Слайд 1. Namespaces и Resource Quotas

Тема: управление изоляцией и ресурсами в Kubernetes
Цель урока: научиться разделять окружения в кластере и ограничивать потребление ресурсов на уровне namespace.

Слайд 2. Зачем нужны Namespaces

Факты:

- Логически разделяют ресурсы внутри кластера.
- Позволяют изолировать приложения разных команд или окружений.
- Помогают избежать конфликтов имён (`deployment/app` в разных ns — разные объекты).
- Позволяют создавать отдельные политики доступа (RBAC).
- Квоты работают **только** внутри namespace.

Объекты, которые не принадлежат namespace:

- Nodes
- PersistentVolume
- StorageClass
- ClusterRole / ClusterRoleBinding

Слайд 3. Основные команды работы с Namespace

```
kubectl get ns
kubectl create ns <name>
kubectl delete ns <name>
kubectl config set-context --current --namespace=<name>
```

Пояснение:

- Переключение контекста по namespace сильно ускоряет работу.
- Namespace можно удалить только если в нём нет зависших финализаторов.

Слайд 4. Типичные сценарии использования Namespaces

Сегментация по окружениям:

- `dev`, `test`, `stage`, `prod`

Сегментация по командам:

- frontend-team
- backend-team
- data-team

Технические сценарии:

- Тестирование фич в изолированном окружении.
- Ограничение доступа — один отдел не может видеть ресурсы другого.
- Ограничение ресурсов через квоты.

Слайд 5. Что такое ResourceQuota

ResourceQuota — объект, который ограничивает потребление ресурсов внутри namespace.

Ограничивает:

- Общее число Pod
- Общее число PVC
- Суммарные requests/limits CPU
- Суммарные requests/limits Memory
- Количество Service, ConfigMap, Secret (опционально)

Назначение:

- Предотвращает “захват” кластера одной командой.
- Защищает кластер от ошибочных манифестов.
- Задаёт верхние границы для namespace.

Слайд 6. Пример ResourceQuota

```yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: demo-quota
spec:
  hard:
    pods: "5"
    requests.cpu: "2"
    requests.memory: "2Gi"
    limits.cpu: "4"
    limits.memory: "4Gi"
```

Объяснение:

- В этом namespace можно создать максимум 5 pod.
- Все pod вместе могут запросить только 2 CPU и 2Gi RAM.
- Лимиты тоже ограничены сверху.

Слайд 7. Как работает ResourceQuota

При создании Pod Kubernetes проверяет:

- Хватает ли доступных `requests` CPU?
- Хватает ли доступных `requests` Memory?
- Хватает ли лимитов?
- Не превышено ли количество pod?

Если нет → ошибка `exceeded quota`.

Важно:

- Квота считается суммарно по namespace.
- Если один Pod запросил много ресурсов — остальным может не хватить.

Слайд 8. Что такое LimitRange

LimitRange — правила минимальных, максимальных и дефолтных запросов и лимитов.

Применяется к каждому контейнеру, а не всему namespace.

Зачем нужен:

- Чтобы контейнеры не создавались без ресурсов.
- Чтобы каждый контейнер имел минимальные ограничения.
- Чтобы ограничить максимальные значения на контейнер.

Слайд 9. Пример LimitRange

```yaml
apiVersion: v1
kind: LimitRange
metadata:
  name: container-limits
spec:
  limits:
  - type: Container
    default:
      cpu: "500m"
      memory: "512Mi"
    defaultRequest:
      cpu: "200m"
      memory: "256Mi"
    max:
      cpu: "1"
      memory: "1Gi"
    min:
      cpu: "100m"
      memory: "128Mi"
```

Слайд 10. ResourceQuota vs LimitRange

Объект	Контроль уровня	Что ограничивает
ResourceQuota	Namespace	Суммарные ресурсы и количество объектов
LimitRange	Под/контейнер	Минимальные/максимальные значения и дефолты

Слайд 11. Как квоты влияют на HPA

Факты:

- HPA увеличивает количество pod.
- Перед созданием нового pod проверяются квоты.
- Если квота не позволяет запустить дополнительные pod → масштабирование не происходит.

Типичная проблема:

- “HPA не скейлится, хотя CPU высокий” → квота ограничила количество pod.

Слайд 12. Практика: создание Namespace

Шаги:

1. Создать:

```
kubectl create ns demo-ns
```

2. Проверить:

```
kubectl get ns
```

3. Перейти в него:

```
kubectl config set-context --current --namespace=demo-ns
```

Слайд 13. Практика: создание ResourceQuota

`quota.yaml`:

```yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: demo-quota
spec:
  hard:
    pods: "5"
    requests.cpu: "2"
    requests.memory: "2Gi"
    limits.cpu: "4"
    limits.memory: "4Gi"
```

Команда:

```
kubectl apply -f quota.yaml
```

Проверить:

Слайд 14. Практика: создание LimitRange

`limits.yaml`:

```yaml
apiVersion: v1
kind: LimitRange
metadata:
  name: container-limits
spec:
  limits:
  - type: Container
    default:
      cpu: "500m"
      memory: "512Mi"
    defaultRequest:
      cpu: "200m"
      memory: "256Mi"
    max:
      cpu: "1"
      memory: "1Gi"
    min:
      cpu: "100m"
```

Слайд 15. Практика: тест создания Pod

Создание Pod без ресурсов:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-test
spec:
  containers:
  - name: nginx
    image: nginx
```

Проверить:

```
kubectl describe pod nginx-test
```

Ожидаемое:

- Kubernetes автоматически добавил request/limit из LimitRange.

Слайд 16. Практика: проверка ошибки квоты

Файл:

```
apiVersion: v1
kind: Pod
metadata:
  name: heavy
spec:
  containers:
  - name: heavy
    image: nginx
    resources:
      requests:
        cpu: "3"
```

Применить:

Ожидаемое:

- Ошибка превышения лимитов ResourceQuota.

Слайд 17. Итоги урока

Вы узнали:

- Как работает изоляция через namespace.
- Как ограничивать ресурсы с помощью ResourceQuota.
- Как управлять минимальными/максимальными ресурсами через LimitRange.
- Как квоты влияют на масштабирование.

Вы сделали:

- Создали namespace.
- Добавили квоту ресурсов.
- Создали LimitRange.
- Проверили работу ограничений на Pod.