job με ίδιο όνομα δεν χρησιμοποιείται ως απόδειξη ότι εκτελέστηκε μια καινούρια προσπάθεια.

Το Job έχει `restartPolicy: Never`, `backoffLimit: 0` και προθεσμία εκτέλεσης. Επιλέξαμε να εμφανίζεται καθαρά η πρώτη αποτυχία. Άλλα jobs μπορεί να χρειάζονται retries, αλλά η επανάληψη migration πρέπει να είναι κατανοητή και ασφαλής για τη συγκεκριμένη αλλαγή. Οι μηχανισμοί τερματισμού και επανάληψης περιγράφονται στα Jobs.

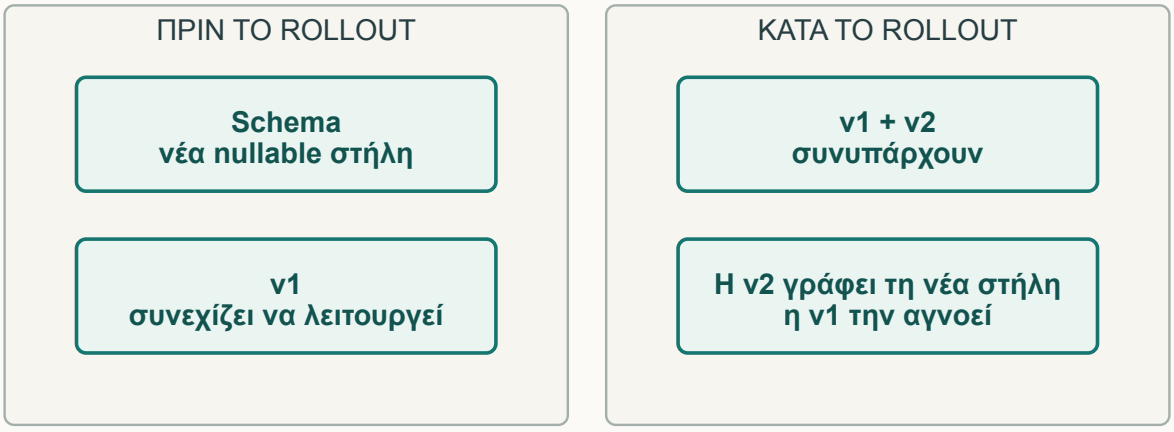
TERMINAL

```
bash scripts/release.sh v2 --fail-migration
kubectl get jobs -l app=book-migration
kubectl get deployment rails-map worker -o wide
```

Το script πρέπει να τελειώσει με αποτυχία πριν αλλάξουν οι εικόνες. Το όνομα του τελευταίου Job βρίσκεται στο `build/last-migration-job`, ώστε να διαβάσεις τα δικά του logs. Η αποτυχία είναι ένας ελεγχόμενος διακόπτης του παραδείγματος, πριν από το πραγματικό migration.

Η νέα στήλη και η παλιά έκδοση

Η `v2` προσθέτει nullable στήλη `processed_by`. Η `v1` συνεχίζει να διαβάζει τον ίδιο πίνακα και δεν απαιτεί να λείπει η νέα στήλη. Ο worker της `v2` τη συμπληρώνει όταν ολοκληρώνει μια εργασία.



Σχήμα 14.3 · Η επέκταση του schema προηγείται. Οι δύο εκδόσεις παραμένουν συμβατές κατά τη συνύπαρξη και κατά την επαναφορά της εφαρμογής.

TERMINAL

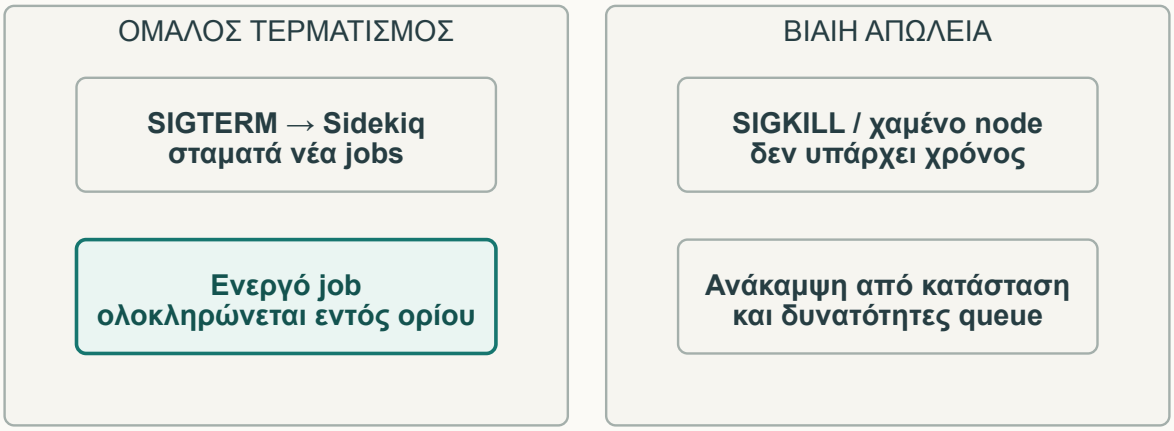
```
bash scripts/release.sh v2
```

Η σειρά έχει πρακτική συνέπεια. Αν μετονομάζαμε ή διαγράφαμε άμεσα στήλη που χρησιμοποιεί η `v1`, θα σπάγαμε τα παλιά Pods όσο ακόμη εξυπηρετούν. Μια συνηθισμένη ασφαλής διαδρομή είναι επέκταση, συμβατός κώδικας, μεταφορά δεδομένων όταν χρειάζεται και αφαίρεση σε επόμενο release αφού πάψει η χρήση.

Το rollback εφαρμογής δεν πρέπει να τρέχει αυτόματα `db:rollback`. Η μετανάστευση δεδομένων μπορεί να μην είναι αναστρέψιμη, ακόμη και αν το αρχείο migration έχει μέθοδο `down`. Το ερώτημα είναι αν η προηγούμενη εφαρμογή λειτουργεί με το τρέχον schema. Οι δυνατότητες του Rails εξηγούνται στα Active Record Migrations.

Μια δουλειά στη μέση του τερματισμού

Το timeout του Sidekiq είναι 20 δευτερόλεπτα και το `terminationGracePeriodSeconds` είναι 30. Ο worker πρέπει να λάβει SIGTERM, να πάψει να παίρνει νέες δουλειές και να έχει χρόνο να ολοκληρώσει ή να χειριστεί τις ενεργές σύμφωνα με τη συμπεριφορά της ουράς.



Σχήμα 14.4 · Ο ομαλός τερματισμός δίνει χρόνο στον worker. Η βίαιη απώλεια process ή node δεν περνά από το ίδιο μονοπάτι.

Στείλε νέο token, περίμενε να γίνει `working` και διέγραψε το συγκεκριμένο worker Pod με τον κανονικό grace period. Μην χρησιμοποιήσεις force deletion. Έλεγξε ότι η δουλειά φτάνει σε `done`, ότι το `completions` παραμένει ένα και ότι δημιουργείται νέος worker με άλλο UID.

Η επιτυχία αυτού του τεστ αφορά graceful shutdown. Το βασικό fetch του Sidekiq δεν υπόσχεται αυτόματη διάσωση κάθε in-flight job σε SIGKILL. Η ουρά, το persistence και η επιχειρησιακή συμφιλίωση καθορίζουν την ανάκαμψη. Το παράδειγμα διαθέτει `bin/requeue` για παλιές μη ολοκληρωμένες εγγραφές. Οι διαφορές περιγράφονται στις οδηγίες Sidekiq σε Kubernetes και Reliability.

CronJob και επανάληψη

Το CronJob του εργαστηρίου αναζητά εκκρεμείς εγγραφές ανά πέντε λεπτά. Το `concurrencyPolicy: Forbid` αποτρέπει ταυτόχρονες εκτελέσεις Jobs που δημιουργεί το ίδιο CronJob. Δεν είναι καθολικό lock της επιχειρησιακής εργασίας και δεν αποκλείει διπλές παραδόσεις από άλλον δρόμο.

TERMINAL

```
kubectl apply -f manifests/ch14/cronjob.yaml
kubectl get cronjob requeue-stale
kubectl create job requeue-manual --from=cronjob/requeue-stale
kubectl wait job/requeue-manual \
  --for=condition=Complete --timeout=120s
```

Το χειροκίνητο Job είναι νέα εκτέλεση. Για δεύτερη δοκιμή χρειάζεται άλλο όνομα. Η άσκηση είναι να εντοπίσεις πού ακριβώς αποτρέπεται το διπλό αποτέλεσμα στη βάση. Η σωστή θέση βρίσκεται στον κώδικα της εφαρμογής και στον μοναδικό περιορισμό, όχι στη λέξη `Forbid`. Οι χρονικοί κανόνες και οι περιορισμοί περιγράφονται στα CronJobs.

ΚΕΦΑΛΑΙΟ 15

Probes που λένε την αλήθεια

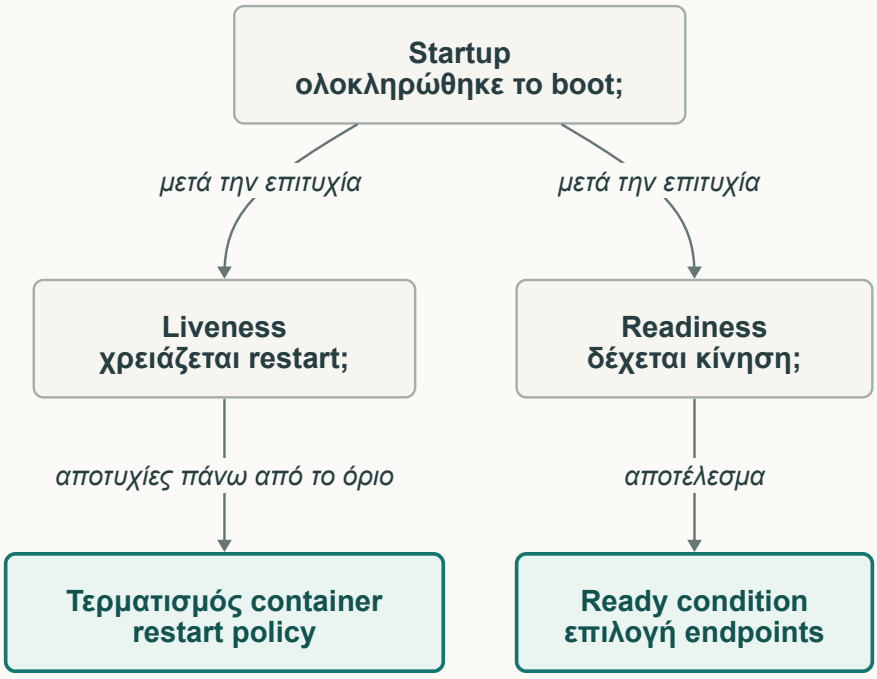
Τρεις έλεγχοι, τρεις διαφορετικές αποφάσεις. Μην τους κάνεις όλους το ίδιο endpoint από συνήθεια.

Έχουμε ήδη χρησιμοποιήσει readiness. Τώρα θα το τοποθετήσουμε δίπλα στο startup και στο liveness. Το kubelet εκτελεί αυτούς τους ελέγχους, αλλά η εφαρμογή είναι αυτή που πρέπει να δώσει μια χρήσιμη απάντηση. Ένα probe που λέει πάντα 200 δεν δημιουργεί υγεία. Ένα probe που αποτυγχάνει σε κάθε μικρή καθυστέρηση μπορεί να τη χειροτερεύει.

Σκέψου ένα Rails process που χρειάζεται χρόνο για boot. Έπειτα εξυπηρετεί σωστά, αλλά κάποια στιγμή η βάση επιβραδύνεται. Η σωστή αντίδραση μπορεί να διαφέρει σε καθεμία από αυτές τις φάσεις.

> **Από το Kamal στο Kubernetes**
> Ένα health endpoint είναι γνώριμο. Το Kubernetes χωρίζει τη χρήση του σε εκκίνηση, αποδοχή κίνησης και πιθανό restart. Η ίδια HTTP αποτυχία έχει διαφορετική συνέπεια ανάλογα με το probe που την παρατηρεί.

Ποια απόφαση παίρνει κάθε probe



Σχήμα 15.1 · Το startup επιτρέπει να αρχίσουν οι άλλοι έλεγχοι. Το readiness επηρεάζει την αποδοχή κίνησης. Το liveness μπορεί να προκαλέσει restart του container.

Probe	Ερώτηση	Συνέπεια επαναλαμβανόμενης αποτυχίας
Startup	Ολοκληρώθηκε η αρχική εκκίνηση;	Τερματισμός container και χειρισμός σύμφωνα με restart policy
Readiness	Μπορεί τώρα να δεχτεί την κίνηση που περιμένουμε;	Το Pod δεν είναι Ready
Liveness	Είναι σε κατάσταση όπου χρειάζεται επανεκκίνηση;	Τερματισμός container και χειρισμός σύμφωνα με restart policy

Όταν υπάρχει startup probe, τα liveness και readiness δεν αρχίζουν πριν πετύχει. Έτσι το αργό boot δεν αντιμετωπίζεται ως κολλημένη εφαρμογή από το πρώτο δευτερόλεπτο. Το startup, όμως, χρειάζεται και αυτό όριο. Δεν είναι αναμονή χωρίς τέλος.

Η πλήρης web διαμόρφωση του κεφαλαίου προσθέτει τους δύο ελέγχους στο παράδειγμα με βάση και worker.

TERMINAL

```
kubectl apply -f manifests/ch15/web.yaml
kubectl rollout status deployment/rails-map --timeout=180s
```

Η εικόνα αυτού του αρχείου είναι `v1`. Αν έχεις ολοκληρώσει το release `v2`, το apply γυρίζει συνειδητά το web στην `v1`, η οποία λειτουργεί με το επεκταμένο schema. Πρόκειται για γνωστή αρχική κατάσταση του πειράματος, όχι για σιωπηρή αναίρεση του migration.

Δώσε χρόνο στο σωστό σημείο

YAML

```
startupProbe:
  httpGet:
    path: /live
    port: http
  periodSeconds: 2
  failureThreshold: 30
livenessProbe:
  httpGet:
    path: /live
    port: http
  periodSeconds: 10
  timeoutSeconds: 2
  failureThreshold: 3
```

Το startup επιτρέπει περίπου ένα λεπτό αποτυχημένων περιοδικών προσπαθειών. Αυτό είναι χρήσιμο μέγεθος σχεδιασμού, όχι ακριβές χρονόμετρο τερματισμού. Η πρώτη εκτέλεση, η διάρκεια των ελέγχων και ο συγχρονισμός επηρεάζουν την πραγματική παρατήρηση.

Το liveness της εφαρμογής απαντά χωρίς έλεγχο βάσης. Ένα προσωρινό πρόβλημα Postgres δεν σημαίνει απαραίτητα ότι χρειάζεται restart όλων των web. Αν ο επανεκκινημένος server συναντήσει την ίδια εξάρτηση, μπορεί απλώς να προσθέσεις boot και νέες συνδέσεις σε ένα ήδη πιεσμένο σύστημα.

Το readiness μπορεί να εκφράζει διαφορετική απόφαση. Στο δείγμα ελέγχει τη δυνατότητα σύνδεσης όταν ενεργοποιείται η βάση. Σε μια μεγαλύτερη εφαρμογή μπορεί να χρειάζεται πιο προσεκτικός διαχωρισμός ανά διαθέσιμη λειτουργία. Οι έννοιες περιγράφονται στα Liveness, Readiness and Startup Probes.

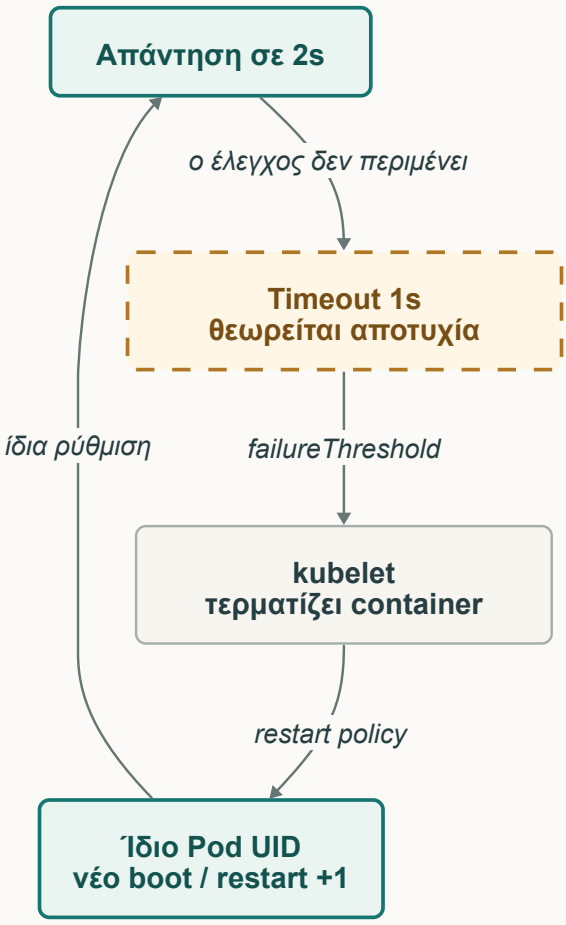
Σπάσ’ το επίτηδες με λάθος timeout

Για να μη χαλάσουμε την κανονική εφαρμογή, δημιουργούμε μικρό χωριστό Deployment με label `app=probe-loop`. Δεν το επιλέγει το Service `rails-map`.

TERMINAL

```
kubectl apply -f manifests/ch15/probe-loop.yaml
kubectl get pods -l app=probe-loop -w
```

Το `/live` αυτής της διεργασίας καθυστερεί σκόπιμα δύο δευτερόλεπτα. Το liveness περιμένει μόνο ένα και θεωρεί μία αποτυχία αρκετή. Η εφαρμογή μπορεί να απαντήσει, αλλά ο έλεγχος την καταδικάζει νωρίτερα.



Σχήμα 15.2 · Το σύντομο timeout παράγει αποτυχία probe, τερματισμό και νέο boot. Αν δεν αλλάξει η αιτία, ο κύκλος επαναλαμβάνεται στο ίδιο Pod.

TERMINAL

```
kubectl describe pods -l app=probe-loop
kubectl logs deployment/probe-loop --previous
```

Σύνδεσε τα `Unhealthy` και `Killing` events με την αύξηση του restart count. Το `--previous` ζητά logs από την προηγούμενη εκτέλεση του container μέσα στο συγκεκριμένο Pod. Δεν ψάχνει αυθαίρετα όλους τους παλιούς Pods του Deployment.

Η διόρθωση στο εργαστήριο αφαιρεί τη σκόπιμη καθυστέρηση, χωρίς να χαλαρώνει τυφλά κάθε probe.

TERMINAL

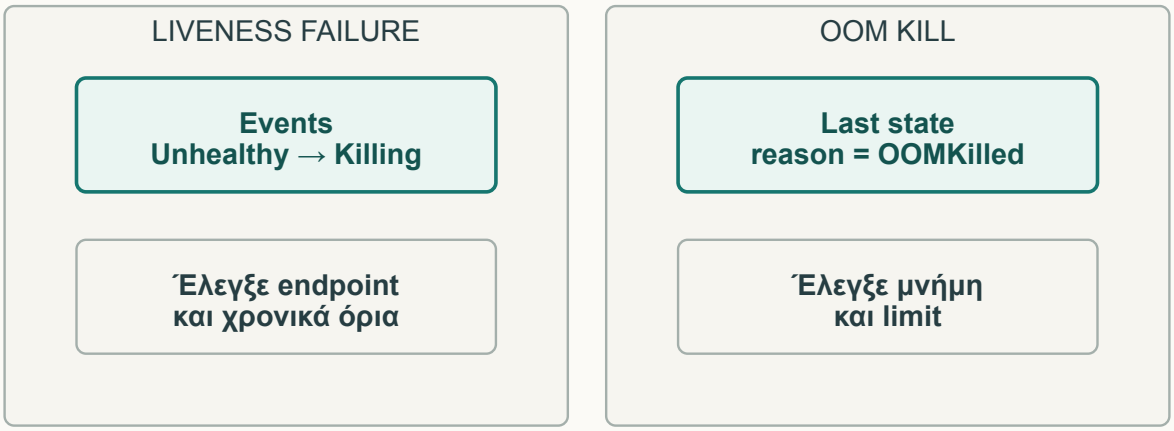
```
kubectl set env deployment/probe-loop LIVE_DELAY=0
kubectl rollout status deployment/probe-loop --timeout=180s
```

Παρατήρησε το νέο Pod για αρκετούς κύκλους probe. Δεν αρκεί ένα στιγμιαίο `Running` αμέσως μετά την εκκίνηση. Όταν ολοκληρώσεις το πείραμα, μηδένισε τα replicas του βοηθητικού Deployment.

TERMINAL

```
kubectl scale deployment/probe-loop --replicas=0
```

Ένα υγιές probe έχει μικρό κόστος



Σχήμα 15.3 · Η διάγνωση ξεκινά από το γεγονός που προηγήθηκε του restart. Ένα liveness failure και ένα OOMKilled χρειάζονται διαφορετική διόρθωση.

Το endpoint πρέπει να είναι φθηνό, προβλέψιμο και σχετικό με την απόφαση. Ένα βαρύ query σε κάθε probe πολλών replicas μπορεί να γίνει μέρος του προβλήματος. Ένα endpoint που περιμένει εξωτερικό API με μεγάλο timeout μπορεί να κρατά threads απασχολημένα και να εμφανίζει παράγωγες αποτυχίες.

Τα HTTP probes προέρχονται από το kubelet προς το Pod. Δεν περνούν υποχρεωτικά από το δημόσιο Gateway και δεν ελέγχουν όλη τη διαδρομή του χρήστη. Ένα επιτυχημένο readiness δεν αποδεικνύει ότι το εξωτερικό πιστοποιητικό είναι σωστό ή ότι το HTTPRoute δείχνει στο σωστό Service.

Η άσκηση είναι να γράψεις μια περίπτωση όπου θα αποτύγχανε μόνο το readiness και μια περίπτωση όπου θα δικαιολογούνταν liveness failure. Έπειτα σύνδεσε καθεμία με την ενέργεια που περιμένεις. Αν και στις δύο περιμένεις restart, γύρισε στον πίνακα. Η επίσημη οδηγία διαμόρφωσης probes δίνει τα πεδία και τα όρια κάθε τύπου.

ΚΕΦΑΛΑΙΟ 16

Requests, limits και QoS

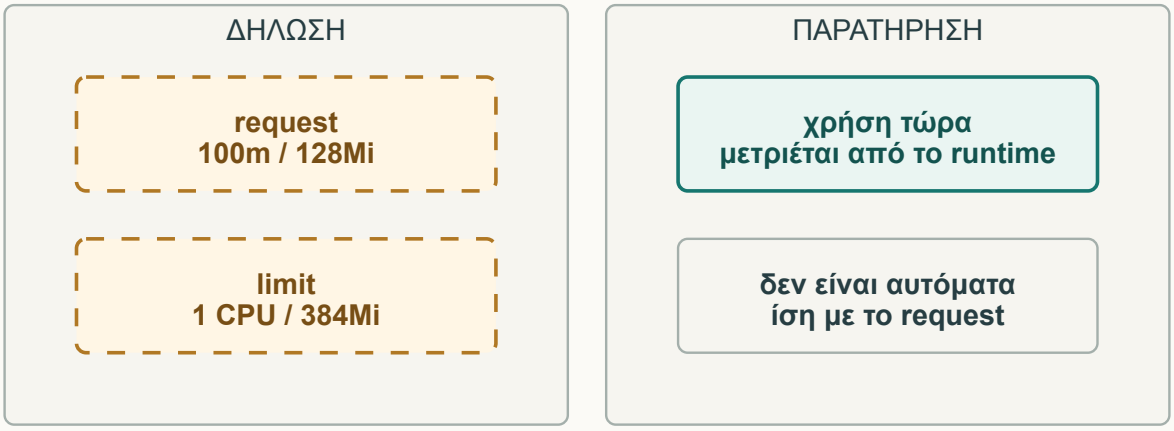
Το request βοηθά να βρεθεί θέση. Το limit περιορίζει τη χρήση. Η πραγματική κατανάλωση είναι τρίτο μέγεθος.

Στα manifests υπάρχουν ήδη CPU και μνήμη. Τώρα θα τα διαβάσουμε ως συμφωνία με το cluster. Το request δεν είναι πρόβλεψη που ενημερώνεται μόνη της επειδή η εφαρμογή έγινε βαρύτερη. Είναι η δηλωμένη ποσότητα που χρησιμοποιεί ο scheduler για την τοποθέτηση και που επηρεάζει την κατανομή πόρων.

Το limit δεν είναι εγγυημένη διαθέσιμη χωρητικότητα. Ένα Pod μπορεί να τοποθετηθεί με βάση το request, ακόμη κι αν όλα τα containers του node δεν μπορούν να φτάσουν ταυτόχρονα στα limits τους. Η σχέση ανάμεσα στα δύο επηρεάζει πόσο πυκνά τοποθετείς εφαρμογές.

> **Από το Kamal στο Kubernetes**
> Σε γνωστούς servers μπορεί ήδη να έχεις μετρήσει τι χωρά. Εδώ το request δίνει αυτή την πληροφορία και στον scheduler. Το limit αφορά την εκτέλεση και δεν αποτελεί μέτρηση της σημερινής χρήσης.

Τρία νούμερα στο ίδιο γράφημα



Σχήμα 16.1 · Το request είναι δήλωση για τοποθέτηση. Η κατανάλωση μετριέται κατά την εκτέλεση. Το limit είναι όριο που επιβάλλεται από το runtime και το λειτουργικό σύστημα.

YAML

```yaml
resources:
  requests:
    cpu: 100m
    memory: 128Mi
  limits:
    cpu: "1"
    memory: 384Mi
```

Το `100m` είναι ένα δέκατο CPU. Το `1` είναι μία CPU μονάδα, ανεξάρτητα από το ότι σε διαφορετικά περιβάλλοντα αυτή μπορεί να αντιστοιχεί σε διαφορετική φυσική απόδοση. Το `Mi` είναι δυαδική μονάδα μνήμης. Μην αντικαθιστάς αδιάφορα το `Mi` με `m`. Το μικρό `m` σημαίνει milli και παράγει εντελώς διαφορετική ποσότητα.

Με Metrics Server διαθέσιμο μπορούμε να συγκρίνουμε δήλωση και μέτρηση.

TERMINAL

```
bash scripts/cluster-up.sh --with-metrics
kubectl top pods
kubectl describe node k8s-book-control-plane
```

Το `top` δείχνει πρόσφατες μετρήσεις. Το τμήμα `Allocated resources` στο node συνοψίζει δηλωμένα requests και limits. Ένας node μπορεί να έχει χαμηλή πραγματική CPU χρήση και παρ' όλα αυτά να μη χωρά άλλο request. Η απόφαση τοποθέτησης δεν βασίζεται μόνο στην εικόνα της τελευταίας στιγμής.

Ένα Pod που δεν χωρά

Το βοηθητικό Deployment ζητά 99 CPUs. Ο τοπικός node δεν διαθέτει αυτή τη χωρητικότητα.

TERMINAL

```
kubectl apply -f manifests/ch16/pending.yaml
kubectl get pods -l app=pending-lab -o wide
kubectl describe pods -l app=pending-lab
```

Το Pod πρέπει να μείνει Pending χωρίς ανάθεση node. Στα events αναζήτησε `FailedScheduling` και την έλλειψη CPU. Δεν υπάρχει λόγος να ψάξεις τα logs του Ruby process. Δεν έχει ξεκινήσει.

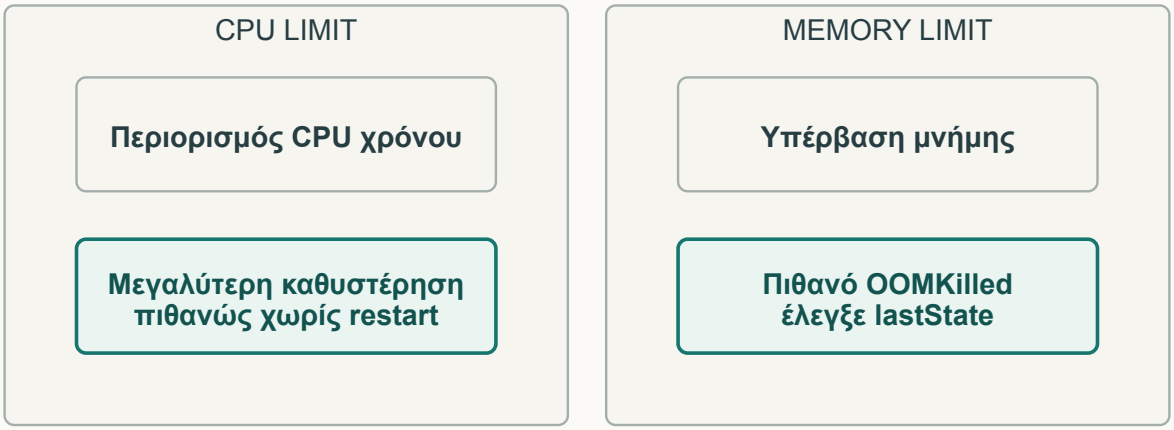
Διόρθωσε το request και περίμενε πραγματική εκτέλεση.

TERMINAL

```
kubectl set resources deployment/pending-lab \
  --requests=cpu=10m,memory=32Mi \
  --limits=cpu=100m,memory=64Mi
kubectl rollout status deployment/pending-lab --timeout=180s
kubectl scale deployment/pending-lab --replicas=0
```

Η μικρή διεργασία απλώς περιμένει. Τα νούμερα της διόρθωσης ανήκουν σε αυτό το βοηθητικό container, όχι στο Rails web. Το να αντιγράψεις requests από άσχετη εφαρμογή επειδή «εκεί δούλεψαν» δεν είναι μέτρηση.

CPU και μνήμη αποτυγχάνουν διαφορετικά



Σχήμα 16.2 · Στην CPU μπορεί να επιβληθεί throttling και να αυξηθεί η καθυστέρηση. Στη μνήμη η υπέρβαση μπορεί να οδηγήσει σε OOM kill. Η ερμηνεία χρειάζεται την πραγματική κατάσταση τερματισμού.

Το CPU limit συνήθως εκδηλώνεται ως περιορισμός χρόνου CPU. Η διεργασία δεν χρειάζεται να τερματιστεί για να γίνει πολύ αργή. Μπορεί να αυξηθούν οι χρόνοι απόκρισης και έπειτα να αποτυγχάνουν probes που είχαν οριακά timeouts.

Η μνήμη δεν ανακτάται με τον ίδιο τρόπο. Όταν ένας container ξεπεράσει το επιτρεπόμενο όριο, μπορεί να τερματιστεί από τον μηχανισμό OOM. Χρειάζεται να δεις το `lastState.terminated.reason` και όχι να θεωρήσεις κάθε exit code 137 απόδειξη της ίδιας αιτίας. Το SIGKILL έχει και άλλες πηγές.

TERMINAL

```
kubectl apply -f manifests/ch16/oom.yaml
kubectl get pods -l app=memory-lab -w
kubectl describe pods -l app=memory-lab
```

Η Ruby διεργασία του πειράματος κρατά συνεχώς νέα blocks μνήμης. Το όριο είναι 32Mi. Περίμενε ένδειξη `OOMKilled` και επανεκκίνηση στο ίδιο Pod. Μετά σταμάτησε το πείραμα.

TERMINAL

```
kubectl scale deployment/memory-lab --replicas=0
```

Σε πραγματικό Rails process δεν αρκεί να αυξάνεις μνήμη μέχρι να εξαφανιστεί το restart. Κοίτα boot, steady state, concurrency, μεγάλα payloads και διάρκεια λειτουργίας. Άλλο ανεπαρκές όριο

για νόμιμη αιχμή και άλλο κατανάλωση που ανεβαίνει χωρίς να σταθεροποιείται. Οι μηχανισμοί ορίζονται στο Resource Management.

Η κατηγορία QoS

Σχήμα 16.3 · Οι κατηγορίες προκύπτουν από τις δηλώσεις πόρων. Δεν είναι βαθμολογία ποιότητας του κώδικα και δεν εγγυώνται ότι μια εφαρμογή δεν θα τερματιστεί.

Για τα συνηθισμένα container-level resources του εργαστηρίου, ένα Pod χωρίς CPU και memory requests ή limits είναι BestEffort. Όταν όλα τα σχετικά containers έχουν ίσα requests και limits για CPU και μνήμη, πληροί τις προϋποθέσεις Guaranteed. Οι ενδιάμεσες περιπτώσεις είναι Burstable.

TERMINAL

```
kubectl get pods -l app=rails-map \
  -o 'custom-columns=NAME:.metadata.name,QOS:.status.qosClass'
```

Το web του βιβλίου είναι Burstable, αφού το request είναι μικρότερο από το limit. Η κατηγορία επηρεάζει τη μεταχείριση σε πίεση πόρων, αλλά δεν αποτελεί καθολική σειρά όπου κάθε Guaranteed Pod επιβιώνει και κάθε άλλο χάνεται. Η πίεση στο node, η υπέρβαση requests, οι προτεραιότητες και ο συγκεκριμένος πόρος έχουν σημασία. Δες τα Pod QoS classes και το Node-pressure eviction.

Η άσκηση είναι να συγκρίνεις τρεις περιπτώσεις. Ένα Pod Pending χωρίς node, ένα web που αργεί χωρίς restarts και ένα container με `OOMKilled`. Για καθεμία διάλεξε πρώτα την παρατήρηση που διακρίνει την αιτία. Μόνο μετά πρότεινε αλλαγή σε request ή limit. Το επόμενο κεφάλαιο θα προσθέσει τους περιορισμούς του ίδιου του node.

Λύση της άσκησης. Για Pending χωρίς node αρχίζεις από `FailedScheduling` και τα requests. Για αργό web με μηδενικά restarts συγκρίνεις CPU χρήση, όριο και latency, χωρίς να συμπεραίνεις throttling μόνο από τη βραδύτητα. Για `OOMKilled` ελέγχεις την κατανάλωση μνήμης και το όριο της συγκεκριμένης εκτέλεσης. Η αυθαίρετη αύξηση όλων των limits δεν διακρίνει τις αιτίες.

ΚΕΦΑΛΑΙΟ 17

Τοποθέτηση και απώλεια node

Δύο replicas στον ίδιο node δεν είναι δύο ανεξάρτητες θέσεις επιβίωσης.

Το πρώτο κεφάλαιο υποσχέθηκε αναπλήρωση αντιγράφων σε διαθέσιμα nodes. Εδώ θα τη δούμε. Θα σταματήσουμε έναν worker του kind, θα κρατήσουμε αιτήματα προς το Service και θα περιμένουμε να αντικατασταθεί το χαμένο web αλλού. Η αναμονή είναι μέρος του μαθήματος.

Χρησιμοποιούμε ξεχωριστό cluster για αυτό το πείραμα. Το αρχικό, με Postgres και Redis, παραμένει στη θέση του. Το μεγαλύτερο έχει ένα control plane και τρεις workers. Τα δύο web replicas τρέχουν χωρίς βάση, ώστε η τοπική αποθήκευση να μη γίνει δεύτερη αιτία αποτυχίας.

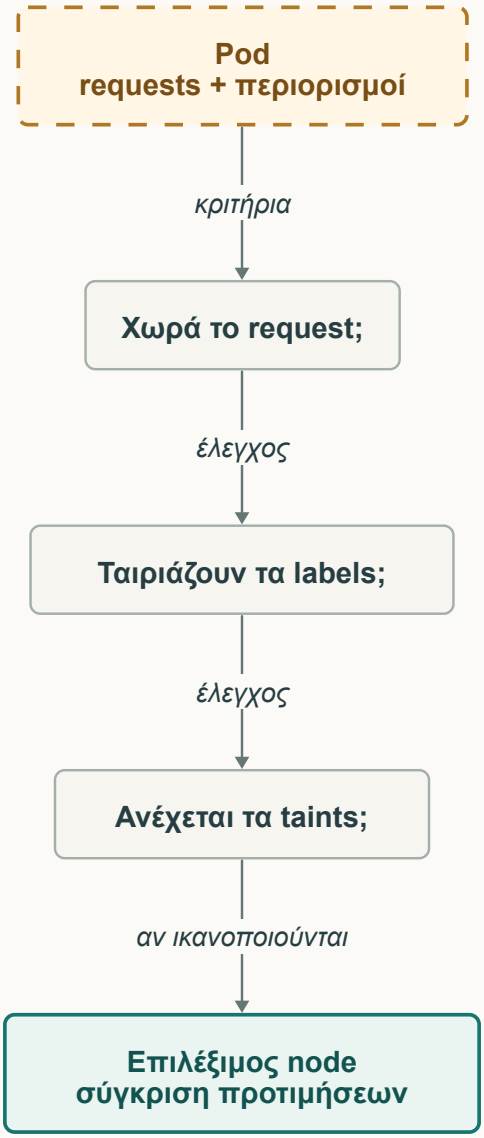
TERMINAL

```
bash scripts/cluster-ha-up.sh
export KUBECONFIG="$PWD/build/kubeconfig-ha"
kubectl config current-context
kubectl get nodes
kubectl get pods -o wide
```

Το context πρέπει να είναι `kind-k8s-book-ha`. Το όνομα του script αναφέρεται στο πείραμα διαθεσιμότητας της εφαρμογής. Το μοναδικό control plane δεν είναι highly available. Επιπλέον, όλα τα kind nodes μοιράζονται το ίδιο Mac και το ίδιο Docker Desktop. Δεν προσομοιώνουμε ανεξάρτητα φυσικά μηχανήματα σε διαφορετικές ζώνες.

> **Από το Kamal στο Kubernetes**
> Όταν διαλέγεις δύο servers, η φυσική κατανομή είναι μέρος της δικής σου επιλογής. Στο cluster πρέπει να τη δηλώσεις. Το πλήθος replicas και οι κανόνες τοποθέτησης απαντούν σε διαφορετικές ερωτήσεις.

Πού επιτρέπεται να πάει το Pod



Σχήμα 17.1 · Ο scheduler ελέγχει πρώτα αν υπάρχει επιτρεπτή θέση. Τα requests, τα labels και τα taints μπορούν να αποκλείσουν nodes πριν συγκριθούν οι υπόλοιπες προτιμήσεις.

Το `nodeSelector` ζητά nodes με συγκεκριμένα labels. Το node affinity εκφράζει πιο σύνθετες υποχρεωτικές συνθήκες ή προτιμήσεις. Ένα `required` που δεν ικανοποιείται αφήνει το Pod Pending. Ένα `preferred` δίνει κατεύθυνση χωρίς να αποτελεί απόλυτη απαγόρευση.

Τα taints λειτουργούν από την πλευρά του node. Μια toleration επιτρέπει σε ένα Pod να ανεχτεί το αντίστοιχο taint. Δεν το στέλνει υποχρεωτικά εκεί και δεν ακυρώνει τα υπόλοιπα κριτήρια. Το `NoSchedule` εμποδίζει νέες τοποθετήσεις χωρίς κατάλληλη toleration. Το `NoExecute` μπορεί να απομακρύνει και Pods που βρίσκονται ήδη εκεί.

Στο εργαστήριο οι workers έχουν `book-pool=workers`. Το web ζητά αυτό το label. Ο παρατηρητής `book-client` τοποθετείται ρητά στο control plane μόνο ως εργαλείο του πειράματος, για να μη σταματήσει μαζί με τον worker που θα χάσουμε. Η άμεση ανάθεση `nodeName` δεν είναι η κανονική στρατηγική τοποθέτησης της εφαρμογής.

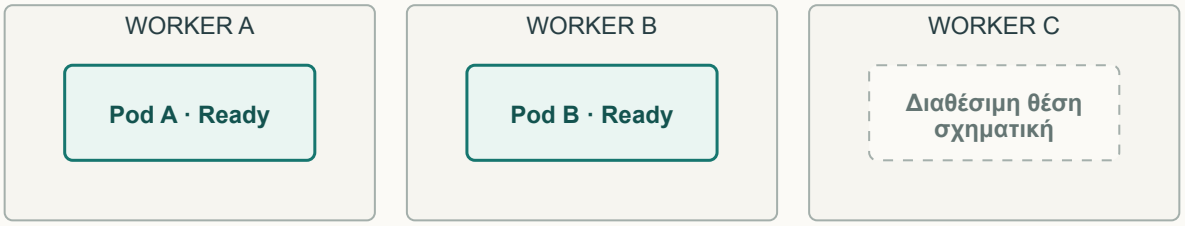
Οι κανόνες ορίζονται στα Assigning Pods to Nodes και Taints and Tolerations.

Δύο replicas σε διαφορετικά workers

manifests/ch17/placement-patch.yaml

```
spec:
  template:
    spec:
      nodeSelector:
        book-pool: workers
      topologySpreadConstraints:
        - maxSkew: 1
          topologyKey: kubernetes.io/hostname
          whenUnsatisfiable: DoNotSchedule
          nodeAffinityPolicy: Honor
          nodeTaintsPolicy: Honor
          labelSelector:
            matchLabels:
              app: rails-map
```

Το topology key χωρίζει τα Pods ανά hostname node. Το `maxSkew: 1` περιορίζει την απόκλιση πλήθους ανάμεσα στα επιλέξιμα domains. Το `DoNotSchedule` κάνει τον περιορισμό υποχρεωτικό για νέες τοποθετήσεις. Δεν μετακινεί αναδρομικά κάθε υπάρχον Pod μέχρι να πετύχει ένα ιδανικό σχέδιο.



Σχήμα 17.2 · Αρχική κατανομή. Δύο web σε διαφορετικά workers και τρίτος worker διαθέσιμος για αναπλήρωση. Το control plane παραμένει έξω από το σύνολο τοποθέτησης του web.

Τα `nodeAffinityPolicy` και `nodeTaintsPolicy` ορίζουν πώς υπολογίζονται οι επιλέξιμοι κόμβοι για τη διασπορά. Εδώ τιμούμε και τους δύο περιορισμούς. Ένα domain που δεν πρέπει να δεχτεί το Pod δεν αντιμετωπίζεται αφελώς ως χρήσιμη κενή θέση. Η λεπτομέρεια έχει σημασία όταν εμφανιστούν taints στον χαμένο node. Δες τα Pod Topology Spread Constraints.

Πριν προκαλέσεις βλάβη, επιβεβαίωσε την πραγματική κατανομή με `-o wide`. Η ύπαρξη του YAML δεν αποδεικνύει ότι τα δύο έτοιμα Pods που κοιτάς ανήκουν ήδη στο νέο template.

Σταμάτησε ένα συγκεκριμένο worker

Ο πλήρης αυτοματοποιημένος έλεγχος βρίσκεται στο παρακάτω script. Ελέγχει context, τοπικό API, project ownership και το Docker label του ακριβούς worker πριν τον σταματήσει. Στο τέλος τον ξεκινά ξανά.

TERMINAL

```
python3 scripts/check-node-loss.py
```

Για να παρακολουθήσεις το πείραμα, άνοιξε άλλα δύο terminals με το ίδιο kubeconfig.

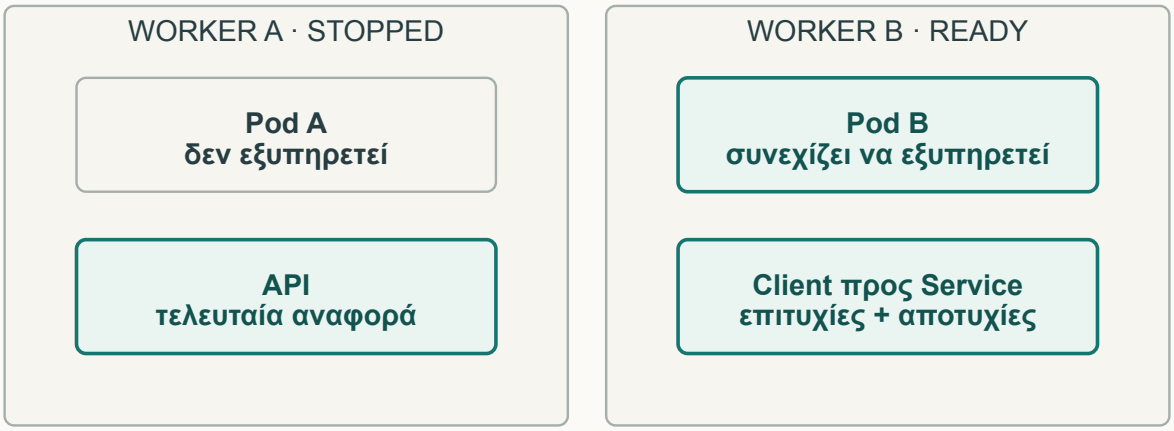
TERMINAL

```
kubectl get nodes -w
```

TERMINAL

```
kubectl get pods -l app=rails-map -o wide -w
```

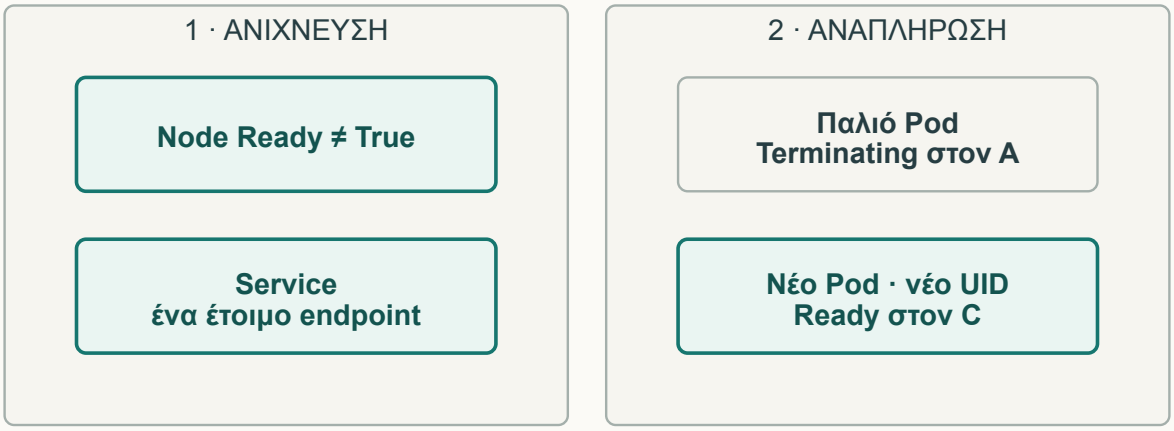
Η κρίσιμη πράξη μέσα στο script είναι `docker stop --timeout 1` στο node container που φιλοξενεί ένα web. Δεν διαγράφουμε απλώς το Pod. Σταματά η τοπική μονάδα που περιέχει kubelet, runtime και containers του node.



Σχήμα 17.3 · Ο worker σταματά. Για ένα διάστημα η τελευταία αναφορά στο API μπορεί ακόμη να φαίνεται έτοιμη. Ο client συνεχίζει να στέλνει αιτήματα.

Ο client του πειράματος καλεί το Service και αποθηκεύει HTTP κωδικούς ή σφάλματα. Η αποτυχία ενός node δεν ενημερώνει ακαριαία κάθε τμήμα του cluster. Μπορούν να χαθούν αιτήματα πριν αφαιρεθεί ο προορισμός από τη συνήθη δρομολόγηση. Η παρουσία δεύτερου έτοιμου replica μειώνει την έκταση της απώλειας, αλλά δεν μηδενίζει κάθε χρονικό παράθυρο.

Τα ρολόγια της αναπλήρωσης



Σχήμα 17.4 · Πρώτα αλλάζει η παρατήρηση του node και η δρομολόγηση. Αργότερα λήγει η ανοχή της εφαρμογής και δημιουργείται αντικαταστάτης σε διαθέσιμο worker.

Τα συνηθισμένα Pods λαμβάνουν tolerations για `not-ready` και `unreachable` με διάρκεια 300 δευτερολέπτων, αν δεν έχουν δηλωθεί άλλες κατάλληλες τιμές. Αυτό εξηγεί γιατί μπορεί να περιμένεις αρκετά λεπτά πριν την απομάκρυνση από έναν μη προσβάσιμο node. Η ανίχνευση του node και η εκκίνηση του νέου container προσθέτουν τον δικό τους χρόνο.

Δεν μειώνουμε αυτά τα νούμερα για να φαίνεται εντυπωσιακότερο το βιβλίο. Μετράμε πότε ο node έπαψε να είναι Ready, πότε μειώθηκαν τα ready endpoints και πότε ο αντικαταστάτης έγινε έτοιμος. Οι πραγματικές τιμές της εκτέλεσης βρίσκονται στο παράρτημα επαλήθευσης και στο `verification/evidence/ch17.json`.

Το παλιό Pod μπορεί να εμφανίζεται Terminating, επειδή το kubelet που θα ολοκλήρωνε τον τερματισμό είναι απρόσιτο. Δεν έχει «μετακομίσει» στο νέο node. Το καινούριο Pod έχει άλλο UID και διαφορετικό `nodeName`. Η επιθυμητή κατάσταση διατηρήθηκε και ένας controller δημιούργησε νέο αντικείμενο.

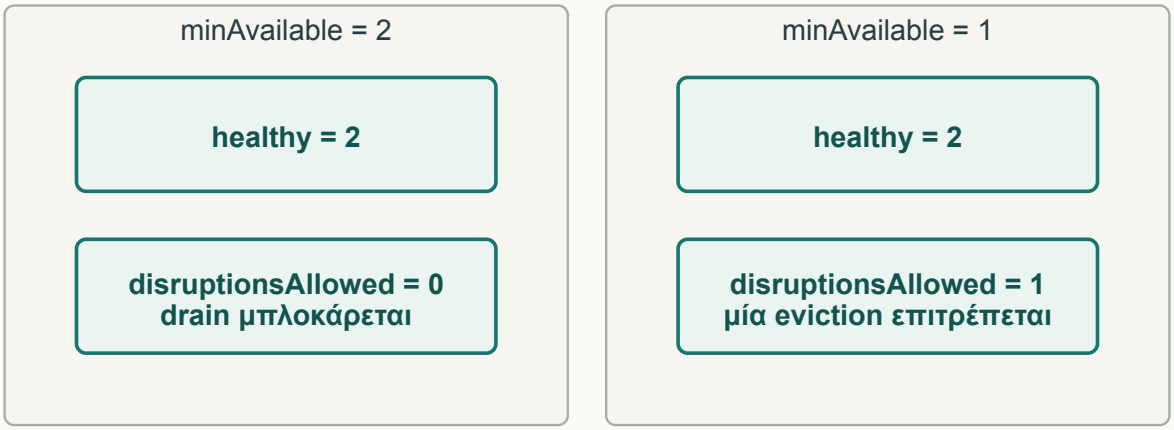
Το Service δεν περιμένει αναγκαστικά να ολοκληρωθεί αυτή η αντικατάσταση για να συνεχίσει να χρησιμοποιεί το άλλο έτοιμο endpoint. Η αφαίρεση χαμένου προορισμού και η αποκατάσταση του πλήθους replicas είναι διαφορετικοί βρόχοι.

Συντήρηση με PodDisruptionBudget

Η απώλεια node ήταν ακούσια από την οπτική του cluster. Τώρα θα ζητήσουμε ελεγχόμενη απομάκρυνση Pods με `drain`. Το PDB του πειράματος αρχικά απαιτεί δύο διαθέσιμα web.

manifests/ch17/pdb.yaml

```
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  name: rails-map
  namespace: k8s-book
spec:
  minAvailable: 2
  selector:
    matchLabels:
      app: rails-map
```



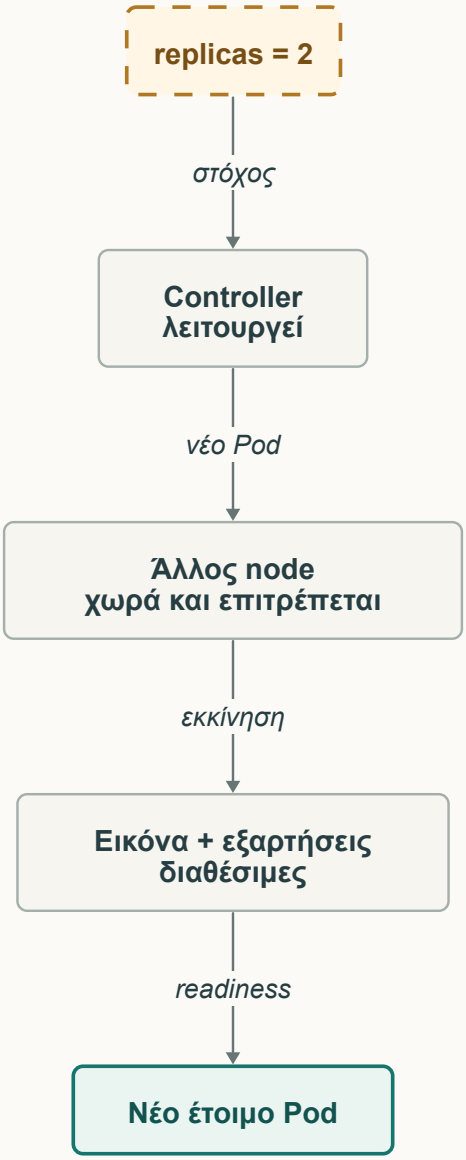
Σχήμα 17.5 · Με δύο healthy replicas και minAvailable δύο, δεν επιτρέπεται ελεγχόμενη eviction. Με minAvailable ένα, μπορεί να φύγει ένα Pod και να αναπληρωθεί αλλού.

Μετά την επαναφορά του χαμένου node, το script επιλέγει worker που φιλοξενεί τώρα ένα web. Εκτελεί `kubectl drain` με timeout και `--ignore-daemonsets`. Η πρώτη προσπάθεια πρέπει να μπλοκαριστεί από το PDB. Έπειτα αλλάζει `minAvailable` σε ένα και επαναλαμβάνει το drain.

Το drain αρχικά κάνει τον node unschedulable. Το PDB περιορίζει τις evictions μέσω του αντίστοιχου API. Δεν εμποδίζει τη φυσική απώλεια του node και δεν είναι ο μηχανισμός που επιβάλλει το `maxUnavailable` του Deployment. Μια άμεση διαγραφή Pod επίσης δεν προστατεύεται με τον ίδιο τρόπο από PDB.

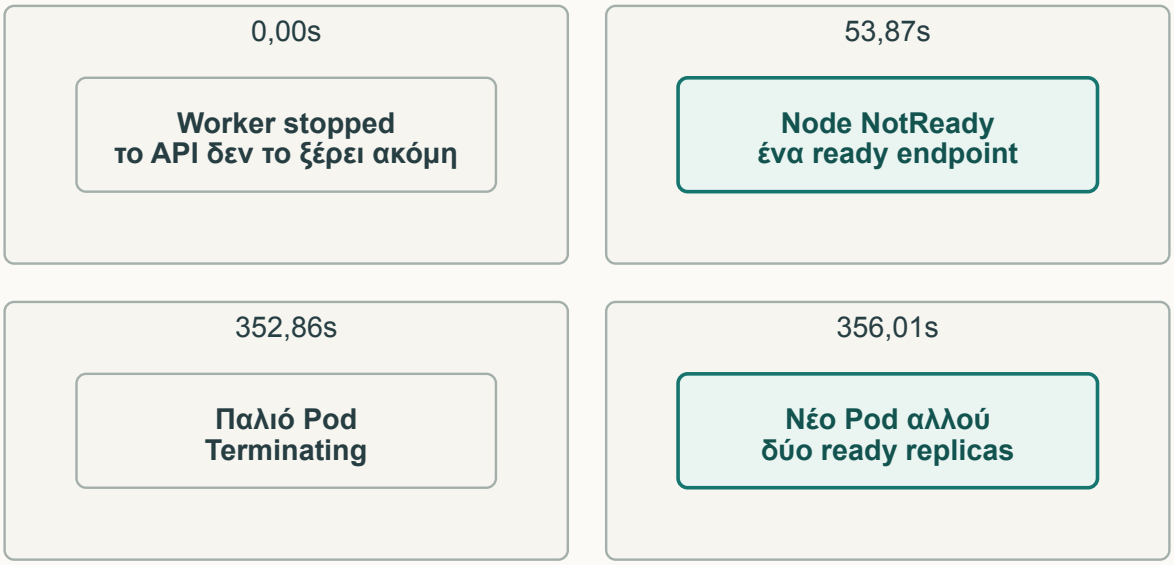
Όταν ολοκληρωθεί η συντήρηση χρειάζεται `uncordon`, ώστε ο node να επιτρέπεται ξανά για νέες τοποθετήσεις. Αυτό δεν μεταφέρει αυτόματα τα Pods πίσω. Η σημασιολογία των ελεγχόμενων και ακούσιων διακοπών περιγράφεται στα Disruptions.

Η απόδειξη και το όριό της



Σχήμα 17.6 · Η αναπλήρωση χρειάζεται controller που συνεχίζει να λειτουργεί, επιτρεπτό node, διαθέσιμους πόρους και εικόνα που μπορεί να ξεκινήσει. Το αντίγραφο δεν προκύπτει μόνο από τον αριθμό replicas.

Η άσκηση είναι να εξηγήσεις τι θα άλλαζε αν ο τρίτος worker δεν είχε χώρο ή αν το web απαιτούσε τοπικό PVC του χαμένου node. Το YAML μπορεί να συνεχίζει να ζητά δύο replicas ενώ το παρατηρημένο σύνολο παραμένει ένα. Το `Pending` τότε δεν είναι αποτυχία της ιδέας του controller. Είναι η ορατή συνέπεια ανεκπλήρωτων προϋποθέσεων.



Σχήμα 17.7 · Τα τέσσερα γεγονότα της καταγεγραμμένης εκτέλεσης. Οι χρόνοι μετρούν από το docker stop και οι αποστάσεις των καρέ δεν παριστάνουν χρονική κλίμακα.

Σε αυτή την εκτέλεση ο client κατέγραψε 800 αιτήματα και 15 αποτυχίες. Το νέο replica έγινε έτοιμο σε περίπου έξι λεπτά. Η εφαρμογή είχε στο ενδιάμεσο το δεύτερο έτοιμο web, αλλά υπήρξε και το παράθυρο μέχρι να αφαιρεθεί ο χαμένος προορισμός. Κράτησε και τα δύο αποτελέσματα όταν εξηγείς τι προσφέρει η αναπλήρωση.

Πριν επιστρέψεις στην εφαρμογή με βάση και worker, άλλαξε ξανά το kubeconfig.

TERMINAL

```
export KUBECONFIG="$PWD/build/kubeconfig"
kubectl config current-context
```

Το όνομα του Service είναι ίδιο στα δύο clusters. Το context καθορίζει σε ποια πραγματικότητα αναφέρεται κάθε επόμενη εντολή.

ΚΕΦΑΛΑΙΟ 18

Μετρήσεις και HPA

Το autoscaling είναι άλλος ένας controller. Χρειάζεται μετρήσεις, στόχο και χώρο για να εκτελέσει την απόφασή του.

Μέχρι εδώ αλλάζαμε τα replicas μόνοι μας. Τώρα θα δώσουμε αυτή τη δουλειά σε HorizontalPodAutoscaler. Το παράδειγμα χρησιμοποιεί CPU utilization και δύο έως τέσσερα web replicas. Θα δημιουργήσουμε φορτίο, θα παρατηρήσουμε αύξηση και μετά θα περιμένουμε τη μείωση.

Εργαζόμαστε ξανά στο μικρό cluster με βάση και worker. Έλεγξε το context πριν συνεχίσεις.

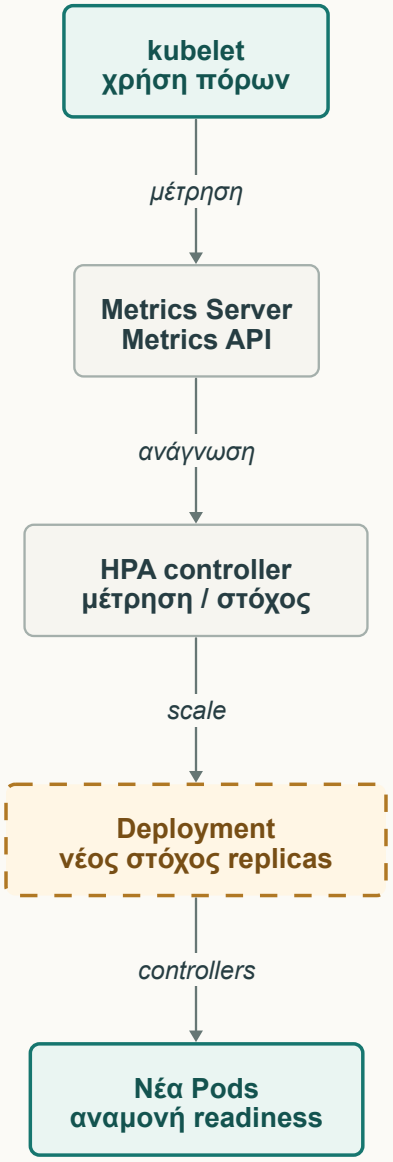
TERMINAL

```
export KUBECONFIG="$PWD/build/kubeconfig"
bash scripts/cluster-up.sh --with-metrics
kubectl top pods
```

Το setup εγκαθιστά Metrics Server 0.9.0, συμβατό με το Kubernetes 1.34 του εργαστηρίου. Η τοπική διαμόρφωση παρακάμπτει την επαλήθευση του kubelet serving certificate με `--kubelet-insecure-tls`, επειδή το συγκεκριμένο kind setup δεν παρέχει την αλυσίδα πιστοποιητικών που απαιτείται. Είναι επιλογή αυτού του εργαστηρίου, όχι γενική διαμόρφωση παραγωγής. Η εγκατάσταση και οι απαιτήσεις περιγράφονται στο Metrics Server.

> **Από το Kamal στο Kubernetes**
> Η χειροκίνητη αύξηση workers ή web αλλάζει έναν αριθμό που διάλεξες εσύ. Ο HPA εισάγει controller που ξαναϋπολογίζει αυτόν τον στόχο από μετρήσεις. Χρειάζεται σαφής συμφωνία για το ποιος γράφει το replicas.

Από τη μέτρηση στην απόφαση



Σχήμα 18.1 · Το kubelet δίνει μετρήσεις στον Metrics Server. Ο HPA τις διαβάζει από το Metrics API και αλλάζει τον στόχο replicas του Deployment. Οι υπόλοιποι controllers εκτελούν τη μεταβολή.

Το Metrics Server δεν είναι πλήρες σύστημα ιστορικών μετρήσεων και alerts. Καλύπτει πρόσφατα resource metrics για λειτουργίες όπως το `top` και το autoscaling. Η απουσία ιστορικού πολλών ημερών δεν διορθώνεται αυξάνοντας τα replicas του.

Η CPU utilization υπολογίζεται σε σχέση με το CPU request. Αν ένα container ζητά `100m` και χρησιμοποιεί `60m`, η χρήση είναι 60% του request. Δεν είναι 60% ολόκληρου του node και δεν είναι 60% του limit `1` που βάλαμε στο web.

Στην απλή περίπτωση, δύο replicas με μέση χρήση 120% και στόχο 60% υποδεικνύουν τέσσερα replicas. Ο βασικός λόγος είναι δύο επί 120 δια 60, με στρογγυλοποίηση προς τα πάνω. Ο πραγματικός controller συνυπολογίζει ελλιπείς μετρήσεις, Pods που δεν είναι ακόμη έτοιμα, ανοχή και όρια μεταβολής. Το παράδειγμα αριθμητικής εξηγεί την κατεύθυνση, όχι κάθε απόφαση του αλγορίθμου.

Ποιος γράφει τα replicas τώρα

Πριν ενεργοποιήσουμε HPA, προετοιμάζουμε το web με τα probes του προηγούμενου μέρους. Η έκδοση `manifests/ch18/web.yaml` κρατά την ίδια εφαρμογή και παραλείπει το `spec.replicas`.

TERMINAL

```
kubectl apply -f manifests/ch15/web.yaml
kubectl rollout status deployment/rails-map --timeout=180s
kubectl apply --server-side --field-manager=book-hpa-app \
  -f manifests/ch18/web.yaml
kubectl apply -f manifests/ch18/hpa.yaml
```

Χρησιμοποιούμε νέο server-side field manager που δεν διεκδικεί το πεδίο replicas. Η παράλειψη δεν στέλνει νέο αριθμό και ο HPA θα το διαχειρίζεται μέσω του scale subresource. Από εδώ και πέρα δεν εφαρμόζουμε παλιό manifest που ξαναδηλώνει σταθερό πλήθος, όσο θέλουμε να παραμένει ενεργό το autoscaling.

Αν διαγράψεις απλώς το `replicas` από αρχείο που μέχρι τώρα διαχειριζόταν το ίδιο πεδίο με client-side apply, η επόμενη εφαρμογή του μπορεί να έχει διαφορετικό αποτέλεσμα λόγω της προηγούμενης δήλωσης και των προεπιλογών. Αυτός είναι πρακτικός λόγος για την παράγραφο ιδιοκτησίας πεδίων του κεφαλαίου 7.

manifests/ch18/hpa.yaml

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: rails-map
  namespace: k8s-book
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: rails-map
  minReplicas: 2
  maxReplicas: 4
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 60
  behavior:
    scaleDown:
      stabilizationWindowSeconds: 30
```

Το παράδειγμα συντομεύει το παράθυρο σταθεροποίησης μείωσης στα 30 δευτερόλεπτα για παρατήρηση στο εργαστήριο. Η προεπιλεγμένη συμπεριφορά μπορεί να περιμένει περισσότερο. Ένας σύντομος θόρυβος στη μέτρηση δεν πρέπει να κάνει τα Pods να προστίθενται και να αφαιρούνται αδιάκοπα.

Βάλε φορτίο και περίμενε

TERMINAL

```
kubectl get hpa rails-map -w
```

Σε δεύτερο terminal ξεκίνησε τον μικρό client φορτίου. Διαρκεί 90 δευτερόλεπτα και σταματά μόνος του.

TERMINAL

```
kubectl exec -i book-client -- \
  env BOOK_LOAD_SECONDS=90 ruby < scripts/load.rb
```

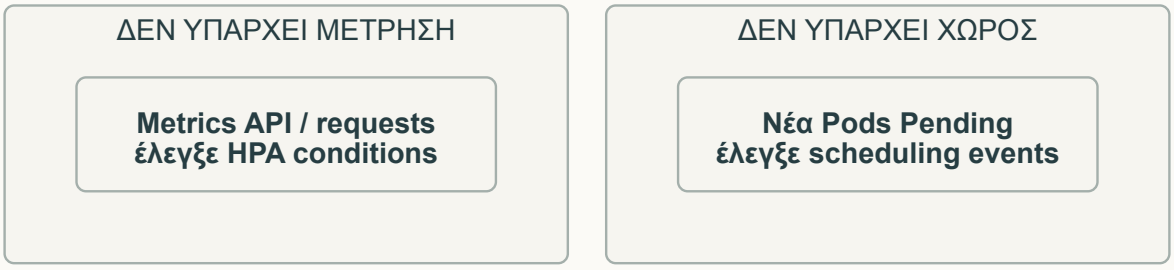
Σχήμα 18.2 · Η αύξηση φορτίου προηγείται της μέτρησης, η μέτρηση της αλλαγής replicas και η αλλαγή της ετοιμότητας των νέων Pods. Καμία από αυτές τις μεταβάσεις δεν είναι ακαριαία.

Το `/work` κάνει περιορισμένη δουλειά CPU. Δεν γράφει νέα δεδομένα και δεν ζητά εξωτερικές υπηρεσίες. Ο στόχος είναι να παρατηρήσουμε τη σχέση CPU, HPA και Deployment. Το εργαλείο καταγράφει HTTP κωδικούς και σφάλματα, αλλά δεν αποτελεί πλήρες benchmark Rails.

Όταν σταματήσει το φορτίο, μη μειώσεις μόνος σου τα replicas. Παρατήρησε το παράθυρο σταθεροποίησης και την επιστροφή στον ελάχιστο αριθμό. Αν έκανες `kubectl scale` στο ενδιάμεσο, θα πρόσθετες άλλον writer και θα έχανες την καθαρή εξήγηση του αποτελέσματος.

Η ακολουθία και οι παράμετροι τεκμηριώνονται στο Horizontal Pod Autoscaling.

Όταν ο HPA δεν κινείται



Σχήμα 18.3 · Μη διαθέσιμη μέτρηση, απουσία request και έλλειψη χώρου είναι διαφορετικά εμπόδια. Το τελευταίο μπορεί να εμφανιστεί αφού ο HPA έχει ήδη αυξήσει τον επιθυμητό αριθμό.

TERMINAL

```
kubectl describe hpa rails-map
kubectl get apiservice v1beta1.metrics.k8s.io
kubectl get pods -l app=rails-map -o wide
```

Αν το target φαίνεται unknown, έλεγξε πρώτα τη διαθεσιμότητα των metrics και τα requests των containers που συμμετέχουν στον υπολογισμό. Άγνωστη μέτρηση δεν σημαίνει μηδενική CPU. Η τυφλή αύξηση του `maxReplicas` δεν λύνει την έλλειψη δεδομένων.

Αν ο HPA ζητά τέσσερα replicas αλλά δύο νέα Pods μένουν Pending, ο controller autoscaling μπορεί να έχει κάνει σωστά τη δουλειά του. Ο scheduler δεν βρήκε χώρο. Ο HPA προσθέτει επιθυμητά Pods, όχι νέους nodes. Για αύξηση node capacity χρειάζεται διαφορετικός μηχανισμός και αντίστοιχη υποδομή.

Η άσκηση είναι να εξηγήσεις γιατί το ίδιο web με request `50m` θα έδειχνε διπλάσια utilization από ό,τι με request `100m`, για την ίδια πραγματική CPU χρήση. Έπειτα εξήγησε γιατί αυτό δεν σημαίνει ότι διπλασιάστηκαν τα αιτήματα των χρηστών. Ο αριθμός που διάλεξες στο request είναι μέρος του μοντέλου ελέγχου.

Λύση της άσκησης. Χρήση `40m` με request `100m` δίνει 40% utilization. Η ίδια χρήση με request `50m` δίνει 80%. Άλλαξε ο παρονομαστής του υπολογισμού, χωρίς να έχει αλλάξει υποχρεωτικά το εισερχόμενο φορτίο. Ο HPA μπορεί έτσι να πάρει διαφορετική απόφαση για την ίδια μετρημένη CPU.

ΚΕΦΑΛΑΙΟ 19

Namespaces και δικαιώματα

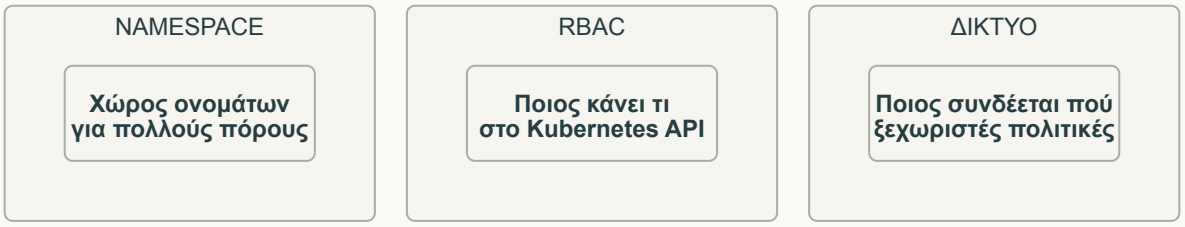
Το namespace απαντά πού βρίσκεται το αντικείμενο. Το RBAC απαντά ποιος μπορεί να κάνει τι σε αυτό.

Μέχρι τώρα το ιδιωτικό kubeconfig του εργαστηρίου είχε δικαιώματα διαχειριστή στο συγκεκριμένο kind cluster. Αυτό βολεύει την κατασκευή του εργαστηρίου. Δεν περιγράφει τα δικαιώματα που χρειάζεται μια εφαρμογή για να απαντήσει HTTP ή να συνδεθεί σε Postgres.

Θα δημιουργήσουμε ταυτότητα που μπορεί να διαβάζει Pods μόνο στο namespace `k8s-book`. Θα επιβεβαιώσουμε ότι δεν μπορεί να διαγράφει Pods, να διαβάζει Secrets ή να διαβάζει Pods άλλου namespace. Οι έλεγχοι εξουσιοδότησης δεν απαιτούν να εκτελέσουμε τις απαγορευμένες αλλαγές.

> **Από το Kamal στο Kubernetes**
> Η πρόσβαση στον server και η πρόσβαση στην εφαρμογή ήταν ήδη διαφορετικά δικαιώματα. Το RBAC αφορά συγκεκριμένες πράξεις στο Kubernetes API. Δεν είναι ο κωδικός της βάσης ούτε πολιτική δικτυακής επικοινωνίας.

Διαφορετικά όρια



Σχήμα 19.1 · Τα namespaces χωρίζουν ονόματα και πεδία εφαρμογής πολλών αντικειμένων. Η εξουσιοδότηση API και η δικτυακή πρόσβαση παραμένουν ξεχωριστοί μηχανισμοί.

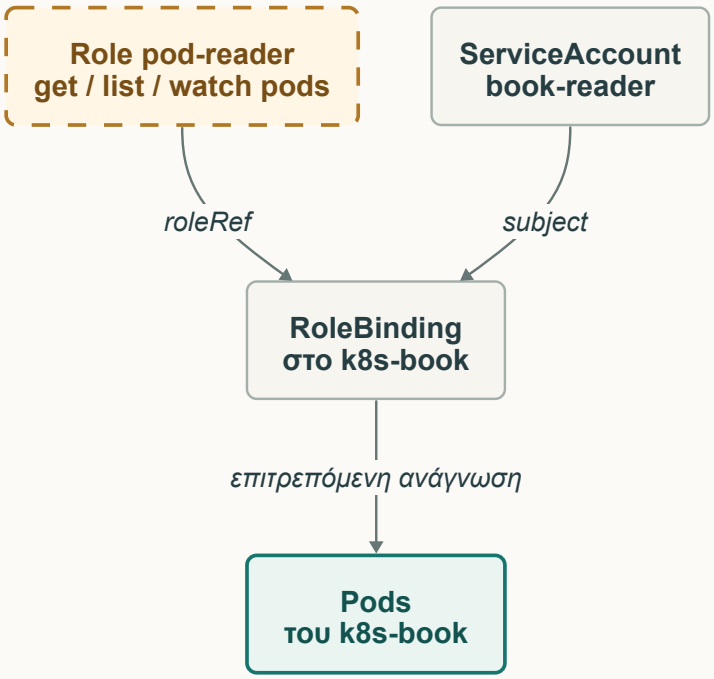
Μπορούν να υπάρχουν δύο Services με το ίδιο όνομα σε διαφορετικά namespaces. Ένα Role ισχύει μέσα στο namespace του. Ένα Node και ένα PersistentVolume είναι αντικείμενα επιπέδου cluster. Δεν αποκτούν namespace επειδή τα κοιτάς με `kubectl -n ...`.

Το namespace δεν δημιουργεί μόνο του firewall. Το να βάλεις Postgres σε άλλο namespace δεν σημαίνει ότι άλλα Pods δεν μπορούν να συνδεθούν στη θύρα του. Οι NetworkPolicies είναι διαφορετικό API και απαιτούν υλοποίηση δικτύωσης που τις εφαρμόζει. Το kind setup αυτού του βιβλίου δεν χρησιμοποιείται ως απόδειξη δικτυακής απομόνωσης.

Αντίστοιχα, το ότι ένας χρήστης μπορεί να δει ένα Service μέσω του API δεν είναι το ίδιο δικαίωμα με το να στείλει HTTP προς την εφαρμογή. Κράτησε χωριστά τα αιτήματα διαχείρισης και τα αιτήματα χρηστών. Οι χώροι ονομάτων εξηγούνται στα Namespaces.

Ταυτότητα, κανόνες και σύνδεση

Το ServiceAccount είναι ταυτότητα για διεργασίες που αλληλεπιδρούν με το Kubernetes API. Δεν είναι Postgres user, Linux user ή Rails account. Το Role περιγράφει επιτρεπόμενες ενέργειες. Το RoleBinding συνδέει την ταυτότητα με τους κανόνες.



Σχήμα 19.2 · Το RoleBinding απονέμει στον book-reader τους κανόνες του pod-reader μέσα στο k8s-book. Χωρίς τη σύνδεση, η ύπαρξη των δύο αντικειμένων δεν δίνει το δικαίωμα.

TERMINAL

```
kubectl apply -f manifests/ch19/rbac.yaml
```

Το κεντρικό τμήμα του Role είναι μικρό.

YAML

```
rules:
  - apiGroups: [""]
    resources: ["pods"]
    verbs: ["get", "list", "watch"]
```

Το κενό string επιλέγει το core API group. Το resource γράφεται στον πληθυντικό όπως εκτίθεται στο API. Το `get` αφορά ανάγνωση συγκεκριμένου αντικειμένου, το `list` κατάλογο και το `watch` παρακολούθηση αλλαγών.

Δεν δώσαμε `delete`, `create`, `patch` ή πρόσβαση σε `secrets`. Ούτε δώσαμε αυτόματα πρόσβαση στο υποresource `pods/log`. Αν μια συγκεκριμένη εργασία χρειάζεται logs, αυτή είναι ξεχωριστή απαίτηση που μπορεί να δηλωθεί στο Role.

Μέτρησε το δικαίωμα

TERMINAL

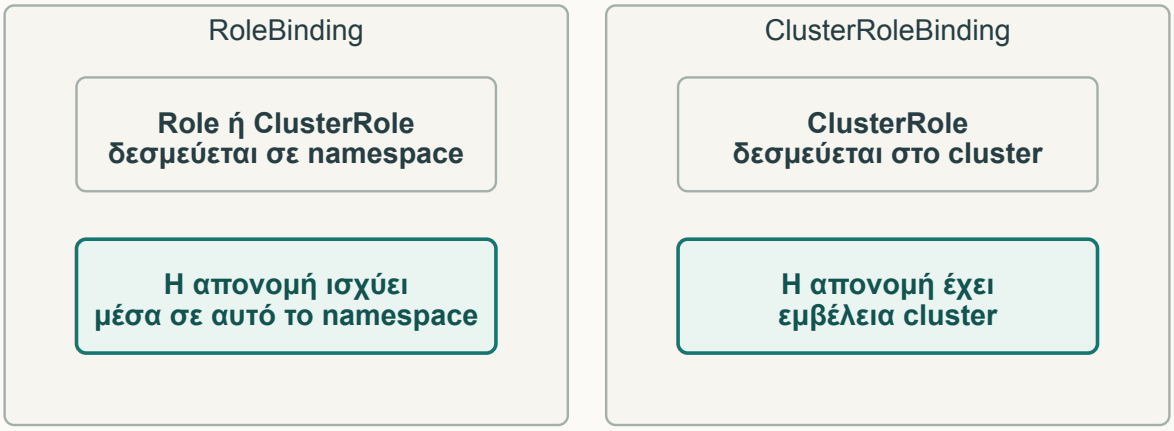
```
kubectl auth can-i get pods -n k8s-book \
  --as=system:serviceaccount:k8s-book:book-reader
kubectl auth can-i delete pods -n k8s-book \
  --as=system:serviceaccount:k8s-book:book-reader
kubectl auth can-i get secrets -n k8s-book \
  --as=system:serviceaccount:k8s-book:book-reader
kubectl auth can-i get pods -n default \
  --as=system:serviceaccount:k8s-book:book-reader
```

Περιμένουμε `yes`, `no`, `no`, `no`. Το `--as` εδώ χρησιμοποιεί την εξουσιοδοτημένη δυνατότητα impersonation του διαχειριστή του τοπικού cluster. Δεν επιτρέπει σε οποιονδήποτε χρήστη να γίνει άλλη ταυτότητα απλώς γράφοντας το όνομά της.

Η δυνατότητα που ελέγχουμε είναι αυτή του ServiceAccount. Δεν διαβάζουμε ή εμφανίζουμε token για να κάνουμε το πείραμα. Η μέθοδος κρατά το ερώτημα καθαρό και τα αποτελέσματα εύκολα συγκρίσιμα.

Role και ClusterRole

Ένα ClusterRole ορίζει κανόνες που μπορούν να χρησιμοποιηθούν σε επίπεδο cluster ή να επαναχρησιμοποιηθούν μέσα σε namespace μέσω RoleBinding. Ένα ClusterRoleBinding απονέμει τους αντίστοιχους κανόνες σε επίπεδο cluster. Η λέξη Cluster στο όνομα του ρόλου δεν καθορίζει μόνη της όλη την εμβέλεια μιας απονομής. Χρειάζεται να διαβάσεις και το binding.



Σχήμα 19.3 · Η ίδια ιδέα κανόνων μπορεί να συνδεθεί με διαφορετική εμβέλεια. Το είδος και το namespace του binding είναι μέρος της απάντησης.

Τα δικαιώματα RBAC προστίθενται. Ένας δεύτερος ρόλος που δεν αναφέρει Secrets δεν ακυρώνει πρόσβαση σε Secrets που έχει ήδη δοθεί από άλλο binding. Αν μια δοκιμή επιστρέφει περισσότερα δικαιώματα από όσα περιμένεις, αναζήτησε όλες τις σχετικές απονομές. Η απουσία μιας ενέργειας από έναν συγκεκριμένο Role δεν είναι κανόνας άρνησης που υπερισχύει των άλλων.

Η πλήρης σημασιολογία περιγράφεται στο Using RBAC Authorization.

Τι χρειάζεται η Rails εφαρμογή

Το web και ο worker του δείγματος δεν καλούν το Kubernetes API. Για αυτό το Pod spec τους έχει `automountServiceAccountToken: false`. Συνεχίζουν να χρησιμοποιούν DNS, Service και τις ρυθμίσεις που παρέχει το kubelet. Η σύνδεση στη βάση χρησιμοποιεί διαπιστευτήρια βάσης, όχι ServiceAccount token.

Αν αργότερα ένας controller ή εργασία πρέπει να διαβάζει το API, δίνεις συγκεκριμένο ServiceAccount και περιορισμένα δικαιώματα για τη δουλειά του. Δεν κληρονομεί σιωπηλά το kubeconfig του ανθρώπου που έκανε deploy. Η διαχείριση των ταυτοτήτων περιγράφεται στα Service Accounts.

Η άσκηση είναι να προσθέσεις μόνο πρόσβαση ανάγνωσης logs στον `pod-reader` και να ξαναελέγξεις `get pods/log`. Η διαγραφή Pod και η ανάγνωση Secret πρέπει να παραμείνουν απαγορευμένες. Το σωστό αποτέλεσμα είναι ένα επιπλέον συγκεκριμένο δικαίωμα, όχι μια γενική μετάβαση σε διαχειριστή.

Λύση και επαναφορά. Το `manifests/ch19/logs-role.yaml` προσθέτει μόνο το resource `pods/log` με verb `get`. Η αρχική ανάγνωση Pods παραμένει στην πρώτη rule.

TERMINAL

```
kubectl apply -f manifests/ch19/logs-role.yaml
kubectl auth can-i get pods/log -n k8s-book \
  --as=system:serviceaccount:k8s-book:book-reader
kubectl auth can-i delete pods -n k8s-book \
  --as=system:serviceaccount:k8s-book:book-reader
kubectl apply -f manifests/ch19/rbac.yaml
```

Οι δύο απαντήσεις είναι `yes` και `no`. Η τελευταία εντολή επαναφέρει το αρχικό Role. Στον έλεγχο της άσκησης επιβεβαιώθηκε επίσης ότι η ανάγνωση Secrets παρέμεινε απαγορευμένη.

ΚΕΦΑΛΑΙΟ 20

Ο μικρός άτλας βλαβών

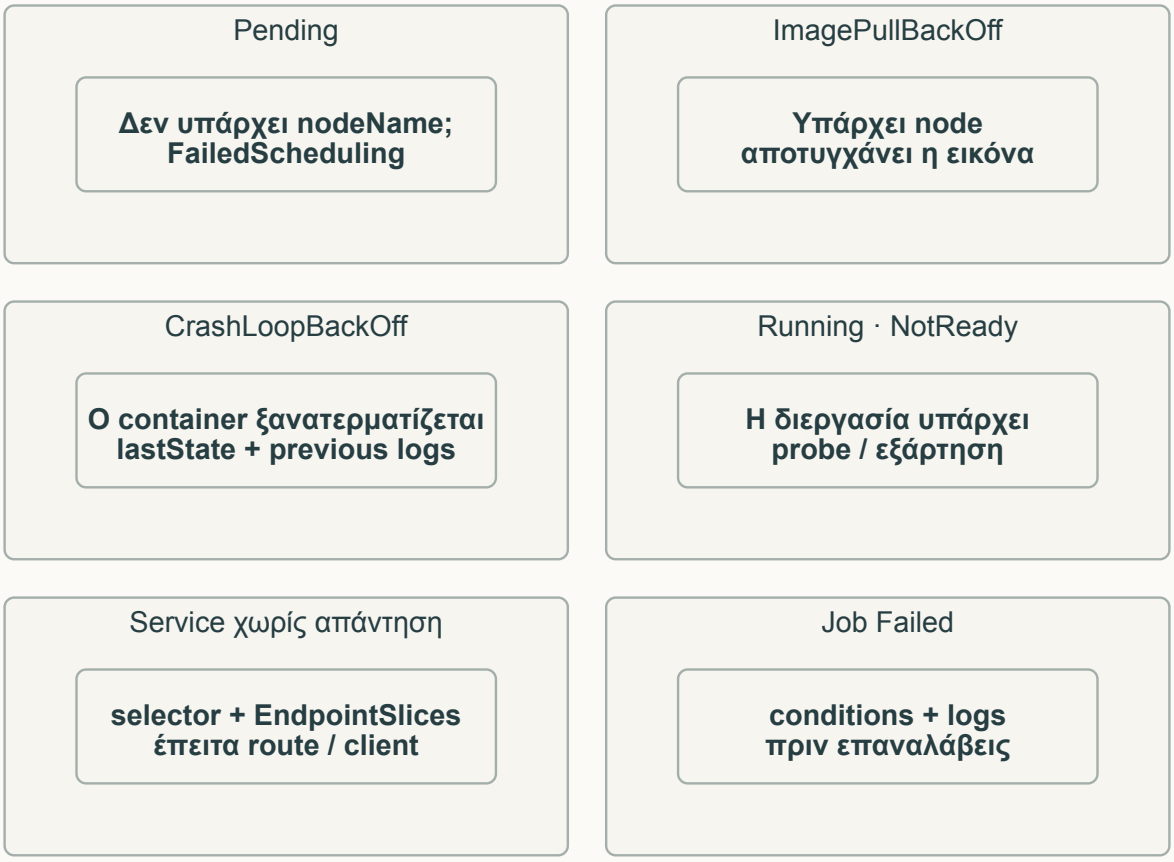
Η σύντομη ένδειξη είναι σημείο εκκίνησης. Η αιτία βρίσκεται ένα επίπεδο βαθύτερα.

Έχουμε προκαλέσει αρκετές βλάβες ώστε οι λέξεις του `kubectl get pods` να συνδέονται με πραγματικά γεγονότα. Αυτή η ενότητα είναι ο χάρτης επιστροφής. Δεν προσθέτει άλλον μηχανισμό στο cluster. Συγκεντρώνει τις διαφορές που χρειάζεται να θυμάσαι όταν πολλές γραμμές του terminal μοιάζουν μεταξύ τους.

Η στήλη `STATUS` του kubectl δεν είναι πάντα η τιμή του `status.phase` στο API. Ενδείξεις όπως `CrashLoopBackOff` και `ImagePullBackOff` περιγράφουν πιο συγκεκριμένες καταστάσεις containers που το εργαλείο προβάλλει συνοπτικά. Το `Terminating` επίσης δεν είναι ξεχωριστή Pod phase. Είναι ένδειξη ότι έχει ζητηθεί διαγραφή και ο τερματισμός εκκρεμεί.

> **Από το Kamal στο Kubernetes**
> Οι γνώριμες βλάβες εφαρμογής παραμένουν. Προστίθενται στάδια πριν από το boot και σχέσεις ανάμεσα σε αντικείμενα. Το όνομα της κατάστασης βοηθά μόνο όταν οδηγεί στο κατάλληλο στοιχείο.

Η αφίσα



Σχήμα 20.1 · Έξι είσοδοι στον ίδιο χάρτη. Κάθε σύμπτωμα οδηγεί σε διαφορετικό πεδίο παρατήρησης.

Αυτό που βλέπεις	Πρώτη χρήσιμη διάκριση	Πού το δοκιμάσαμε
Pending	Υπάρχει nodeName;	Requests και τοποθέτηση, κεφάλαια 16–17
ImagePullBackOff	Γιατί απέτυχε το pull;	Ανύπαρκτη εικόνα, κεφάλαιο 6
CrashLoopBackOff	Τι τερμάτισε την προηγούμενη εκτέλεση;	Probe loop και OOM, κεφάλαια 15–16
Running με 0/1 ready	Αποτυγχάνει η ετοιμότητα ή ακόμη ξεκινά;	Readiness, κεφάλαια 9–10
Service χωρίς επιτυχία	Δεν έχει endpoints ή δεν είναι ready;	Τα τρία καρέ, κεφάλαιο 10
Job Failed	Ποιο Pod και ποια εκτέλεση απέτυχαν;	Migration, κεφάλαιο 14

Το Pending είναι ευρύτερο από «δεν χώρεσε στο node». Ένα Pod μπορεί να έχει ήδη ανατεθεί και να περιμένει εικόνα, mount ή άλλη προετοιμασία. Η ύπαρξη `nodeName` χωρίζει τις δύο διαδρομές. Για αυτό το `-o wide` συχνά δίνει πιο χρήσιμη πρώτη πληροφορία από μια βιαστική προσπάθεια ανάγνωσης logs.

Η εφαρμογή δεν ξεκίνησε

Στο ανύπαρκτο image, το API δέχτηκε τη δήλωση και ο scheduler βρήκε node. Η αποτυχία ήρθε στο runtime. Το event με την αποτυχημένη λήψη ήταν πιο χρήσιμο από το όνομα της συνοπτικής κατάστασης.

Αν το Pod περιμένει Secret ή ConfigMap που δεν υπάρχει, η εικόνα μπορεί να είναι απολύτως σωστή. Αν περιμένει volume, το θέμα μπορεί να βρίσκεται στο claim, στον provisioner ή στην τοποθέτηση. Η γενική φράση «δεν κάνει boot» καλύπτει πολλές διαφορετικές εργασίες πριν καν ξεκινήσει ο κώδικάς σου.

TERMINAL

```
kubectl get pods -o wide
kubectl describe pods -l app=rails-map
kubectl get events --sort-by=.metadata.creationTimestamp
```

Κοίτα το namespace και τον στόχο του event. Ένα προειδοποιητικό event από παλιό Pod ή από άλλο πείραμα δεν αποτελεί αυτόματα εξήγηση για την τρέχουσα αποτυχία. Χρόνος, όνομα και UID δίνουν το πλαίσιο.

Η εφαρμογή ξεκίνησε και έφυγε

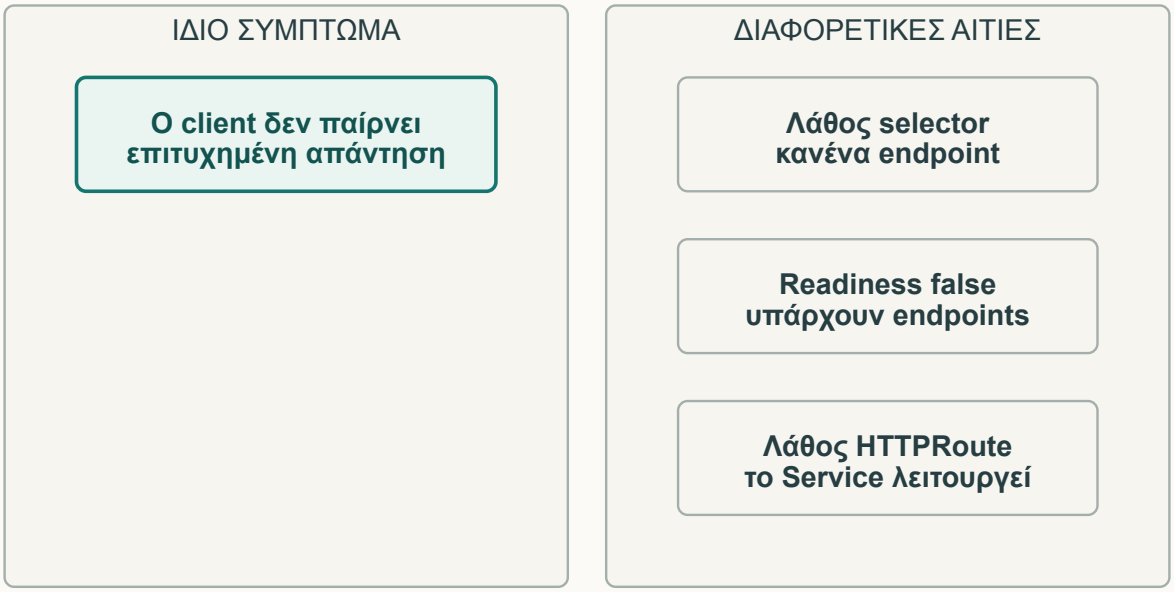
Το CrashLoopBackOff περιγράφει την καθυστέρηση ανάμεσα σε επαναλαμβανόμενες εκκινήσεις. Δεν εξηγεί γιατί τερματίζεται η διεργασία. Μπορεί να είναι σφάλμα κώδικα, λάθος εντολή, OOM ή τερματισμός από liveness probe.

Στο πείραμα του κεφαλαίου 15 υπήρχε event για αποτυχημένο liveness πριν από το restart. Στο κεφάλαιο 16 η προηγούμενη κατάσταση έγραφε `OOMKilled`. Το ίδιο γενικό σύμπτωμα απαιτούσε διαφορετική αλλαγή.

Πρόσεξε και την περίπτωση exit code μηδέν. Μια διεργασία που ολοκληρώνει επιτυχώς και τερματίζει μπορεί να ξαναξεκινά όταν ανήκει σε workload με restart policy Always. Αν ήθελες εργασία που ολοκληρώνεται, ίσως χρειάζεσαι Job αντί για Deployment. Η επιτυχία της διεργασίας και η καταλληλότητα του controller είναι διαφορετικά ερωτήματα.

Οι καταστάσεις και οι μεταβάσεις περιγράφονται στο Pod lifecycle.

Η εφαρμογή τρέχει, ο client αποτυγχάνει



Σχήμα 20.2 · Το ίδιο εξωτερικό σύμπτωμα μπορεί να προέρχεται από επιλογή Pods, ετοιμότητα ή HTTP route. Η διόρθωση ακολουθεί την πρώτη λανθασμένη σχέση στον χάρτη.

Ένα `Running` web δεν αποδεικνύει ότι ο client χρησιμοποιεί το σωστό hostname. Ένα επιτυχημένο port forward προς Pod δεν ελέγχει τον selector του Service. Ένα επιτυχημένο Service μέσα στο cluster δεν επαληθεύει το TLS του Gateway.

Στο εργαστήριο ο λάθος host επέστρεψε διαφορετική απάντηση από το route με ανύπαρκτο backend. Η πρακτική αξία βρίσκεται στη σύνδεση με τα conditions. Στο πρώτο δεν ταίριαζε η HTTP επιλογή. Στο δεύτερο το route ανέφερε μη επιλύσιμη αναφορά backend.

Μην θεραπεύεις ένα λάθος route με restart του Postgres. Μπορεί να δημιουργήσεις πραγματική διακοπή σε λειτουργικό εξάρτημα χωρίς να αλλάξεις την αρχική αιτία. Η διάγνωση Services ακολουθεί αυτή τη διαδρομή από εφαρμογή, selectors και endpoints προς τη δικτύωση.

Η άσκηση της αφίσας

Διάλεξε ένα από τα πειράματα που έχουν ήδη περάσει. Πριν το προκαλέσεις, γράψε το πρώτο πεδίο που περιμένεις να αλλάξει και ένα πεδίο που περιμένεις να μείνει ίδιο.

Για λάθος selector Service, τα UIDs των web πρέπει να μείνουν ίδια και τα endpoints να αλλάξουν. Για container restart, το UID πρέπει να μείνει ίδιο και το restart count να αυξηθεί. Για αντικατάσταση Pod, το νέο UID είναι το κρίσιμο στοιχείο. Για απώλεια node, πρόσθεσε το χρονικό παράθυρο όπου το API διατηρεί τελευταία γνωστή κατάσταση.

Αν η πρόβλεψή σου δεν επιβεβαιωθεί, μην αλλάξεις αμέσως δεύτερο πράγμα. Κράτησε την έξοδο και εντόπισε ποια υπόθεση ήταν λάθος. Το επόμενο κεφάλαιο οργανώνει ακριβώς αυτή τη σειρά δουλειάς.

ΚΕΦΑΛΑΙΟ 21

Η σειρά της διάγνωσης

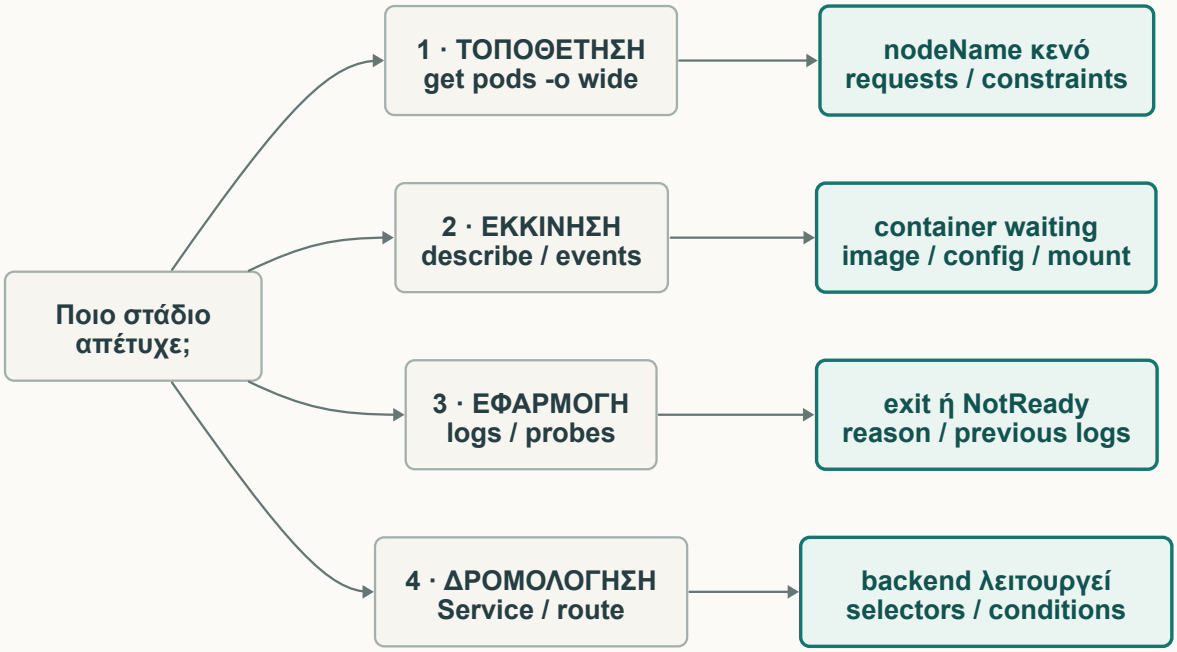
Βρες το πρώτο στάδιο που δεν ολοκληρώθηκε. Η επόμενη εντολή πρέπει να απαντά σε αυτό, όχι σε όλο το cluster μαζί.

Η χρήσιμη σειρά είναι τοποθέτηση, εκκίνηση, εφαρμογή και δρομολόγηση. Δεν είναι άκαμπτο τελετουργικό που επαναλαμβάνεις από την αρχή σε κάθε γνωστή βλάβη. Είναι τρόπος να αποφύγεις να ψάχνεις Rails logs για container που δεν δημιουργήθηκε ή Gateway rules για Pod που δεν έχει node.

Ξεκίνα με το πραγματικό αίτημα που αποτυγχάνει. Από ποιον client, προς ποιο όνομα και ποια θύρα; Με ποιο context κοιτάς το API; Η φράση «δεν δουλεύει το Kubernetes» δεν ορίζει ακόμη πρόβλημα που μπορείς να ελέγξεις.

> **Από το Kamal στο Kubernetes**
> Η εντολή logs χρειάζεται τώρα πιο ακριβή στόχο από έναν σταθερό server. Pod, container, UID και προηγούμενη εκτέλεση καθορίζουν ποια ιστορία διαβάζεις και ποια μπορεί να έχει ήδη χαθεί.

Το δέντρο που κρατάς ανοιχτό



Σχήμα 21.1 · Τέσσερις διαδρομές διάγνωσης. Η πρώτη πληροφορία αποκλείει άσχετα επίπεδα πριν διαβάσεις περισσότερο output.

TERMINAL

```
kubectl config current-context
kubectl get deployment,pods,service -n k8s-book
kubectl get pods -n k8s-book -o wide
```

Στην τοποθέτηση αναζητάς `nodeName`, requests, constraints και scheduling events. Στην εκκίνηση κοιτάς waiting reason, image, references και mounts. Στην εφαρμογή χρειάζεσαι logs, προηγούμενο τερματισμό και probes. Στη δρομολόγηση συγκρίνεις Service selector, EndpointSlices και HTTPRoute conditions.

Το δέντρο είναι εννοιολογικό. Οι εργασίες των controllers συμβαίνουν παράλληλα. Ένα Pod μπορεί να τρέχει ενώ το route παραμένει λανθασμένο. Αυτό δεν αναιρεί τη σειρά διάγνωσης. Σημαίνει ότι έχεις ήδη αποδείξει πως κάποια προηγούμενα στάδια ολοκληρώθηκαν.

Conditions και events

Τα conditions δίνουν τρέχουσα καταγεγραμμένη κατάσταση συγκεκριμένων πτυχών ενός αντικειμένου. Τα events προσφέρουν χρονολογημένες εξηγήσεις ενεργειών και αποτυχιών. Το `describe` συγκεντρώνει χρήσιμα πεδία και σχετικά events χωρίς να χρειάζεται να διαβάσεις ολόκληρο το JSON.

```
TERMINAL

kubectl describe deployment rails-map
kubectl describe pods -l app=rails-map
kubectl get events --sort-by=.metadata.creationTimestamp
```

Η παλιά αποτυχία image pull μπορεί να παραμένει ορατή αφού διορθώσεις την εικόνα. Αυτό δεν σημαίνει ότι το τρέχον rollout εξακολουθεί να αποτυγχάνει. Κοίτα τη νέα γενιά, το τρέχον Pod template και τα αντικείμενα που είναι τώρα ενεργά.

Αν ένα Deployment αναφέρει ότι ξεπέρασε την προθεσμία προόδου, κατέβα στα Pods της νέας αναθεώρησης. Το condition δείχνει ποια υπόσχεση δεν εκπληρώθηκε. Η πραγματική αιτία μπορεί να είναι readiness, image, πόροι ή άλλη εξάρτηση. Το `rollout status` είναι τελικός έλεγχος προόδου, όχι πλήρης εξήγηση.

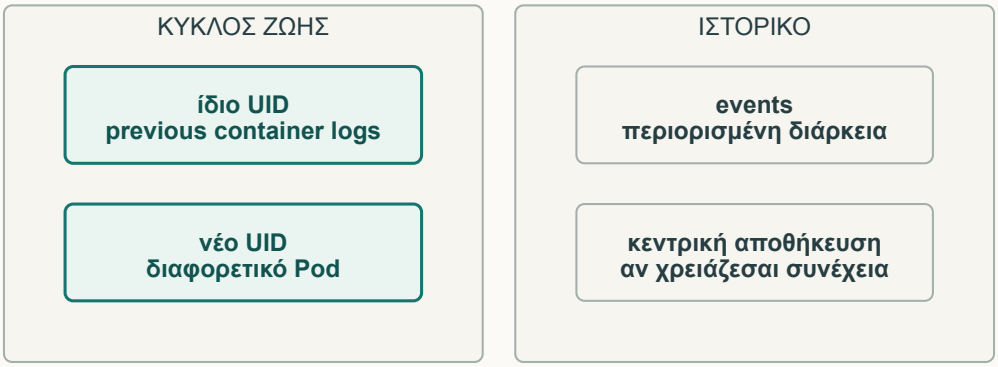
Τα σωστά logs

Για το κανονικό web μπορείς να δεις τις πρόσφατες γραμμές με το όνομα του Pod στην έξοδο.

```
TERMINAL

kubectl logs -l app=rails-map -c web \
  --prefix=true --tail=40
```

Αν θέλεις να ξαναδείς την προηγούμενη εκτέλεση container, χρησιμοποίησε το ξεχωριστό πείραμα probe με ένα replica. Περίμενε πρώτα να συμβεί restart.

```
TERMINAL

kubectl apply -f manifests/ch15/probe-loop.yaml
kubectl get pods -l app=probe-loop
kubectl logs deployment/probe-loop --previous --tail=40
```



Σχήμα 21.2 · Το previous ακολουθεί εκτελέσεις container μέσα στο ίδιο Pod. Η αντικατάσταση Pod δημιουργεί άλλο UID και διαφορετικό σημείο αναζήτησης.

Αν δεν υπάρχει προηγούμενη εκτέλεση, η εντολή μπορεί να αποτύχει. Αυτό αφορά τον συγκεκριμένο στόχο. Σε πραγματική βλάβη διάλεξε ρητά το Pod και το container που εμφανίζουν το πρόβλημα.

Όταν τελειώσεις, σταμάτησε ξανά το βοηθητικό workload.

TERMINAL

```
kubectl scale deployment/probe-loop --replicas=0
```

Η διαγραφή Pods, η περιστροφή αρχείων και η διάρκεια διατήρησης περιορίζουν το τοπικό ιστορικό. Για αναζήτηση πέρα από τη ζωή ενός Pod χρειάζεσαι χωριστή συλλογή και αποθήκευση logs. Δες το Logging Architecture.

Έλεγξε τη διαδρομή από μέσα προς τα έξω

Αν τα web είναι έτοιμα, κάλεσε το Service από τον client του cluster.

TERMINAL

```
kubectl exec book-client -- ruby -rnet/http -e \
  'puts Net::HTTP.get(URI("http://rails-map/version"))'
kubectl get endpointslices \
  -l kubernetes.io/service-name=rails-map -o yaml
kubectl get httproute rails-map -o yaml
```

Αν πετύχει το εσωτερικό Service και αποτύχει το εξωτερικό HTTP, έλεγξε host, path, parent listener, backend reference και πρόσβαση στον proxy. Αν αποτύχει και το Service, ξεκίνα από αυτό.

Σε αποτυχία TLS ξεχώρισε τη σύνδεση TCP, την επαλήθευση πιστοποιητικού και την HTTP απάντηση. Ο client μπορεί να σταματήσει πριν στείλει HTTP request. Χωρίς αίτημα δεν περιμένεις εγγραφή στα Rails access logs.

Μία αλλαγή, ένας επανέλεγχος

Κράτησε το αρχικό σύμπτωμα και το στοιχείο που στηρίζει την υπόθεσή σου. Κάνε μία αλλαγή και επανάλαβε το ίδιο αίτημα από τον ίδιο client. Ο επανέλεγχος πρέπει να διασχίζει το σημείο που απέτυχε.

Διόρθωση selector απαιτεί έλεγχο μέσω Service. Διόρθωση TLS απαιτεί HTTPS με επαλήθευση. Για migration χρειάζεσαι επιτυχημένο Job και λειτουργία του κώδικα με το schema. Το `kubectl get pods` δεν αποδεικνύει όλα αυτά.

Χρησιμοποίησε το δέντρο σε δύο προηγούμενες βλάβες. Γράψε τρεις γραμμές για καθεμία. Σύμπτωμα, στοιχείο που ξεχώρισε την αιτία και ίδιος επανέλεγχος μετά τη διόρθωση. Οι γενικές διαδικασίες βρίσκονται στο Debug applications.

Παράδειγμα λύσης. Λάθος selector σημαίνει μηδέν endpoints με έτοιμα web. Μετά τη διόρθωση επανάλαβε το ίδιο HTTP αίτημα. Σε liveness loop αναζήτησε `Unhealthy` και `Killing`. Έπειτα παρατήρησε αρκετούς κύκλους probe χωρίς νέο restart.

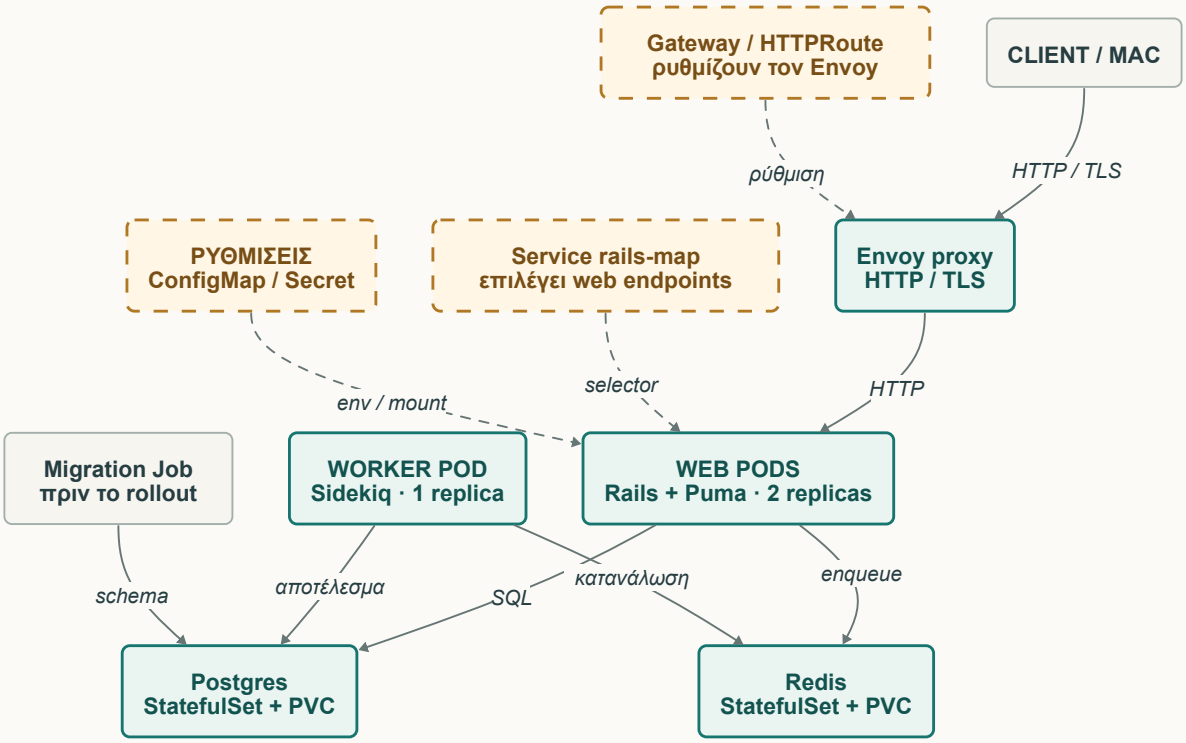
ΚΕΦΑΛΑΙΟ 22

Όλη η εφαρμογή από το μηδέν

Το deploy τελειώνει όταν έχεις ελέγξει την εφαρμογή, την εργασία και τα δεδομένα που περίμενες να μείνουν.

Ο χάρτης έχει πλέον όλα του τα κομμάτια. Δύο web Pods δέχονται HTTP. Ένας Sidekiq worker καταναλώνει εργασίες από Redis και ενημερώνει την Postgres. Ένα migration Job προετοιμάζει το schema πριν αλλάξουν οι εκδόσεις. Το Gateway δίνει την είσοδο και το Service επιλέγει τα έτοιμα web endpoints.

Το τελευταίο πείραμα τα συνδυάζει. Θα περάσουμε από v1 σε v2 ενώ φτάνουν αιτήματα και υπάρχει εργασία σε εξέλιξη. Θα ελέγξουμε ότι η ίδια εργασία ολοκληρώθηκε μία φορά, ότι ο νέος worker χρησιμοποιεί το νέο πεδίο και ότι η επιστροφή στη v1 διαβάζει ακόμη τα δεδομένα.



Σχήμα 22.1 · Ο πλήρης χάρτης. Τα συνεχόμενα βέλη είναι κίνηση της εφαρμογής. Οι διακεκομμένες συνδέσεις δείχνουν ρυθμίσεις, επιλογή ή διαχείριση. Το Service επιλέγει endpoints και η κίνηση φτάνει στα αντίστοιχα web Pods.

> **Από το Kamal στο Kubernetes**
> Η συνοχή του release εξακολουθεί να μετριέται στην εφαρμογή. Η επιτυχία του εργαλείου δεν αρκεί χωρίς έλεγχο αιτημάτων, εργασιών και δεδομένων. Το cluster κάνει ορατά περισσότερα ενδιάμεσα στάδια αυτής της ίδιας ευθύνης.

Ένα καινούριο τοπικό εργαστήριο

Το `final-lab-up.sh` χρησιμοποιεί νέο cluster με όνομα `k8s-book-final`. Είναι άλλη εγκατάσταση του μικρού προφίλ, με έναν node. Τα προηγούμενα clusters μένουν στη θέση τους, ώστε να μπορείς να συγκρίνεις τις καταγραφές. Χρειάζεται διαθέσιμη μνήμη στο Docker για όσα αφήνεις ταυτόχρονα ενεργά.

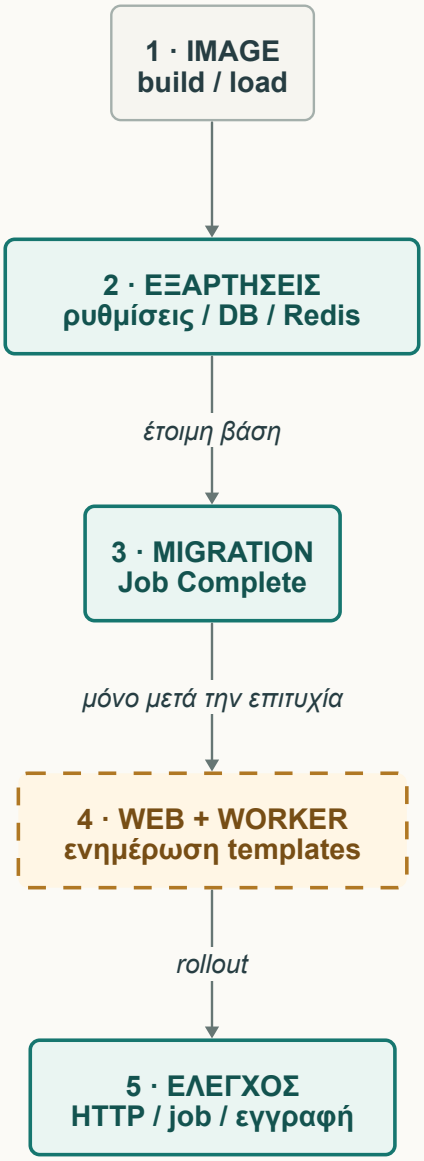
TERMINAL

```
bash scripts/final-lab-up.sh
export KUBECONFIG="$PWD/build/kubeconfig-final"
kubectl config current-context
kubectl get pods,pvc
```

Το context πρέπει να είναι `kind-k8s-book-final`. Η πρώτη εκτέλεση στήνει νέο schema v1. Αν ξανατρέξεις το script στο ίδιο cluster, η υπάρχουσα βάση παραμένει. Το «από το μηδέν» περιγράφει την πρώτη εγκατάσταση, όχι υπόσχεση ότι κάθε εκτέλεση μηδενίζει δεδομένα.

Η σειρά των αρχείων

Τα scripts εκτελούν την εγκατάσταση, αλλά η σειρά πρέπει να είναι αναγνώσιμη. Πρώτα υπάρχουν το cluster και οι εικόνες. Ύστερα το Secret σύνδεσης και οι δύο StatefulSets με τα PVC τους. Μετά μπορεί να ξεκινήσει ο migration Job. Μόνο η επιτυχία του επιτρέπει να ενημερωθούν web και worker.



Σχήμα 22.2 · Η σειρά του release. Κάθε επόμενο βήμα έχει συγκεκριμένη προϋπόθεση. Η αποτυχία migration σταματά την αλλαγή των Pod templates.

Αρχείο ή βήμα	Τι προσθέτει	Τι ελέγχεις μετά
`scripts/cluster-up.sh`	API, node, namespace και αρχικό web	Σωστό context και node Ready
`scripts/build-images.sh`	Τις εικόνες v1 και v2 στο node	Επιτυχία build και φόρτωσης

Αρχείο ή βήμα	Τι προσθέτει	Τι ελέγχεις μετά
`ch13/database-secret.yaml`	Δημόσιες δοκιμαστικές ρυθμίσεις βάσης	Υπάρχει το αναμενόμενο Secret
`ch13/postgres.yaml`	Postgres, Service και PVC	StatefulSet έτοιμο και PVC Bound
`ch13/redis.yaml`	Redis, Service και PVC	Redis έτοιμη να δεχτεί συνδέσεις
`ch14/migrate-v1.yaml`	Το αρχικό schema	Job Complete
`ch14/worker.yaml`	Sidekiq ως χωριστό Deployment	Worker έτοιμος και logs εκκίνησης
`ch12/configmap.yaml`	Το δημόσιο μήνυμα ρύθμισης	`message` με την αναμενόμενη τιμή
`ch22/web.yaml`	Web με βάση, Redis, probes και όρια	Δύο έτοιμα web
`ch10/service.yaml`	Σταθερό όνομα προς τα web	Δύο ready endpoints
`ch11/gateway.yaml` και `route.yaml`	HTTP, TLS και κανόνα προς το Service	Conditions και πραγματικό αίτημα

Οι διαδρομές του πίνακα που αρχίζουν με `ch` βρίσκονται μέσα στο `manifests/`. Ο migration Job αποκτά μοναδικό όνομα σε κάθε release. Ο ίδιος κώδικας δεν εκτελείται ανεξάρτητα από κάθε web Pod.

Το τελικό web manifest συνδυάζει όσα είδες χωριστά. Το `DATABASE_URL` έρχεται από Secret. Το `BOOK_MESSAGE` έρχεται από ConfigMap και υπάρχει επίσης κανονικό volume mount. Η startup probe επιτρέπει το boot, η readiness ελέγχει αν το web μπορεί να εξυπηρετήσει και η liveness αφορά τη διεργασία. Τα requests συμμετέχουν στην τοποθέτηση και τα limits επιβάλλουν όρια εκτέλεσης.

Δεν προσθέτουμε HPA σε αυτή τη σύνθεση. Ο στόχος των δύο replicas παραμένει σταθερός, ώστε να μετράμε ένα rolling update χωρίς ταυτόχρονη αλλαγή κλίμακας. Το κεφάλαιο 18 έχει το χωριστό πείραμα όπου ο HPA αποκτά τον έλεγχο του πλήθους.

Το πρώτο αίτημα από το Mac

Σε δεύτερο terminal κράτησε ανοιχτή την πρόσβαση στο Gateway του τελικού cluster.

TERMINAL

```
BOOK_CLUSTER=k8s-book-final \
BOOK_HTTP_PORT=9088 BOOK_HTTPS_PORT=9443 \
  bash scripts/gateway-forward.sh
```

Στο πρώτο terminal έλεγξε HTTP και HTTPS. Το δεύτερο αίτημα επαληθεύει το πιστοποιητικό με τη δοκιμαστική CA του εργαστηρίου. Δεν παρακάμπτει τον έλεγχο TLS.

```
TERMINAL

curl --fail -H 'Host: rails.book.test' \
  http://127.0.0.1:9088/version
curl --fail --cacert build/tls/book.crt \
  --resolve rails.book.test:9443:127.0.0.1 \
  https://rails.book.test:9443/version
```

Η απάντηση δίνει `v1` και το όνομα του Pod. Δεν χρειάζεται να αλλάζει Pod σε κάθε δεύτερο αίτημα. Η κατανομή συνδέσεων και η επαναχρησιμοποίησή τους δεν είναι υπόσχεση αυστηρού round robin ανά HTTP request.

Δημιούργησε τώρα μία εργασία. Το token είναι κλειδί του δικού μας παραδείγματος. Στείλε το ίδιο token ξανά και έλεγξε ότι υπάρχει μία εγγραφή.

```
TERMINAL

curl --fail -H 'Host: rails.book.test' \
  --data 'token=chapter-22-first' \
  http://127.0.0.1:9088/tasks
curl --fail -H 'Host: rails.book.test' \
  http://127.0.0.1:9088/tasks
```

Στην αρχή η εγγραφή μπορεί να είναι `queued` ή `working`. Αργότερα πρέπει να είναι `done` με `completions` ίσο με ένα. Το unique index προστατεύει την ταυτότητα της εγγραφής και το κλείδωμα στη βάση προστατεύει τη μετάβαση κατάστασης. Ούτε το Deployment ούτε η ουρά προσφέρουν από μόνα τους αυτή τη σημασιολογία.

Το release με εργασία σε εξέλιξη

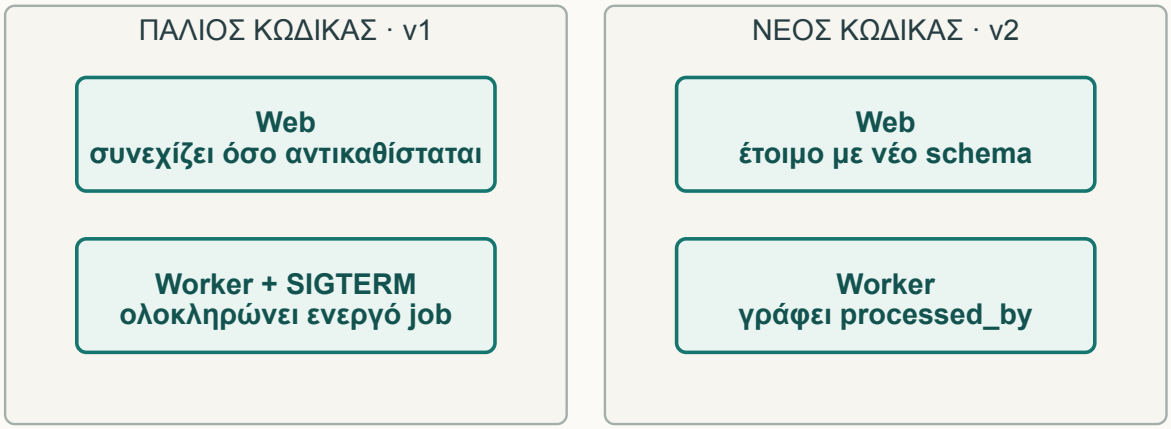
Σταμάτησε το χειροκίνητο port-forward με Ctrl-C πριν τον αυτοματοποιημένο έλεγχο. Το script χρειάζεται τις ίδιες θύρες και κρατά δική του καταγραφή κάθε εξωτερικού αιτήματος.

```
TERMINAL

python3 scripts/check-integration.py
```

Η πρώτη εκτέλεση προϋποθέτει ότι το νέο cluster έχει ακόμη schema v1. Μετά από επιτυχημένη μετάβαση σε v2, η επανάληψη γράφεται ρητά ως επανάληψη με το υπάρχον διευρυμένο schema.

```
TERMINAL

python3 scripts/check-integration.py --repeat
```

Το πείραμα επιμηκύνει την εργασία σε 20 δευτερόλεπτα, στέλνει το ίδιο token δύο φορές και περιμένει να δει `working` πριν αρχίσει το release. Έπειτα κρατά συνεχή HTTP αιτήματα προς το Gateway και εκτελεί τη σειρά του migration και των δύο rollouts.



Σχήμα 22.3 · Παλιός και νέος κώδικας συνυπάρχουν για λίγο. Το nullable πεδίο επιτρέπει στη v1 να ολοκληρώσει την παλιά εργασία και στη v2 να γράψει επιπλέον πληροφορία.

Η v2 προσθέτει το nullable `processed_by`. Δεν μετονομάζει πεδίο που χρησιμοποιεί η v1 και δεν απαιτεί αμέσως τιμή σε παλιές εγγραφές. Ο παλιός worker λαμβάνει SIGTERM και έχει χρόνο να ολοκληρώσει την ενεργή δουλειά. Ο νέος worker αναλαμβάνει επόμενες εργασίες και γράφει το δικό του όνομα στο νέο πεδίο.

Το script ελέγχει ξεχωριστά ότι άλλαξε το UID του worker, ότι η αρχική εγγραφή ολοκληρώθηκε μία φορά και ότι μια νέα εργασία της v2 έγραψε το `processed_by`. Η απλή ύπαρξη καινούριου Pod δεν θα αποδείκνυε κανένα από αυτά τα αποτελέσματα.

Η επαναφορά έχει συγκεκριμένο όριο

Η χειροκίνητη εντολή για τη νέα έκδοση είναι η ακόλουθη. Το πρόθεμα `BOOK_CLUSTER` είναι ουσιαστικό, επειδή τα scripts γνωρίζουν και το αρχικό εργαστήριο.

TERMINAL

```
BOOK_CLUSTER=k8s-book-final bash scripts/release.sh v2
BOOK_CLUSTER=k8s-book-final bash scripts/release.sh v1
```

Η δεύτερη εντολή επαναφέρει συμβατό κώδικα v1. Δεν εκτελεί `db:rollback`. Η στήλη της v2 και οι εγγραφές παραμένουν. Αυτή είναι σκόπιμη επιλογή της άσκησης και όχι γενικός κανόνας ότι κάθε παλιός κώδικας συνεργάζεται με κάθε νέο schema.

Σχήμα 22.4 · Τρεις χωριστοί έλεγχοι μετά την επιστροφή στη v1. Έκδοση HTTP, αποτέλεσμα εργασίας και διατήρηση δεδομένων.

Για πιο απαιτητική αλλαγή θα χώριζες την επέκταση, τη μετάβαση του κώδικα και την αφαίρεση του παλιού schema σε διαφορετικά βήματα. Το αναστρέψιμο Pod template δεν κάνει αναστρέψιμη μια καταστροφική αλλαγή δεδομένων.

Τι μετρήθηκε πραγματικά

Η πρώτη εκτέλεση έγινε σε νέο cluster και απέδειξε τη μετάβαση από αρχικό schema. Η εγκατάσταση, το migration, η εργασία και τα rollouts έγιναν. Ένα εσωτερικό HTTP GET μετά το rollback όμως έληξε με timeout και διέκοψε τους τελευταίους ελέγχους. Αργότερα η ίδια διαδρομή απάντησε κανονικά. Στη δεύτερη προσπάθεια ένα εσωτερικό GET απέτυχε μετά το rollout της v2.

Ο έλεγχος διορθώθηκε ώστε να περιμένει με όριο χρόνου την τελική κατάσταση του Service και να καταγράφει τυχόν αποτυχημένα GET. Η χωριστή ροή εξωτερικών HTTP αιτημάτων συνεχίζει να καταγράφει κάθε αποτυχία χωρίς επανάληψη του ίδιου δείγματος. Η αλλαγή στον έλεγχο δεν προσθέτει retry στον client της εφαρμογής.

Εκτέλεση	Εξωτερικά αιτήματα	Αποτυχημένες απαντήσεις	Τελικοί έλεγχοι
Πρώτη, νέο schema v1	524	1	Διακοπή σε εσωτερικό timeout μετά το rollback
Δεύτερη, υπάρχον schema	62	0	Διακοπή σε εσωτερικό timeout μετά το v2 rollout
Τρίτη, υπάρχον schema	257	0	Πέρασαν όλοι

Στην τρίτη εκτέλεση παρατηρήθηκαν και οι δύο εκδόσεις από το Gateway, επιτυχές TLS, νέα εργασία με `processed_by`, μία ολοκλήρωση για το διπλό token και επιτυχής επιστροφή στη v1 με τα δεδομένα στη θέση τους. Δεν εμφανίστηκε εσωτερικό timeout σε εκείνη την εκτέλεση.

Αυτή η μέτρηση δεν αποδεικνύει μηδενικές διακοπές σε κάθε deploy. Η αρχική αποτυχία παραμένει μέρος του αποτελέσματος. Δεν απομονώθηκε επαρκώς η αιτία της ώστε να αποδοθεί αποκλειστικά στο Puma, στο δίκτυο ή στη διάδοση των endpoints. Χρειάζεται νέα στοχευμένη καταγραφή για τέτοιο συμπέρασμα.

Η απώλεια Redis node, το failover Postgres, το SIGKILL worker και το ασυμβίβαστο schema χρειάζονται χωριστά πειράματα. Δεν ελέγχθηκαν σε αυτή την εκτέλεση.

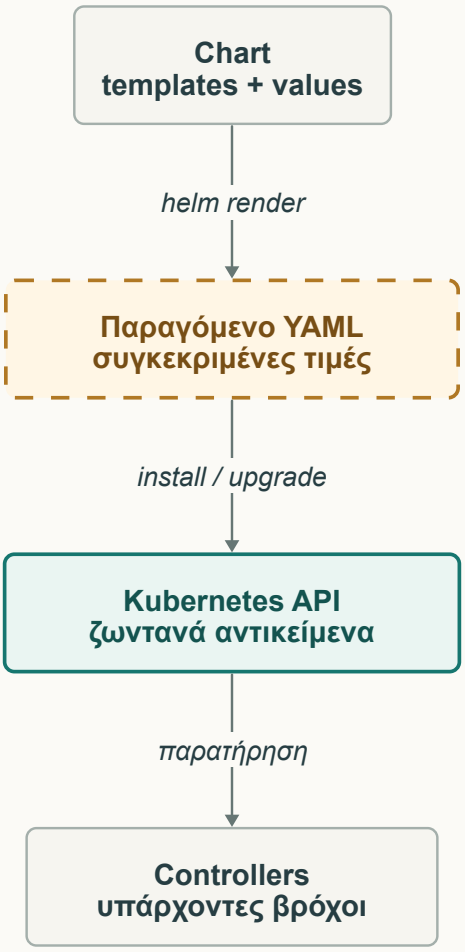
ΚΕΦΑΛΑΙΟ 23

Helm σε ένα κεφάλαιο

Όταν τα ίδια manifests χρειάζονται ονόματα, εκδόσεις και ρυθμίσεις για διαφορετικές εγκαταστάσεις, χρειάζεσαι τρόπο να τα συσκευάσεις.

Ως εδώ τα YAML ήταν απλά αρχεία. Αυτό βοήθησε να φαίνεται ποιο πεδίο αλλάζει τη συμπεριφορά. Αν όμως μοιράσεις μια εφαρμογή σε άλλες ομάδες, δεν θέλεις να τους ζητάς να ψάχνουν είκοσι αρχεία για image tag, host και μέγεθος δίσκου. Θέλεις ένα πακέτο με σαφείς παραμέτρους.

Το Helm ονομάζει αυτό το πακέτο chart. Το chart περιέχει metadata, προεπιλεγμένα values και templates. Η συγκεκριμένη εγκατάσταση του στο cluster λέγεται release. Το ίδιο chart μπορεί να εγκατασταθεί με διαφορετικό όνομα και διαφορετικές τιμές. Οι βασικές έννοιες και η δομή περιγράφονται στο επίσημο Getting Started.



Σχήμα 23.1 · Το chart και οι τιμές παράγουν κανονικά Kubernetes manifests. Το API και οι controllers εξακολουθούν να εκτελούν τη δουλειά που ήδη γνωρίζεις.

Χρησιμοποίησες ήδη ένα πραγματικό chart όταν το εργαστήριο εγκατέστησε το Envoy Gateway. Τώρα θα δεις τον μηχανισμό σε ένα μόνο ConfigMap. Δεν χρειάζεται να μετατρέψουμε όλη την εφαρμογή σε templates για να γίνει κατανοητός.

> **Από το Kamal στο Kubernetes**
> Οι κοινές ρυθμίσεις μιας εγκατάστασης μπορούν να παραμετροποιούνται με values ή overlays. Το παραγόμενο αποτέλεσμα όμως είναι Kubernetes YAML. Η συσκευασία δεν αλλάζει τη σημασία των πεδίων που ήδη έμαθες.

Ένα chart που διαβάζεται ολόκληρο

Επιστρέφουμε στο αρχικό cluster και χρησιμοποιούμε χωριστό όνομα ConfigMap. Το μήνυμα του κανονικού web δεν αλλάζει από αυτό το παράδειγμα.

TERMINAL

```
export KUBECONFIG="$PWD/build/kubeconfig"
kubectl config current-context
```

Το metadata του chart δηλώνει την έκδοση του πακέτου.

charts/book-settings/Chart.yaml

```
apiVersion: v2
name: book-settings
description: One ConfigMap for the book's Helm example
type: application
version: 0.1.0
appVersion: "v1"
```

Το `version` αφορά το chart. Το `appVersion` είναι πληροφορία για την εφαρμογή και δεν αλλάζει αυτόματα κάποιο image. Μια εγκατάσταση έχει επιπλέον τη δική της release revision. Αυτοί οι τρεις αριθμοί απαντούν σε διαφορετικές ερωτήσεις.

Το προεπιλεγμένο αρχείο values έχει μία τιμή.

charts/book-settings/values.yaml

```
message: first
```

Το template παίρνει αυτή την τιμή και τα στοιχεία του release.

charts/book-settings/templates/configmap.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-settings
  namespace: {{ .Release.Namespace }}
data:
  message: {{ .Values.message | quote }}
```

Τα `{{ ... }}` εκτελούνται από το Helm πριν φτάσει το αποτέλεσμα στο Kubernetes. Το API δεν γνωρίζει τι είναι `.Values.message`. Θα λάβει ένα κανονικό ConfigMap με συγκεκριμένο όνομα και συγκεκριμένο string.

Δες πρώτα το αποτέλεσμα τοπικά.

TERMINAL

```
helm lint charts/book-settings
helm template book-chart charts/book-settings \
  -n k8s-book
```

Το όνομα θα είναι `book-chart-settings` και το μήνυμα `first`. Το `template` δεν δημιουργεί αντικείμενο στο cluster. Το `lint` επίσης δεν αποδεικνύει ότι όλα τα παραγόμενα αντικείμενα θα

γίνουν έτοιμα. Στον έλεγχο του κεφαλαίου εξετάστηκε και η αποδοχή του παραγόμενου YAML από το πραγματικό API.

Εγκατάσταση, αλλαγή και ιστορικό

TERMINAL

```
helm upgrade --install book-chart charts/book-settings \
  -n k8s-book --reset-values
kubectl get configmap book-chart-settings \
  -o jsonpath='{.data.message}{"\n"}'
helm history book-chart -n k8s-book
```

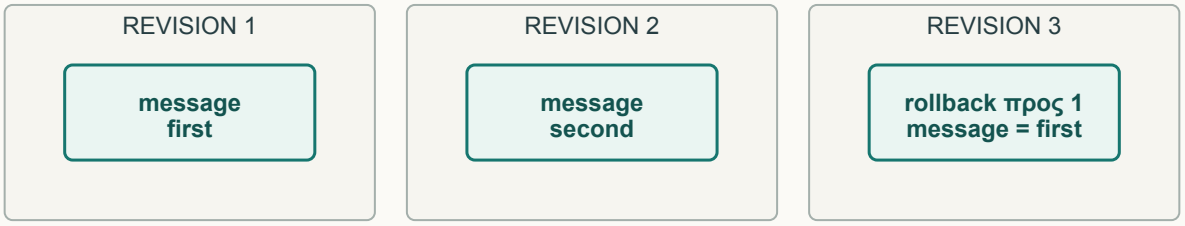
Η πρώτη εγκατάσταση δημιουργεί revision 1. Το παράδειγμα χρησιμοποιεί `upgrade --install` ώστε η εντολή να μπορεί επίσης να ενημερώσει release που υπάρχει ήδη. Αν έχεις επαναλάβει την άσκηση, διάβασε τον πραγματικό αριθμό revision από το history.

Η δεύτερη ρύθμιση βρίσκεται στο `values-second.yaml`. Περιέχει μόνο `message: second`.

TERMINAL

```
helm upgrade book-chart charts/book-settings \
  -n k8s-book \
  -f charts/book-settings/values-second.yaml
kubectl get configmap book-chart-settings \
  -o jsonpath='{.data.message}{"\n"}'
helm get manifest book-chart -n k8s-book
```

Η ζωντανή τιμή πρέπει τώρα να είναι `second`. Το `get manifest` σου επιτρέπει να διαβάσεις τα manifests του release, χωρίς να μαντεύεις τι παρήγαγε ο συνδυασμός templates και values.



Σχήμα 23.2 · Το history αφορά revisions της εγκατάστασης. Το rollback δημιουργεί νέα revision που χρησιμοποιεί το περιεχόμενο μιας προηγούμενης.

Στην πρώτη εκτέλεση του παραδείγματος μπορείς να επιστρέψεις στην αρχική revision με την ακόλουθη εντολή.

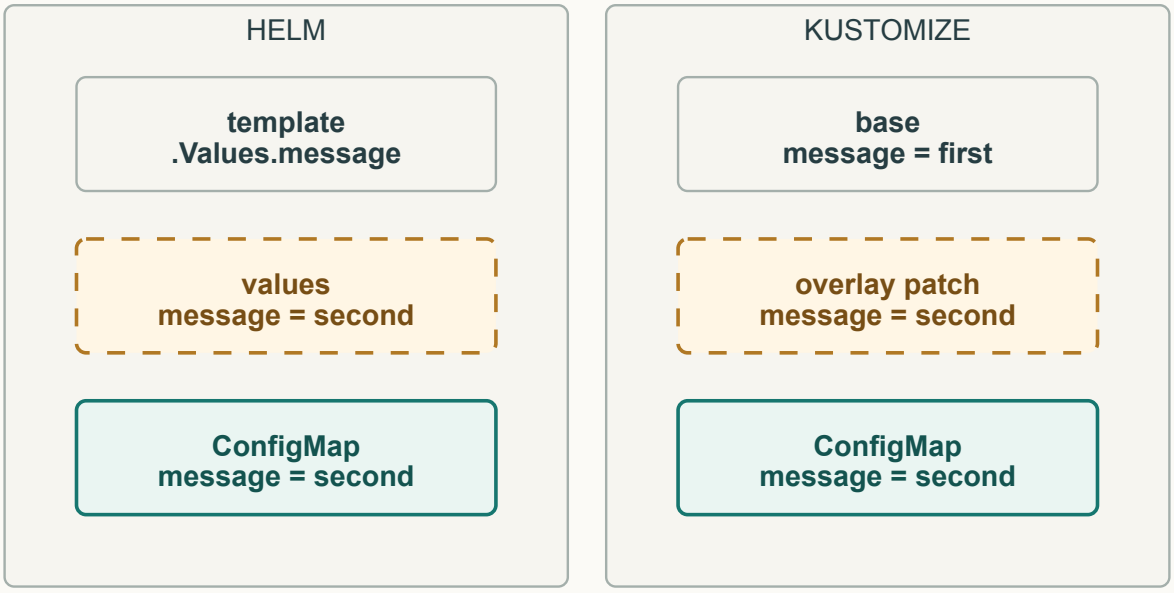
TERMINAL

```
helm rollback book-chart 1 -n k8s-book
kubectl get configmap book-chart-settings \
  -o jsonpath='{.data.message}{"\n"}'
```

Η καταγεγραμμένη εκτέλεση με Helm 4.2.4 πέρασε από `first` σε `second` και πίσω σε `first`. Η επαναφορά αφορούσε το συγκεκριμένο ConfigMap. Σε μεγαλύτερο chart δεν θα σήμαινε αυτόματη επαναφορά δεδομένων της Postgres, εξωτερικών υπηρεσιών ή κάθε αλλαγής που έκανε ένας migration Job.

Το ίδιο μικρό πρόβλημα με Kustomize

Αν έχεις λίγα περιβάλλοντα και θέλεις παραλλαγές των δικών σου YAML, μπορεί να αρκεί το Kustomize που είναι διαθέσιμο μέσα από το kubectl. Κρατά ένα base από κανονικά manifests και εφαρμόζει τις αλλαγές ενός overlay. Δεν χρειάζεται γλώσσα template μέσα σε κάθε πεδίο.



Σχήμα 23.3 · Δύο διαδρομές για το ίδιο μήνυμα. Το Helm χρησιμοποιεί values και template. Το Kustomize χρησιμοποιεί base και patch. Και οι δύο καταλήγουν σε κανονικό ConfigMap.

Το base του παραδείγματος έχει ένα ConfigMap με `message` ίσο με `first`. Το overlay `lab` αλλάζει το μήνυμα σε `second` και ορίζει το namespace.

manifests/ch23/overlays/lab/kustomization.yaml

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
namespace: k8s-book
resources:
- ../../base
patches:
- path: message.yaml
```

Το patch είναι επίσης απλό YAML.

manifests/ch23/overlays/lab/message.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: kustomize-settings
data:
  message: second
```

Δες το αποτέλεσμα πριν το εφαρμόσεις.

TERMINAL

```
kubectl kustomize manifests/ch23/overlays/lab
kubectl apply -k manifests/ch23/overlays/lab
kubectl get configmap kustomize-settings \
  -o jsonpath='{.data.message}{"\n"}'
```

Η τιμή που μετρήθηκε ήταν `second`. Το overlay δεν τροποποίησε το base αρχείο στον δίσκο. Παρήγαγε μια διαφορετική επιθυμητή κατάσταση. Οι δυνατότητες base, patch και generation τεκμηριώνονται στο Declarative Management Using Kustomize.

Για τη δική μας μικρή εφαρμογή τα απλά manifests παραμένουν κατανοητά. Το Helm έχει πρακτική αξία όταν θέλεις να εγκαταστήσεις ένα συντηρούμενο πακέτο ή να διανείμεις το δικό σου με σαφείς παραμέτρους. Το Kustomize έχει πρακτική αξία όταν θέλεις μικρές, ορατές αποκλίσεις πάνω σε κοινή βάση.

Κανένα από τα δύο δεν αντικαθιστά τη γνώση του παραγόμενου YAML. Αν ένας selector δεν βρίσκει Pods, η διάγνωση παραμένει ίδια. Πρώτα διάβασε το τελικό Service και τα labels. Μετά γύρισε στο template ή στο overlay που παρήγαγε τη λάθος τιμή.

Η άσκηση είναι να αλλάξεις το μήνυμα και με τους δύο τρόπους, να κρατήσεις το παραγόμενο YAML και να συγκρίνεις μόνο το `data.message`. Έπειτα εξήγησε ποιος εκτελεί συνεχή συμφιλίωση αφού κλείσει η εντολή Helm. Για αυτό το ConfigMap, το Helm CLI δεν μένει ως daemon να διορθώνει κάθε μεταγενέστερο χειροκίνητο edit.

Λύση της άσκησης. Και οι δύο διαδρομές παράγουν συγκεκριμένο `data.message` στο ConfigMap. Μετά την εντολή, το αντικείμενο παραμένει στο API. Το Helm CLI δεν παρακολουθεί συνεχώς τη χειροκίνητη αλλαγή του. Οι controllers του Kubernetes συνεχίζουν μόνο τις ευθύνες που αντιστοιχούν στα αντικείμενα που διαχειρίζονται.

ΚΕΦΑΛΑΙΟ 24

Ο χάρτης του τι έρχεται μετά

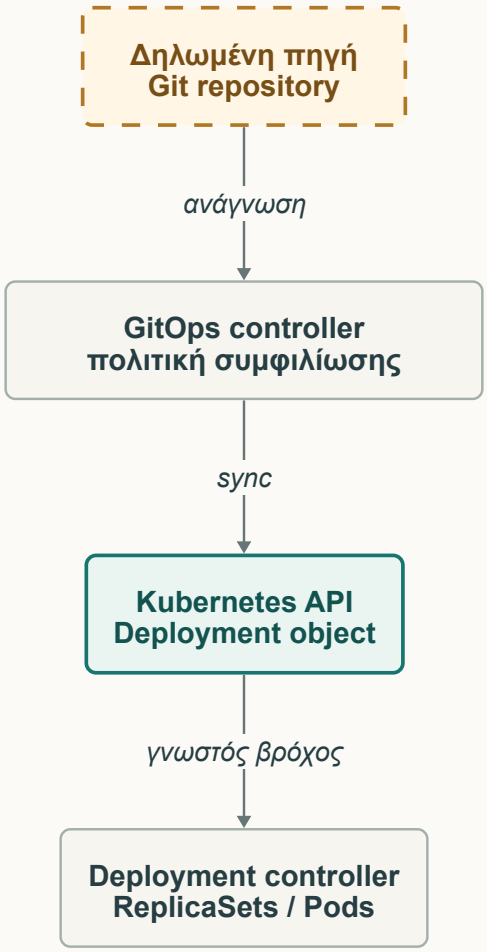
Το επόμενο εργαλείο πρέπει να αποκτήσει συγκεκριμένη θέση στον χάρτη και συγκεκριμένη ευθύνη.

Έχεις πλέον αρκετές έννοιες ώστε μια νέα λέξη να μη μοιάζει με καινούριο σύμπαν. Μπορείς να ρωτήσεις πού δρα, ποια δήλωση διαβάζει, τι αλλάζει και πώς ελέγχεται η αποτυχία της. Αυτό το κεφάλαιο τοποθετεί πέντε επόμενα θέματα. Δεν εγκαθιστά άλλη στοίβα πάνω στο εργαστήριο.

> **Από το Kamal στο Kubernetes**
> Η μετάβαση δεν απαιτεί να αγοράσεις όλο τον επόμενο χάρτη. Το ίδιο κριτήριο επιλογής εργαλείου παραμένει. Μια συγκεκριμένη ανάγκη, σαφής ευθύνη και τρόπος να ελέγξεις αν καλύφθηκε.

GitOps δίπλα στο apply

Ως τώρα εσύ έστελνες τα manifests στο API. Σε μια ροή GitOps, ένας controller παρακολουθεί μια δηλωμένη πηγή, συχνά repository Git, και συγκρίνει το επιθυμητό περιεχόμενο με το cluster. Η αλλαγή που εγκρίνεται στο repository μπορεί έτσι να γίνει η αλλαγή που εφαρμόζεται.



Σχήμα 24.1 · Ο επόμενος βρόχος βρίσκεται πριν από τους βρόχους που ήδη ξέρεις. Η πηγή περιγράφει τα αντικείμενα και ο GitOps controller φροντίζει τη συμφωνία τους με το cluster.

Το πρακτικό κέρδος εμφανίζεται όταν χρειάζεσαι ορατό ιστορικό αλλαγών, αναπαραγωγή περιβάλλοντος και κοινή διαδικασία έγκρισης. Η ευθύνη του Deployment controller παραμένει η ίδια. Ο GitOps controller μπορεί να διορθώνει το Deployment object, ενώ ο Deployment controller φροντίζει τα ReplicaSets και τα Pods.

Αν επεξεργαστείς χειροκίνητα ένα πεδίο, η επαναφορά του εξαρτάται από την πολιτική του συγκεκριμένου εργαλείου. Στο Argo CD η αυτόματη διόρθωση drift έχει χωριστή ρύθμιση `selfHeal`. Δεν είναι σωστό να θεωρείς ότι κάθε εγκατάσταση με auto sync σβήνει αμέσως κάθε χειροκίνητη αλλαγή. Η διάκριση φαίνεται στο Automated Sync Policy.

Η σύνδεση με το κεφάλαιο 7 είναι άμεση. Ποιος κατέχει το πεδίο και ποιος μπορεί να το ξαναγράψει; Αν η κανονική πηγή βρίσκεται στο Git, μια προσωρινή εντολή `kubectl` χρειάζεται συνεννόηση με αυτή τη ροή. Αλλιώς διορθώνεις τη ζωντανή τιμή και αφήνεις τη δηλωμένη αιτία στη θέση της.

Operators δίπλα στην ειδική γνώση

Ένας operator χρησιμοποιεί custom resources και controllers για μια συγκεκριμένη περιοχή. Αντί να δηλώνεις μόνο γενικά Pods, μπορεί να δηλώνεις ένα αντικείμενο που περιγράφει ένα σύστημα με δικούς του κανόνες λειτουργίας. Ο controller μεταφράζει αυτή τη δήλωση σε ενέργειες και παρατηρεί την πρόοδο.

Σχήμα 24.2 · Ο operator προσθέτει ειδική γνώση πάνω στο ίδιο API. Η εγκατάσταση ενός CRD ορίζει τον νέο τύπο. Ο controller είναι εκείνος που εφαρμόζει τη συμπεριφορά.

Για μια βάση, η αξία μπορεί να βρίσκεται σε αυτοματισμούς backup, αλλαγής έκδοσης ή διαχείρισης replicas. Το ότι ένα πακέτο ονομάζεται operator δεν αποδεικνύει ότι προσφέρει όλα αυτά ή ότι τα εκτελεί με τον τρόπο που χρειάζεσαι. Διάβασε τις συγκεκριμένες εγγυήσεις, τις προϋποθέσεις αποθήκευσης και τη διαδικασία επαναφοράς.

Το κεφάλαιο 13 έδειξε γιατί ένα PVC δεν είναι backup και ένα StatefulSet δεν είναι πλήρης αρχιτεκτονική βάσης. Ένας operator μπορεί να αναλάβει μέρος αυτής της επιπλέον δουλειάς. Χρειάζεται όμως δικό του πείραμα αποτυχίας και restore. Το γενικό μοτίβο περιγράφεται στο Operator pattern.

Το CRD μόνο του δεν δημιουργεί αυτή τη συμπεριφορά. Αν ο controller λείπει, το API μπορεί να δέχεται τον νέο τύπο χωρίς να συμβαίνει η αναμενόμενη ενέργεια. Είναι το ίδιο ερώτημα που έκανες στο κεφάλαιο 5, τώρα για ένα αντικείμενο ειδικού σκοπού.

Service mesh δίπλα στην κίνηση

Το Gateway ήταν το σημείο εισόδου στο παράδειγμα. Ένα service mesh ασχολείται ευρύτερα με την επικοινωνία μεταξύ workloads. Ανάλογα με την υλοποίηση και τη ρύθμιση, μπορεί να προσθέσει ταυτότητα υπηρεσιών, mTLS, πολιτικές κίνησης και τηλεμετρία σε αυτό το επίπεδο. Το What is Istio? δίνει ένα συγκεκριμένο παράδειγμα αυτής της περιοχής.

Αν έχεις δύο διεργασίες και ένα πρόβλημα selector, το mesh δεν απαντά στην πρώτη σου ανάγκη. Αν έχεις πολλές υπηρεσίες και πρέπει να εφαρμόσεις κοινές πολιτικές επικοινωνίας, τότε υπάρχει πιο συγκεκριμένο ερώτημα προς αξιολόγηση. Μαζί με την επιπλέον δυνατότητα προστίθενται components, ρυθμίσεις και νέα σημεία διάγνωσης.

Θα χρειαστείς ξανά το κεφάλαιο 10 για τη βασική διαδρομή και το κεφάλαιο 21 για τη σειρά ελέγχου. Ένα επιπλέον proxy δεν κάνει μια αποτυχημένη SQL συναλλαγή επιτυχημένη και δεν κάνει μια εφαρμογή αυτόματα ανθεκτική σε επανάληψη αιτήματος. Η σημασιολογία των retries πρέπει να συμφωνεί με τη σημασιολογία της εφαρμογής.

Managed cluster δίπλα στην ευθύνη λειτουργίας

Στο kind είδες το control plane ως μέρος του τοπικού εργαστηρίου. Σε managed Kubernetes ένας πάροχος αναλαμβάνει συγκεκριμένα τμήματα της λειτουργίας του cluster. Το ακριβές όριο αλλάζει ανά υπηρεσία και τρόπο λειτουργίας. Δεν μεταφέρεται αυτομάτως κάθε ευθύνη της εφαρμογής στον πάροχο.

Σχήμα 24.3 · Τα επόμενα εργαλεία σε τρεις διαφορετικές θέσεις. Το mesh επηρεάζει την επικοινωνία, το managed cluster μεταφέρει συμφωνημένες ευθύνες υποδομής και η παρατήρηση κόστους καλύπτει όλο το σύστημα.

Παραμένεις υπεύθυνος για όσα δηλώνεις και για το πώς συμπεριφέρεται ο κώδικάς σου. Images, δικαιώματα εφαρμογής, probes, όρια πόρων, συμβατότητα migrations και ανάκτηση δεδομένων χρειάζονται δικές σου αποφάσεις. Το ποιος ενημερώνει nodes, λειτουργικό και πρόσθετα πρέπει να διαβαστεί στη συγκεκριμένη υπηρεσία. Το Security in Amazon EKS είναι ένα παράδειγμα ρητής κατανομής ευθυνών, όχι καθολικός πίνακας για κάθε πάροχο.

Το πρώτο εξωτερικό cluster αξίζει να αντιμετωπιστεί ως νέο περιβάλλον επαλήθευσης. Έχει διαφορετικό δίκτυο, storage classes, load balancers, δικαιώματα και πραγματικές χρεώσεις. Η

επιτυχία του kind αποδεικνύει τις παρατηρήσεις του εργαστηρίου. Δεν αποτελεί πιστοποίηση παραγωγικής ετοιμότητας σε οποιοδήποτε cloud.

Το κόστος έχει περισσότερα από ένα ρολόγια

Οι nodes και ο πιθανός λογαριασμός του control plane είναι μόνο μέρος της εικόνας. Υπάρχουν δίσκοι, snapshots, load balancers, μεταφορά δεδομένων, logs και ο χρόνος που ξοδεύεις για λειτουργία και διάγνωση. Οι συγκεκριμένες τιμές αλλάζουν και χρειάζονται υπολογισμό πάνω σε πραγματική αρχιτεκτονική.

Τα requests του κεφαλαίου 16 συνδέουν την εφαρμογή με τη δεσμευμένη χωρητικότητα. Ο HPA του κεφαλαίου 18 μπορεί να αλλάξει τα Pods που ζητάς, αλλά δεν προσθέτει από μόνος του μηχανήματα σε κάθε cluster και δεν εγγυάται μικρότερο λογαριασμό. Η κατανομή του κεφαλαίου 17 μπορεί να χρειάζεται επιπλέον χώρο για να επιβιώσει μια βλάβη. Αυτός ο χώρος έχει λόγο ύπαρξης και κόστος.

Για την επόμενη άσκηση κράτησε μία πραγματική εφαρμογή και γράψε τις ανάγκες της. Πόση διακοπή ανέχεται, τι πρέπει να αποκαθίσταται από backup, ποιος εγκρίνει ένα deploy και τι μετράς σήμερα. Μετά τοποθέτησε μόνο την προσθήκη που απαντά σε συγκεκριμένο κενό.

Το βιβλίο άρχισε με τη δήλωση «θέλω δύο αντίγραφα» και την παρατήρηση ότι λείπει ένα. Κλείνει με την ίδια συνήθεια σε μεγαλύτερη κλίμακα. Κοίτα τι ζήτησες, ποιος ανέλαβε να το κάνει και ποια απόδειξη έχεις ότι συνέβη.

ΠΑΡΑΡΤΗΜΑ

Ο πίνακας μετάφρασης

Ο πίνακας συγκεντρώνει τις αντιστοιχίσεις που χρησιμοποιήσαμε. Η τρίτη στήλη κρατά το όριο της αναλογίας. Χρησιμοποίησέ τον για να βρεις τη σωστή ερώτηση και μετά γύρισε στο αντίστοιχο κεφάλαιο.

Αυτό που ξέρεις	Το αντίστοιχο εδώ	Πού σταματά η αναλογία	Κεφάλαιο
web ή worker υπηρεσία στο compose	Deployment	Βάσεις και εργασίες που τελειώνουν χρειάζονται διαφορετική διαχείριση.	5, 9, 13, 14
δημοσιευμένο port	Service και πρόσβαση από έξω	Το προεπιλεγμένο ClusterIP είναι εσωτερικό στο cluster.	10, 11
Traefik ή nginx μπροστά	Gateway, HTTPRoute και controller	Τα αντικείμενα δηλώνουν κανόνες. Η εγκατεστημένη υλοποίηση περνά την κίνηση.	11
αρχείο env	ConfigMap και Secret	Env vars και αρχεία volume ενημερώνονται διαφορετικά.	12
κρυφές τιμές του Kamal	Secret	Το base64 δεν είναι κρυπτογράφηση και δεν κάνει το manifest κατάλληλο για Git.	12, 19
named volume	PersistentVolumeClaim	Διάρκεια ζωής και πρόσβαση από άλλα nodes εξαρτώνται από storage και πολιτικές.	13
healthcheck του compose	startup, liveness και readiness	Άλλο εκκίνηση, άλλο restart container, άλλο αποδοχή κίνησης.	9, 15
`kamal deploy`	Ενημέρωση Deployment και έλεγχος rollout	Τα migrations και η σειρά ενεργειών χρειάζονται συντονισμό.	9, 14, 22
`kamal rollback`	`kubectl rollout undo`	Επαναφέρει Pod template. Δεν αναιρεί migrations ή αλλαγές σε άλλα αντικείμενα.	9, 14
`kamal app exec`	`kubectl exec`	Επιλέγεις συγκεκριμένο Pod και container.	6
`kamal app logs`	`kubectl logs`	Επιλέγεις Pod, container και εκτέλεση. Η μακρά διατήρηση θέλει χωριστή αποθήκευση.	6, 21

Αυτό που ξέρεις	Το αντίστοιχο εδώ	Πού σταματά η αναλογία	Κεφάλαιο
`docker ps` σε server	`kubectl get pods`	Η προεπιλογή αφορά το τρέχον namespace. Το `-A` βλέπει όλα.	3, 6
αντίγραφα σε δύο servers	Replicas και κανόνες τοποθέτησης	Δύο replicas δεν εγγυώνται διαφορετικά nodes.	9, 17
systemd που ξανασηκώνει διεργασία	Kubelet για restart, controllers για αναπλήρωση	Restart container και νέο Pod είναι διαφορετικά γεγονότα.	5

Το Service δεν είναι ιδιοκτήτης των Pods. Το ConfigMap δεν είναι process environment που αλλάζει αναδρομικά. Το PVC δεν είναι υπόσχεση backup. Αν μια αναλογία σε οδηγεί σε μία από αυτές τις προβλέψεις, γύρισε στο αντίστοιχο πείραμα.

ΠΑΡΑΡΤΗΜΑ

Εντολές ανά ερώτηση

Οι εντολές υποθέτουν ότι βρίσκεσαι στον φάκελο του βιβλίου. Τα ονόματα αντιστοιχούν στο εργαστήριο. Έλεγξε πρώτα το context, ειδικά όταν έχεις κρατήσει περισσότερα clusters.

Πού κοιτάω

TERMINAL

```
export KUBECONFIG="$PWD/build/kubeconfig"
kubectl config current-context
kubectl get nodes
kubectl get pods -n k8s-book -o wide
```

Για το κεφάλαιο 17 η διαδρομή είναι `build/kubeconfig-ha`. Για το κεφάλαιο 22 είναι `build/kubeconfig-final`. Η αλλαγή namespace δεν αλλάζει cluster.

Τι ζήτησα και τι έγινε

TERMINAL

```
kubectl get deployment rails-map -o yaml
kubectl get replicasets -l app=rails-map
kubectl get pods -l app=rails-map -o wide
kubectl rollout status deployment/rails-map --timeout=180s
```

Το YAML περιέχει επιθυμητή και παρατηρημένη κατάσταση. Το `rollout status` περιμένει πρόοδο συγκεκριμένου Deployment. Δεν είναι έλεγχος όλων των εξαρτήσεών του.

Ποιο αντικείμενο είναι αυτό

TERMINAL

```
kubectl get pods -l app=rails-map \
  -o 'custom-columns=NAME:.metadata.name,UID:.metadata.uid'
kubectl get pods -l app=rails-map \
  -o 'custom-columns=OWNER:.metadata.ownerReferences[0].name'
```

Το UID διακρίνει ένα αντικείμενο από αντικαταστάτη με παρόμοιο όνομα. Ο owner δείχνει την καταγεγραμμένη ιδιοκτησία, όχι κάθε σχέση που έχει αυτό το Pod.

Πού σταμάτησε η εκκίνηση

```
TERMINAL

kubectl describe pods -l app=rails-map
kubectl get events --sort-by=.metadata.creationTimestamp
kubectl get pods -l app=rails-map -o yaml
```

Ψάξε `nodeName`, waiting reason, termination reason και conditions. Σύγκρινε τον χρόνο του event με το τρέχον rollout.

Τι έγραψε η εφαρμογή

```
TERMINAL

kubectl logs -l app=rails-map -c web \
  --prefix=true --tail=40
kubectl logs deployment/worker --tail=40
```

Για προηγούμενη εκτέλεση container πρόσθεσε `--previous` στο συγκεκριμένο Pod. Αν το Pod αντικαταστάθηκε, η εντολή δεν αποτελεί αυτόματη αναζήτηση στο παλιό UID.

Έχει το Service προορισμούς

```
TERMINAL

kubectl get service rails-map -o yaml
kubectl get pods -l app=rails-map --show-labels
kubectl get endpointslices \
  -l kubernetes.io/service-name=rails-map -o yaml
kubectl exec book-client -- ruby -rnet/http -e \
  'puts Net::HTTP.get(URI("http://rails-map/version"))'
```

Ξεχώρισε κανένα match από υπάρχοντα endpoints με `ready` false. Μετά έλεγξε το ίδιο αίτημα από τον εξωτερικό client.

Δέχτηκε ο controller τη διαδρομή

```
TERMINAL

kubectl get gateway book -o yaml
kubectl get httproute rails-map -o yaml
```

Διάβασε τις conditions μαζί με το `observedGeneration`. Έλεγξε host, listener, backend reference και τον πραγματικό κωδικό HTTP.

Υπάρχει πίεση πόρων

TERMINAL

```
kubectl top pods
kubectl get hpa rails-map
kubectl describe hpa rails-map
kubectl describe nodes
```

Οι δύο πρώτες περιοχές μετρήσεων χρειάζονται Metrics API και εγκατεστημένο HPA όπου αναφέρεται. Ένα Pending Pod μπορεί να περιμένει χωρητικότητα παρότι ο HPA ζητά περισσότερα replicas.

Επιτρέπεται η συγκεκριμένη πράξη

TERMINAL

```
kubectl auth can-i get pods -n k8s-book \
  --as=system:serviceaccount:k8s-book:book-reader
kubectl auth can-i delete pods -n k8s-book \
  --as=system:serviceaccount:k8s-book:book-reader
```

Στο αρχικό Role η πρώτη απάντηση είναι `yes` και η δεύτερη `no`. Το `--as` στο εργαστήριο χρησιμοποιεί την ικανότητα impersonation του τοπικού διαχειριστή. Δεν αποκτά κάθε χρήστης το δικαίωμα να υποδύεται άλλον.

ΠΑΡΑΡΤΗΜΑ

Μικρό γλωσσάρι

Οι ελληνικές αποδόσεις βοηθούν στην ανάγνωση. Τα ονόματα API και τα πεδία μένουν στα αγγλικά για να τα βρίσκεις στις εντολές και στην τεκμηρίωση.

Στο κείμενο	Όρος	Τι σημαίνει εδώ
επιθυμητή κατάσταση	desired state	Αυτό που δηλώνει ο υπεύθυνος writer στα αντικείμενα API.
παρατηρημένη κατάσταση	observed state	Αυτό που αναφέρουν τα components μετά από παρατήρηση.
συμφιλίωση	reconciliation	Επαναλαμβανόμενη προσπάθεια να συμφωνήσουν στόχος και παρατήρηση.
ελεγκτής	controller	Βρόχος που παρατηρεί αντικείμενα και κάνει ενέργειες για συγκεκριμένη ευθύνη.
σύμπλεγμα	cluster	Control plane και nodes που συμμετέχουν στο ίδιο Kubernetes.
κόμβος	node	Μηχάνημα ή VM εκτέλεσης Pods. Στο kind είναι container.
χώρος ονομάτων	namespace	Πεδίο ονομάτων και οριοθέτησης πολλών πόρων μέσα σε cluster.
επιλογή στόχου	context	Συνδυασμός cluster, στοιχείων χρήστη και προεπιλεγμένου namespace του client.
ετικέτα	label	Ζεύγος κλειδιού και τιμής για οργάνωση και επιλογή.
επιλογέας	selector	Κριτήριο που βρίσκει αντικείμενα από labels.
ιδιοκτήτης	owner reference	Καταγεγραμμένη σχέση εξαρτώμενου αντικειμένου με ιδιοκτήτη.
αναγνωριστικό	UID	Ταυτότητα συγκεκριμένου αντικειμένου κατά τη διάρκεια της ζωής του.
πρότυπο Pod	Pod template	Η περιγραφή από την οποία ένας controller δημιουργεί Pods.
αντίγραφο	replica	Ένα μέλος του ζητούμενου πλήθους workload.

Στο κείμενο	Όρος	Τι σημαίνει εδώ
κυλιόμενη ενημέρωση	rolling update	Σταδιακή αντικατάσταση με ελεγχόμενη συνύπαρξη παλιών και νέων Pods.
ετοιμότητα	readiness	Αν το Pod είναι έτοιμο για τη χρήση που περιγράφει ο έλεγχος.
έλεγχος ζωής	liveness probe	Έλεγχος του kubelet που μπορεί να προκαλέσει restart container.
έλεγχος εκκίνησης	startup probe	Έλεγχος που προηγείται της κανονικής λειτουργίας των άλλων probes.
αίτημα πόρων	request	Ποσότητα που χρησιμοποιείται στην τοποθέτηση και σε άλλους υπολογισμούς.
όριο πόρων	limit	Όριο που επιβάλλεται στην εκτέλεση με διαφορετικό τρόπο ανά πόρο.
σημείο προορισμού	endpoint	Διεύθυνση και σχετικές πληροφορίες προορισμού μιας υπηρεσίας.
αίτημα αποθήκευσης	PVC	Δήλωση απαίτησης persistent storage που συνδέεται με παροχή τόμου.
μόνιμος τόμος	PV	Αντικείμενο που περιγράφει παρεχόμενη persistent αποθήκευση.
ελεγχόμενη απομάκρυνση	eviction	Αίτημα απομάκρυνσης Pod μέσω ειδικού API με σχετικούς ελέγχους.
ανοχή	toleration	Άδεια να ανεχτεί το Pod συγκεκριμένο taint.
απόκλιση	drift	Διαφορά της ζωντανής κατάστασης από την πηγή που θεωρείς επιθυμητή.

Το `Running` είναι φάση Pod. Τα `CrashLoopBackOff` και `ImagePullBackOff` που εμφανίζει συχνά το kubectl είναι ενδείξεις κατάστασης container, όχι πρόσθετες φάσεις του Pod. Το `Terminating` είναι επίσης χρήσιμη ένδειξη της παρουσίασης και συνδέεται με τη διαδικασία διαγραφής.

ΠΑΡΑΡΤΗΜΑ

YAML για αντιγραφή

Τα πραγματικά αρχεία βρίσκονται δίπλα στο βιβλίο. Το παρακάτω ζεύγος είναι πλήρες ελάχιστο web και Service, για cluster που έχει ήδη το namespace και την εικόνα v1. Το web λειτουργεί χωρίς βάση. Τα τελικά αρχεία της εφαρμογής με Postgres και Redis βρίσκονται στον πίνακα που ακολουθεί.

manifests/ch09/web.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: rails-map
  namespace: k8s-book
spec:
  replicas: 2
  revisionHistoryLimit: 5
  progressDeadlineSeconds: 90
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
  selector:
    matchLabels:
      app: rails-map
  template:
    metadata:
      labels:
        app: rails-map
        tier: web
    spec:
      terminationGracePeriodSeconds: 30
      containers:
        - name: web
          image: k8s-book/rails-map:v1
          imagePullPolicy: IfNotPresent
          ports:
            - name: http
              containerPort: 3000
          readinessProbe:
            httpGet:
              path: /ready
              port: http
            periodSeconds: 2
            failureThreshold: 1
          resources:
            requests:
              cpu: 100m
              memory: 128Mi
            limits:
              cpu: "1"
              memory: 384Mi
```

manifests/ch10/service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: rails-map
  namespace: k8s-book
spec:
  selector:
    app: rails-map
  ports:
    - name: http
      port: 80
      targetPort: http
  type: ClusterIP
```

Το Deployment selector πρέπει να συμφωνεί με τα labels του Pod template. Ο Service selector πρέπει να επιλέγει τα ίδια web Pods. Το `targetPort` παραπέμπει στη θύρα `http` του container και όχι στη θύρα που βλέπει ο client του Service.

Ο κατάλογος της εφαρμογής

Χρειάζεσαι	Αρχεία μέσα στο `manifests/`
Ελάχιστο web και ανεξάρτητο Pod	`ch05/deployment.yaml`, `ch05/independent-pod.yaml`
Web με rolling update	`ch09/web.yaml`
Service και εσωτερικό client	`ch10/service.yaml`, `ch10/client.yaml`
Τοπικό Gateway και HTTPRoute	`ch11/gateway.yaml`, `ch11/route.yaml`
Ρύθμιση μέσω env, mount και subPath	`ch12/configmap.yaml`, `ch12/config-patch.yaml`
Δοκιμαστικό Secret σύνδεσης	`ch13/database-secret.yaml`
Postgres και Redis με PVC	`ch13/postgres.yaml`, `ch13/redis.yaml`
Worker και migrations	`ch14/worker.yaml`, `ch14/migrate-v1.yaml`, `ch14/migrate-v2.yaml`
Περιοδική συμφιλίωση εργασιών	`ch14/cronjob.yaml`
Web με τρία probes	`ch15/web.yaml`
Σκόπιμη πίεση πόρων	`ch16/pending.yaml`, `ch16/oom.yaml`
Κατανομή και PDB	`ch17/placement-patch.yaml`, `ch17/pdb.yaml`
HPA και web χωρίς δηλωμένο replicas	`ch18/hpa.yaml`, `ch18/web.yaml`

Χρειάζεσαι	Αρχεία μέσα στο `manifests/`
Περιορισμένος reader	`ch19/rbac.yaml`
Τελικό web με βάση και ρυθμίσεις	`ch22/web.yaml`

Ορισμένα αρχεία είναι εναλλακτικά στάδια του ίδιου Deployment. Μην εφαρμόσεις όλο τον κατάλογο μαζί. Το `ch09` και το `ch22`, για παράδειγμα, δηλώνουν διαφορετική λειτουργία για το ίδιο όνομα `rails-map`.

Τα δοκιμαστικά credentials του βιβλίου είναι δημόσια και αφορούν μόνο το τοπικό εργαστήριο. Για πραγματική εγκατάσταση αλλάζουν η προέλευση των μυστικών τιμών, οι δικτυακές πολιτικές, η αποθήκευση και ο τρόπος πρόσβασης. Το μικρό αρχείο είναι σημείο εκκίνησης για την έννοια που διδάσκει, όχι πλήρες production template.

ΠΑΡΑΡΤΗΜΑ

Τι ελέγχθηκε σε αυτή την έκδοση

Οι καταγραφές προέρχονται από πραγματική εκτέλεση στις 7 και 8 Σεπτεμβρίου 2026. Δεν αρκέστηκε ο έλεγχος σε επιτυχημένο `apply`. Εξετάστηκαν οι μεταβάσεις, οι σκόπιμες βλάβες και η επαναφορά τους. Τα αναλυτικά αποτελέσματα βρίσκονται στα `verification/ch01.md` έως `verification/ch24.md`, με σύντομο συγκεντρωτικό στο `VERIFICATION.md`.

Το περιβάλλον αναφοράς

Στοιχείο	Έκδοση ή συνθήκη
Mac	Apple M3 Max, 36 GiB RAM, macOS 26.5.2
Docker Engine	29.4.0, 14 CPU, περίπου 7,75 GiB RAM για Docker
kind	0.33.0
Kubernetes	1.34.11, linux/arm64
kubectl	1.34.0, darwin/arm64
Rails / Ruby	8.1.3.1 / 3.4.10
Puma / Sidekiq	8.0.2 / 8.1.7
Postgres / Redis	17.11 / 8.2.9
Envoy Gateway / Gateway API	1.9.1 / 1.6.1
Metrics Server	0.9.0
Helm / ενσωματωμένο Kustomize	4.2.4 / 5.7.1
Παραγωγή PDF	D2 0.8.2, Pandoc και Typst 0.15.1

Υπήρχαν δύο τοπολογίες. Μία με έναν control plane node για τα μικρά πειράματα και μία με έναν control plane και τρεις workers για την απώλεια node. Το τελικό κεφάλαιο έστησε νέα ανεξάρτητη εγκατάσταση της μικρής τοπολογίας. Όλες οι εγκαταστάσεις μοιράζονταν το ίδιο Docker Desktop. Άλλα υπάρχοντα workloads του Mac συνέχισαν να τρέχουν.

Τα images Ruby, Kubernetes, Postgres και Redis έχουν συγκεκριμένα digests στα αρχεία. Οι ακριβείς runtime image IDs και οι εκδόσεις διατηρούνται στο `verification/evidence/`. Δεν έγινε runtime έλεγχος σε amd64.

Το πρώτο build

Το πρώτο build σε απομονωμένο builder με 0B cache πήρε **77,76 δευτερόλεπτα**. Το τελικό Dockerfile περιόρισε τα εργαλεία μεταγλώττισης σε όσα χρειάζονται τα gems και χρησιμοποίησε οκτώ παράλληλες εργασίες Bundler. Σε άλλον νέο builder με άδεια cache ολοκληρώθηκε σε **57,43 δευτερόλεπτα**. Το επόμενο build με ίδια cache και ίδιες πηγές πήρε **1,72 δευτερόλεπτα**.

Η μέτρηση περιλαμβάνει λήψη Ruby base image, apt packages και gems, δημιουργία της εικόνας και φόρτωσή της στο τοπικό Docker. Δεν περιλαμβάνει τη χωριστή εκκίνηση του BuildKit builder. Το δίκτυο ήταν η διαθέσιμη σύνδεση του Mac προς τα δημόσια registries και repositories. Δεν μετρήθηκε ανεξάρτητα το bandwidth. Το όριο του ενός λεπτού επιτεύχθηκε σε αυτή την καταγεγραμμένη συνθήκη και δεν αποτελεί εγγύηση για κάθε σύνδεση.

Η νέα εικόνα απάντησε στα `/version`, `/live` και `/ready`. Το Puma επιβεβαιώθηκε ως PID 1. Η εφαρμογή παραμένει χωρίς asset pipeline, με τέσσερα άμεσα gems στο Gemfile.

Οι παρατηρήσεις ανά ομάδα κεφαλαίων

Κεφάλαια	Τι αποδείχθηκε στο εργαστήριο
1–4	Δημιουργία clusters, σωστό context, nodes και διαχωρισμός ευθυνών με πραγματικά API objects.
5	Managed Pod αντικαταστάθηκε με νέο UID. Ανεξάρτητο Pod δεν αναπληρώθηκε. Restart container κράτησε UID. Η άσκηση τριών replicas πέρασε.
6–8	Εικόνα και HTTP λειτούργησαν. Ανύπαρκτο image απέτυχε. SSA conflict διατήρησε την πρώτη τιμή. Οι δύο selectors έδωσαν δύο και μηδέν Pods.
9–10	Readiness σταμάτησε rollout και undo επανέφερε λειτουργία. Το Service κράτησε ClusterIP. Ξεχώρισαν κενά endpoints από δύο μη έτοιμα.
11–12	HTTP, επαληθευμένο TLS και επαναφορά route πέρασαν. ConfigMap άλλαξε κανονικό mount, ενώ env και subPath περίμεναν νέο Pod.
13–14	PVC και εγγραφή επέζησαν αντικατάστασης Postgres Pod. Dump επανήλθε σε νέα βάση. Worker ολοκλήρωσε ενεργό job σε graceful shutdown. Migration failure σταμάτησε release.
15–16	Liveness προκάλεσε restart. Διόρθωση σταμάτησε τον βρόχο. Υπερβολικό request εμπόδισε scheduling. Υπέρβαση μνήμης έδωσε OOMKilled.

Κεφάλαια	Τι αποδείχθηκε στο εργαστήριο
17–18	Χαμένος worker οδήγησε σε αναπλήρωση αλλού. PDB περιόρισε drain. HPA αύξησε replicas σε τέσσερα και επέστρεψε στα δύο.
19–21	Τα περιορισμένα δικαιώματα πέρασαν. Ελέγχθηκαν εντολές διάγνωσης και previous logs μετά από πραγματικό restart.
22	Πλήρης εγκατάσταση, v2 migration, ενεργό διπλό job, νέο processed_by και συμβατή επαναφορά v1 με δεδομένα. Οι αρχικές αστοχίες καταγράφονται στο κεφάλαιο.
23–24	Helm και Kustomize ελέγχθηκαν στο API. Τα επόμενα εργαλεία ελέγχθηκαν ως έννοιες με πρωτογενείς πηγές, χωρίς εγκατάσταση.

Οι αποτυχίες έμειναν στη μέτρηση

Στο κεφάλαιο 10 καταγράφηκαν **2 αποτυχίες σε 200 αιτήματα** κατά το rollout. Στην απώλεια worker του κεφαλαίου 17 καταγράφηκαν **15 σε 800**. Στην πρώτη συνδυαστική εκτέλεση του κεφαλαίου 22 καταγράφηκε **1 σε 524**, ενώ ένα εσωτερικό timeout διέκοψε τους τελικούς ελέγχους. Η ολοκληρωμένη τρίτη εκτέλεση είχε **0 σε 257** και πέρασε όλους τους ελέγχους δεδομένων και εργασιών.

Αυτά είναι διαφορετικά πειράματα, με διαφορετική διάρκεια και διαδρομή αιτημάτων. Δεν αθροίζονται σε κοινό ποσοστό διαθεσιμότητας της εφαρμογής. Η επιτυχία ενός rollout δεν συνεπάγεται ότι κάθε client είδε μηδενικές αποτυχίες.

Γεγονός στο πείραμα απώλειας node	Χρόνος από το stop
Node έπαψε να είναι Ready	53,87s
Χαμένο endpoint αφαιρέθηκε από τα έτοιμα	53,87s
Παλιό Pod εμφανίστηκε Terminating	352,86s
Αντικαταστάτης έγινε Ready αλλού	356,01s

Δεν συντομεύτηκαν οι συνηθισμένες tolerations των 300 δευτερολέπτων. Ο χαμένος worker επανήλθε και όλοι οι nodes έμειναν uncordoned μετά τον έλεγχο.

Τι παραμένει χωριστό

Αυτή είναι η πρώτη πλήρης γραφή του βιβλίου. Δεν έχει ακόμη διαβαστεί από ανεξάρτητο δοκιμαστικό κοινό. Η καταλληλότητα της εισαγωγής του έτοιμου Deployment στο κεφάλαιο 5 χρειάζεται και αυτόν τον ανθρώπινο έλεγχο.

Δεν έγιναν cloud deployment, HA βάσης, δοκιμή SIGKILL worker, benchmark CPU throttling ή διαδραστικό QA του προαιρετικού k9s. Η διαγραφή και επαναδημιουργία cluster αναβλήθηκε με ρητή οδηγία να παραμείνουν οι τοπικοί πόροι για τον πρωινό έλεγχο. Οι σχετικές εντολές βρίσκονται στο `CLEANUP-MORNING.md` και δεν εκτελέστηκαν κατά την παραγωγή.

Τα διαγράμματα είναι vector SVG. Η παραγωγή διαβάζει τα πραγματικά manifests, ελέγχει τους τοπικούς συνδέσμους και δημιουργεί το PDF με την εντολή `make pdf`. Το `verification/pdf-review.md` καταγράφει τον τελικό οπτικό έλεγχο και τις σελίδες του αρχείου που παραδόθηκε.