Microservices with CAP

A comprehensive guide on deploying your CAP application as microservices.

Table of Contents

Create a Solution Monorepo

Assumed we want to create a composite application consisting of two or more micro services, each living in a separate GitHub repository, for example:

- https://github.com/capire/bookstore
- https://github.com/capire/reviews
- https://github.com/capire/orders

With some additional repositories, used as dependencies in the same manner, like:

- https://github.com/capire/common
- https://github.com/capire/bookshop
- https://github.com/capire/data-viewer

This guide describes a way to manage development and deployment via *monorepos* using *NPM workspaces* and *git submodules* techniques.

1. Create a new monorepo root directory using *NPM workspaces*:

   sh
   ```sh
   mkdir capire
   cd capire
   echo "{\"name\":\"@capire/samples\",\"workspaces\":[\"*\"]}" > package.js
   ```

2. Add the previously mentioned projects as `git` submodules:

   sh
   ```sh
   git init
   git submodule add https://github.com/capire/bookstore
   git submodule add https://github.com/capire/reviews
   git submodule add https://github.com/capire/orders
   git submodule add https://github.com/capire/common
   git submodule add https://github.com/capire/bookshop
   ```

```
git submodule add https://github.com/capire/data-viewer
git submodule update --init
```

Add a *.gitignore* file with the following content:

```txt
node_modules
gen
```

> The outcome of this looks and behaves exactly as the monorepo layout in *cap/samples* , so we can exercise the subsequent steps in there...

3. Test-drive locally:

```sh
npm install
```

```sh
cds w bookshop
```

```sh
cds w bookstore
```

Each microservice can be started independently. If you start each microservice, one after the other in a different terminal, the connection is already established.

↳ *Learn more about Automatic Bindings by `cds watch`*

▶ *The project structure*

Using a Shared Database

You can deploy your model to a single database and then share it across applications, if you have one of the following scenarios:

- multiple CAP applications relying on the same domain model
- a monolithic CAP application that you want to split up **on the service level only, while still sharing the underlying database layer**

In the following steps, we create an additional project to easily collect the relevant models from these projects, and act as a vehicle to deploy these models to SAP HANA in a controlled way.

▶ *Why a shared database?*

Add a Project For Shared Database

1. Add another `cds` project to collect the models from these projects:

```sh
cds init shared-db --add hana
```

```sh
npm add --workspace shared-db @capire/bookstore
npm add --workspace shared-db @capire/reviews
npm add --workspace shared-db @capire/orders
```

> Note how *NPM workspaces* allows us to use the package names of the projects, and nicely creates symlinks in *node_modules* accordingly.

2. Add a `shared-db/db/schema.cds` file as a mashup to actually collect the models:

shared-db/db/schema.cds

```cds
using from '@capire/bookstore';
using from '@capire/reviews';
using from '@capire/orders';
```

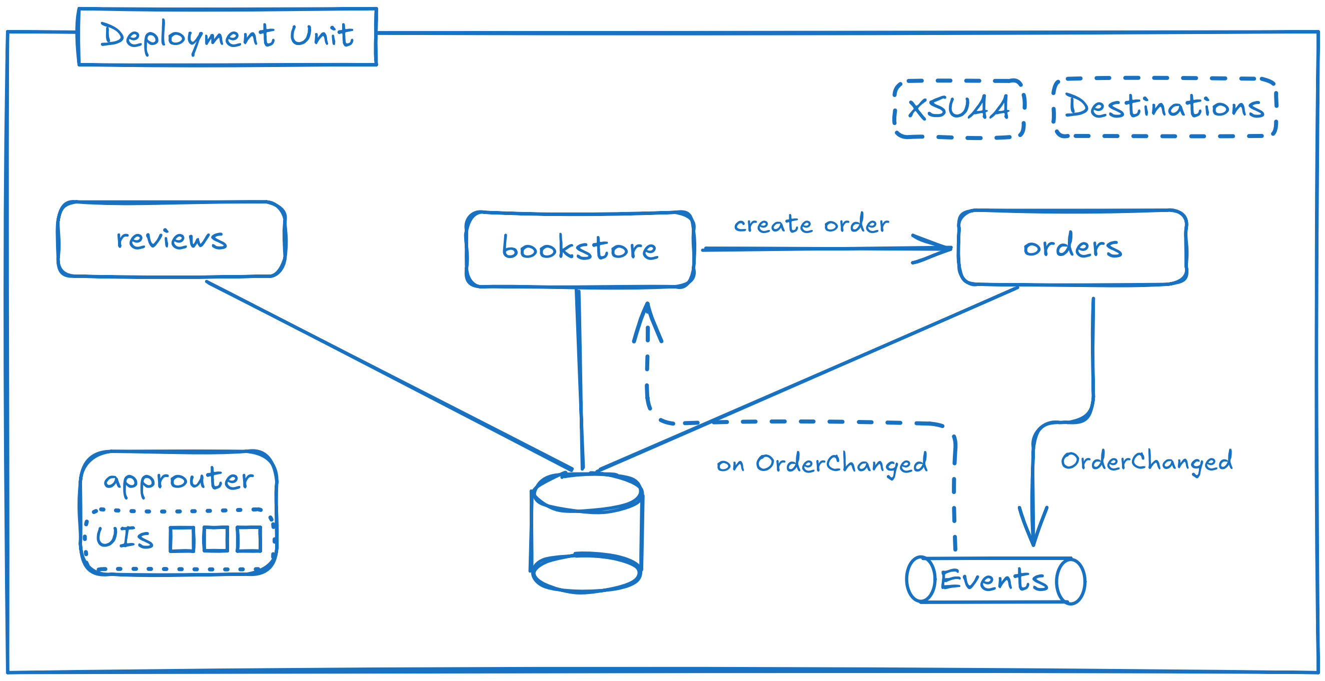
> Note: the `using` directives refer to `index.cds` files existing in the target packages. Your projects may have different entry points.

▶ *Try it out*

▶ *Other project structures*

All-in-one Deployment

This section is about how to deploy all 3+1 projects at once with a common *mta.yaml*.



@capire/samples already has an all-in-one deployment implemented. Similar steps are necessary to convert projects with multiple CAP applications into a shared database deployment.

Deployment Descriptor

Add initial multitarget application configuration for deployment to Cloud Foundry:

```shell
cds add mta
```

↳ *Learn more about **how to deploy to Cloud Foundry**.*

Database

Add initial database configuration using the command:

```shell
cds add hana
```

Delete the generated *db* folder as we don't need it on the root level:

```shell
rm -r db
```

Update the `db-deployer` path to use our `shared-db` project created previously:

mta.yaml

```yaml
    - name: samples-db-deployer
-     path: gen/db
+     path: shared-db/gen/db
```

Add build command for generation of the database artifacts:

mta.yaml

```yaml
  build-parameters:
    before-all:
      - builder: custom
        commands:
          - npm ci
-         - npx cds build --production
+         - npx cds build ./shared-db --for hana --production
```

`cds build --ws`

If the CDS models of every NPM workspace contained in the monorepo should be considered, then instead of creating this `shared-db` folder, you can also use:

```shell
cds build --for hana --production --ws
```

The `--ws` aggregates all models in the NPM workspaces.

In this walkthrough, we only include a subset of the CDS models in the deployment.

▶ *Configure each app for cloud readiness*

Applications

Replace the MTA module for `samples-srv` with versions for each CAP service and adjust `name`, `path`, and `provides[0].name` to match the module name. Also change the `npm-ci` builder to the `npm` builder.

mta.yaml

yaml

```yaml
modules:
  - name: bookstore-srv
    type: nodejs
    path: bookstore/gen/srv
    parameters:
      instances: 1
      buildpack: nodejs_buildpack
    build-parameters:
      builder: npm
    provides:
      - name: bookstore-api
        properties:
          srv-url: ${default-url}
    requires:
      - name: samples-db
      - name: samples-auth
      - name: samples-messaging
      - name: samples-destination

  - name: orders-srv
    type: nodejs
    path: orders/gen/srv
    parameters:
      instances: 1
      buildpack: nodejs_buildpack
    build-parameters:
      builder: npm
    provides:
      - name: orders-api
        properties:
          srv-url: ${default-url}
    requires:
      - name: samples-db
      - name: samples-auth
      - name: samples-messaging
      - name: samples-destination

  - name: reviews-srv
    type: nodejs
    path: reviews/gen/srv
    parameters:
      instances: 1
      buildpack: nodejs_buildpack
    build-parameters:
      builder: npm
    provides:
      - name: reviews-api
        properties:
          srv-url: ${default-url}
    requires:
      - name: samples-db
      - name: samples-auth
      - name: samples-messaging
      - name: samples-destination
...
```

Add build commands for each module to be deployed:

mta.yaml

```yaml
  build-parameters:
    before-all:
      - builder: custom
        commands:
          - npm ci
          - npx cds build ./shared-db --for hana --production
+         - npx cds build ./orders --for nodejs --production --ws-pack
+         - npx cds build ./reviews --for nodejs --production
+         - npx cds build ./bookstore --for nodejs --production --ws-pack
```

--ws-pack

Note that we use the *--ws-pack* option for some modules. It's important for node modules referencing other repository-local node modules.

Authentication

Add security configuration using the command:

```shell
cds add xsuaa --for production
```

Add the admin role

xs-security.json

```json
{
  "scopes": [
+   {
+     "name": "$XSAPPNAME.admin",
+     "description": "admin"
+   }
  ],
  "role-templates": [
+   {
+     "name": "admin",
+     "scope-references": [
+       "$XSAPPNAME.admin"
+     ],
+     "description": "cap samples multi-service shared-db"
+   }
  ]
}
```

▶ *Configure each app for cloud readiness*

Messaging

The messaging service is used to organize asynchronous communication between the CAP services.

```shell
cds add enterprise-messaging
```

Relax the publish filters for the message topics

event-mesh.json

```json
{
  ...
  "rules": {
    "topicRules": {
        "publishFilter": [
-         "${namespace}/*"
+         "*"
        ],
        "subscribeFilter": [
          "*"
        ]
      },
      "queueRules": {
        "publishFilter": [
          "${namespace}/*"
        ],
        "subscribeFilter": [
          "${namespace}/*"
        ]
      }
    }
  }
```

Parameterize the properties *emname* and *namespace* :

event-mesh.json

```json
  {
-   "emname": "samples-emname",
    "version": "1.1.0",
-   "namespace": "default/samples/1",
    ...
  }
```

mta.yaml

```yaml
resources:
  - name: samples-messaging
    type: org.cloudfoundry.managed-service
    parameters:
      service: enterprise-messaging
      service-plan: default
      path: ./event-mesh.json
+     config:
```

```
+         emname: bookstore-${org}-${space}
+         namespace: cap/samples/${space}
```

▶ *Configure each app for cloud readiness*

Destinations

Add destination configuration for connectivity between the apps:

shell
```
cds add destination
```

Add destinations that point to the API endpoints of the orders and reviews applications:

mta.yaml

yaml
```yaml
modules:
...
  - name: destination-content
    type: com.sap.application.content
    requires:
      - name: orders-api
      - name: reviews-api
      - name: bookstore-api
      - name: samples-auth
        parameters:
          service-key:
            name: xsuaa_service-key
      - name: samples-destination
        parameters:
          content-target: true
    build-parameters:
      no-source: true
    parameters:
      content:
        instance:
          existing_destinations_policy: update
          destinations:
            - Name: orders-dest
              URL: ~{orders-api/srv-url}
              Authentication: OAuth2ClientCredentials
              TokenServiceInstanceName: samples-auth
              TokenServiceKeyName: xsuaa_service-key
            - Name: reviews-dest
              URL: ~{reviews-api/srv-url}
              Authentication: OAuth2ClientCredentials
              TokenServiceInstanceName: samples-auth
              TokenServiceKeyName: xsuaa_service-key
...
```

Use the destinations in the bookstore application:

mta.yaml

```yaml
modules:
  - name: bookstore-srv
    ...
+   properties:
+     cds_requires_ReviewsService_credentials: {"destination": "reviews-dest"
+     cds_requires_OrdersService_credentials: {"destination": "orders-dest","
```

▶ *Configure each app for cloud readiness*

Approuter

Add approuter configuration using the command:

```shell
cds add approuter
```

The approuter serves the UIs and acts as a proxy for requests toward the different apps.

Since the approuter folder is only necessary for deployment, we move it into a `.deploy` folder.

```shell
mkdir .deploy
mv app/router .deploy/app-router
```

mta.yaml

```yaml
  modules:
    ...
    - name: samples
      type: approuter.nodejs
-     path: app/router
+     path: .deploy/app-router
    ...
```

Static Content

The approuter can serve static content. Since our UIs are located in different NPM workspaces, we create symbolic links to them as an easy way to deploy them as part of the approuter.

```shell
mkdir .deploy/app-router/resources
cd .deploy/app-router/resources
ln -s ../../../bookshop/app/vue bookshop
ln -s ../../../orders/app/orders orders
ln -s ../../../reviews/app/vue reviews
cd ../../..
```

> **Simplified Setup**
>
> This is a simplified setup which deploys the static content as part of the approuter. See Deploy to Cloud Foundry for a productive UI setup.

Configuration

Add destinations for each app url:

mta.yaml

```yaml
  modules:
    ...
    - name: samples
      type: approuter.nodejs
      ....
      requires:
-       - name: service-api
-         group: destinations
-         properties:
-           name: service-api
-           url: ~{srv-url}
-           forwardAuthToken: true
+       - name: orders-api
+         group: destinations
+         properties:
+           name: orders-api
+           url: ~{srv-url}
+           forwardAuthToken: true
+       - name: reviews-api
+         group: destinations
+         properties:
+           name: reviews-api
+           url: ~{srv-url}
+           forwardAuthToken: true
+       - name: bookstore-api
+         group: destinations
+         properties:
+           name: bookstore-api
+           url: ~{srv-url}
+           forwardAuthToken: true
```

The *xs-app.json* file describes how to forward incoming request to the API endpoint / OData services and is located in the *.deploy/app-router* folder. Each exposed CAP Service endpoint needs to be directed to the corresponding application which is providing this CAP service.

.deploy/app-router/xs-app.json

```json
  {
    "routes": [
-     {
-       "source": "^/(.*)$",
-       "target": "$1",
-       "destination": "srv-api",
-       "csrfProtection": true
-     }
+     {
+       "source": "^/admin/(.*)$",
+       "target": "/admin/$1",
+       "destination": "bookstore-api",
+       "csrfProtection": true
+     },
+     {
+       "source": "^/browse/(.*)$",
+       "target": "/browse/$1",
+       "destination": "bookstore-api",
+       "csrfProtection": true
+     },
+     {
+       "source": "^/user/(.*)$",
+       "target": "/user/$1",
+       "destination": "bookstore-api",
+       "csrfProtection": true
+     },
+     {
+       "source": "^/odata/v4/orders/(.*)$",
+       "target": "/odata/v4/orders/$1",
+       "destination": "orders-api",
+       "csrfProtection": true
+     },
+     {
+       "source": "^/reviews/(.*)$",
+       "target": "/reviews/$1",
+       "destination": "reviews-api",
+       "csrfProtection": true
+     }
    ]
  }
```

Add routes for static content:

.deploy/app-router/xs-app.json

```json
  {
    "routes": [
      ...
+     {
+       "source": "^/app/(.*)$",
+       "target": "$1",
+       "localDir": "resources",
+       "cacheControl": "no-cache, no-store, must-revalidate"
+     }
```

```
    ]
  }
```

The *`/app/*`* route exposes our UIs, so bookstore is available as *`/app/bookstore`* , orders as *`/app/orders`* and reviews as *`/app/reviews`* . Due to the *`/app`* prefix, make sure that static resources are accessed via relative paths inside the UIs.

Add the *`bookshop/index.html`* as initial page when visiting the app:

.deploy/app-router/xs-app.json

```json
  {
+   "welcomeFile": "app/bookshop/index.html",
    "routes": {
      ...
    }
  }
```

Additionally, the welcomeFile is important for deployed Vue UIs as they obtain CSRF-Tokens via this url.

Deploy

Before deploying you need to log in to Cloud Foundry: *`cf login --sso`*

Start the deployment and build process:

```sh
cds up
```

↳ *Learn more about `cds up` .*

Once the app is deployed, you can get the url of the approuter via

```shell
cf apps
```

```
name                          requested state    processes    routes
bookstore-srv                 started            web:1/1      my-capire-bookstor
orders-srv                    started            web:1/1      my-capire-orders-s
reviews-srv                   started            web:1/1      my-capire-reviews-
```

```
samples                       started         web:1/1     my-capire-samples.
samples-db-deployer           stopped         web:0/1
```

You can then navigate to this url and the corresponding apps

text

```
<url>/                    -> bookshop
<url>/app/bookshop        -> bookshop
<url>/app/orders          -> orders
<url>/app/reviews         -> reviews
```

Deployment as Separate MTA

This is an alternative to the all-in-one deployment. Assume the applications each already have their own *mta.yaml*. For example by running `cds add mta` in the *reviews*, *orders* and *bookstore* folder.

Database

We can add the previously created `shared-db` project as its own MTA deployment:

shared-db/

sh

```sh
cds add mta
```

This adds everything necessary for a full CAP application. Since we only want the database and database deployment, remove everything else like the srv module and destination and messaging resources:

▶ *Diff*

Binding to shared database

The only thing left to care about is to ensure all 3+1 projects are bound and connected to the same database at deployment, subscription, and runtime.

Configure the *mta.yaml* of the other apps to bind to the existing shared database, for example, in the reviews module:

```yaml
...
modules:
  ...

-   - name: reviews-db-deployer
-     type: hdb
-     path: gen/db
-     parameters:
-       buildpack: nodejs_buildpack
-     requires:
-       - name: reviews-db

resources:
  ...
  - name: reviews-db
-     type: com.sap.xs.hdi-container
+     type: org.cloudfoundry.existing-service
      parameters:
-       service: hana
-       service-plan: hdi-shared
+       service-name: shared-db-db
```

Subsequent updates

Whenever one of the projects has changes affecting the database, the database artifacts need to be deployed prior to the application deployment. With a single *mta.yaml*, this is handled in the scope of the MTA deployment. When using multiple deployment units, ensure to first deploy the `shared-db` project before deploying the others.

Late-Cut Microservices

Microservices have been attributed with a multitude of benefits like

- granular scalability,

- deployment agility,
- distributed development, and so on.

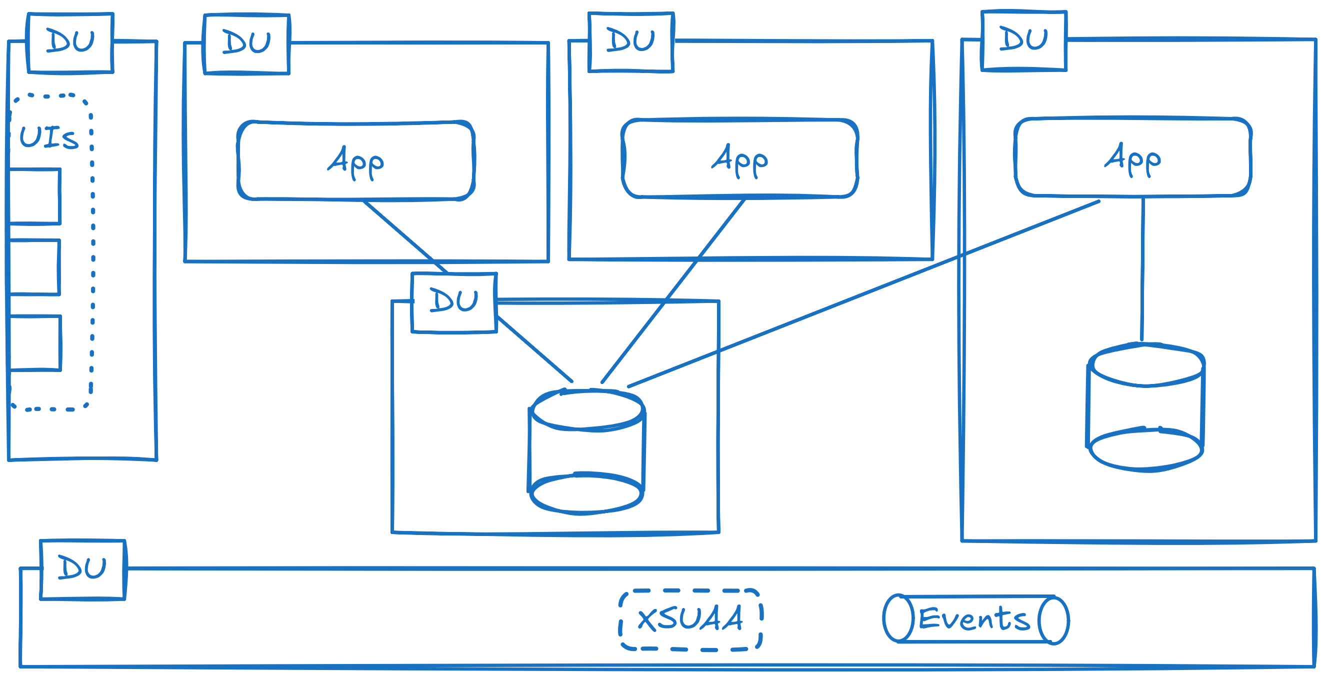
While these benefits exist, they are accompanied by complexity and performance losses. True microservices each constitute their own deployment unit with their own database. The benefits attributed to microservices can be broken down into multiple aspects.

Aspect	Benefits	Drawbacks
App Instances	Scalability, Resilience	Requires Statelessness
Modules	Distributed Development, Structure	
Applications	Independent Scalability, Fault Tolerance	Communication Overhead
Deployment Units	Faster Deploy Times, Independent Deployments	Configuration Complexity
Databases	Depends	Data Consistency, Fragmentation

Flexibility in Deployments

Instead of just choosing between a monolith and microservices, these aspects can be combined into an architecture that fits the specific product.

Since each cut not only has benefits, but also drawbacks, it's important to choose which benefits actually help the overall product and which drawbacks can be accepted.

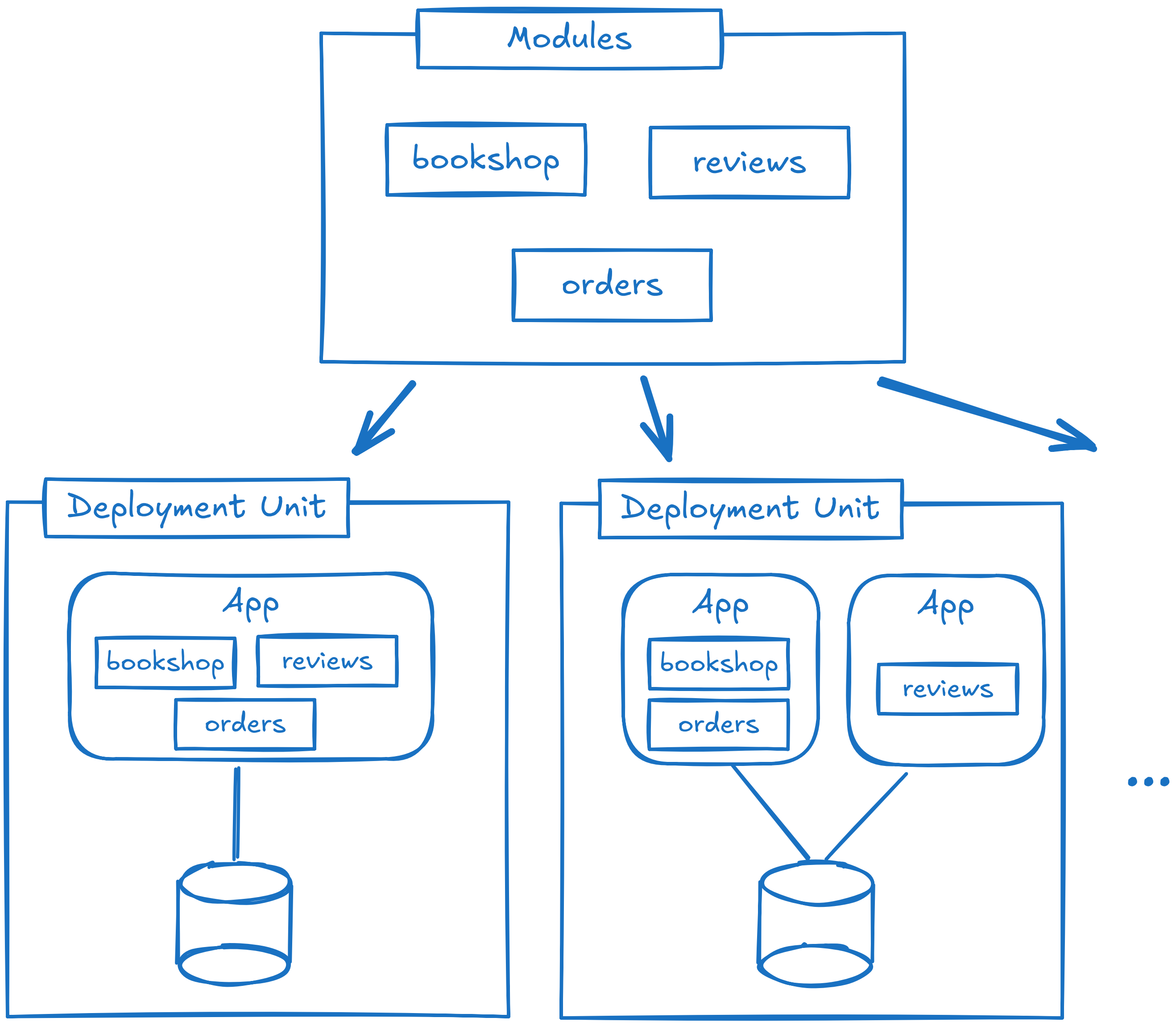


A Late Cut

When developing a product, it may initially not be apparent where the boundaries are.

Keeping this in mind, an app can be developed as a modular application with use case specific CAP services. It can first be deployed as a monolith / modulith. Once the boundaries are clear, it can then be split into multiple applications.

Generally, the semantic separation and structure can be enforced using modules. The deployment configuration is then an independent step on top. In this way, the same application can be deployed as a monolith, as microservices with a shared database, as true microservices, or a combination of these, just via configuration change.



Best Practices

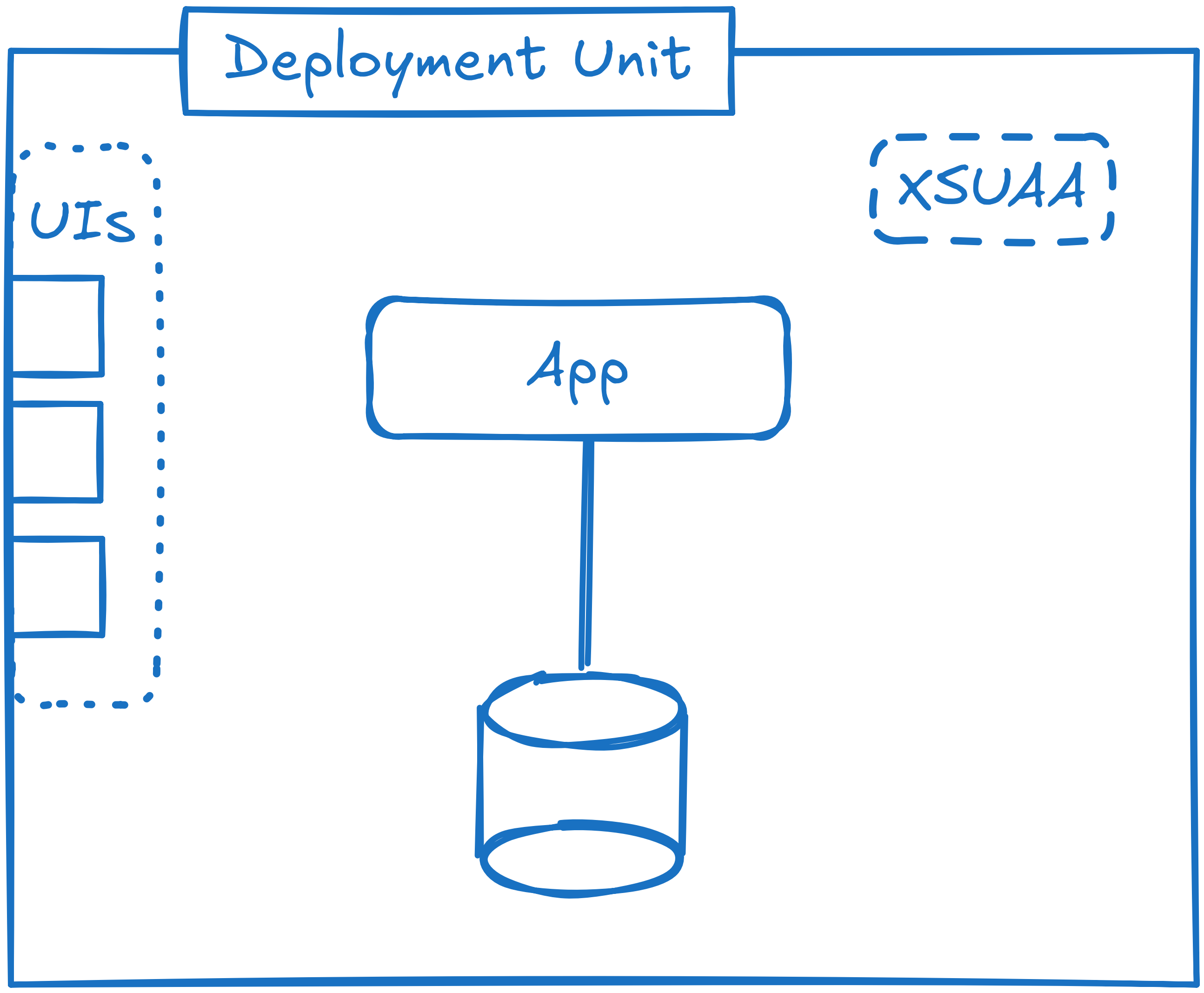
- Prefer a late cut

- Stay flexible in where to cut
- Prefer staying loosely coupled → for example, ReviewsService → reviewed events → UPDATE average ratings
- Leverage database-level integration selectively → Prefer referring to (public) service entities, not (private) database entities

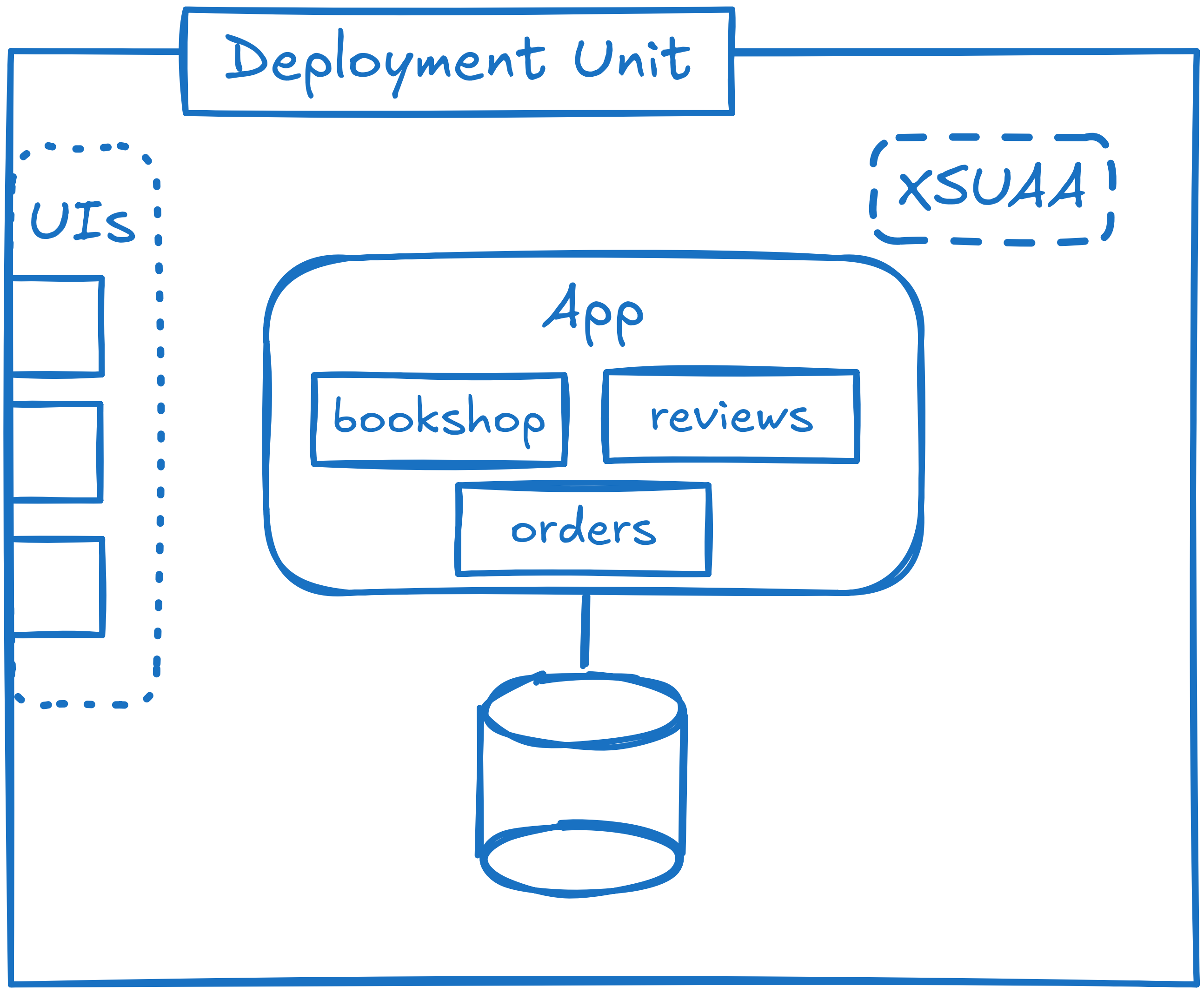
Appendix

Monolith or Microservice

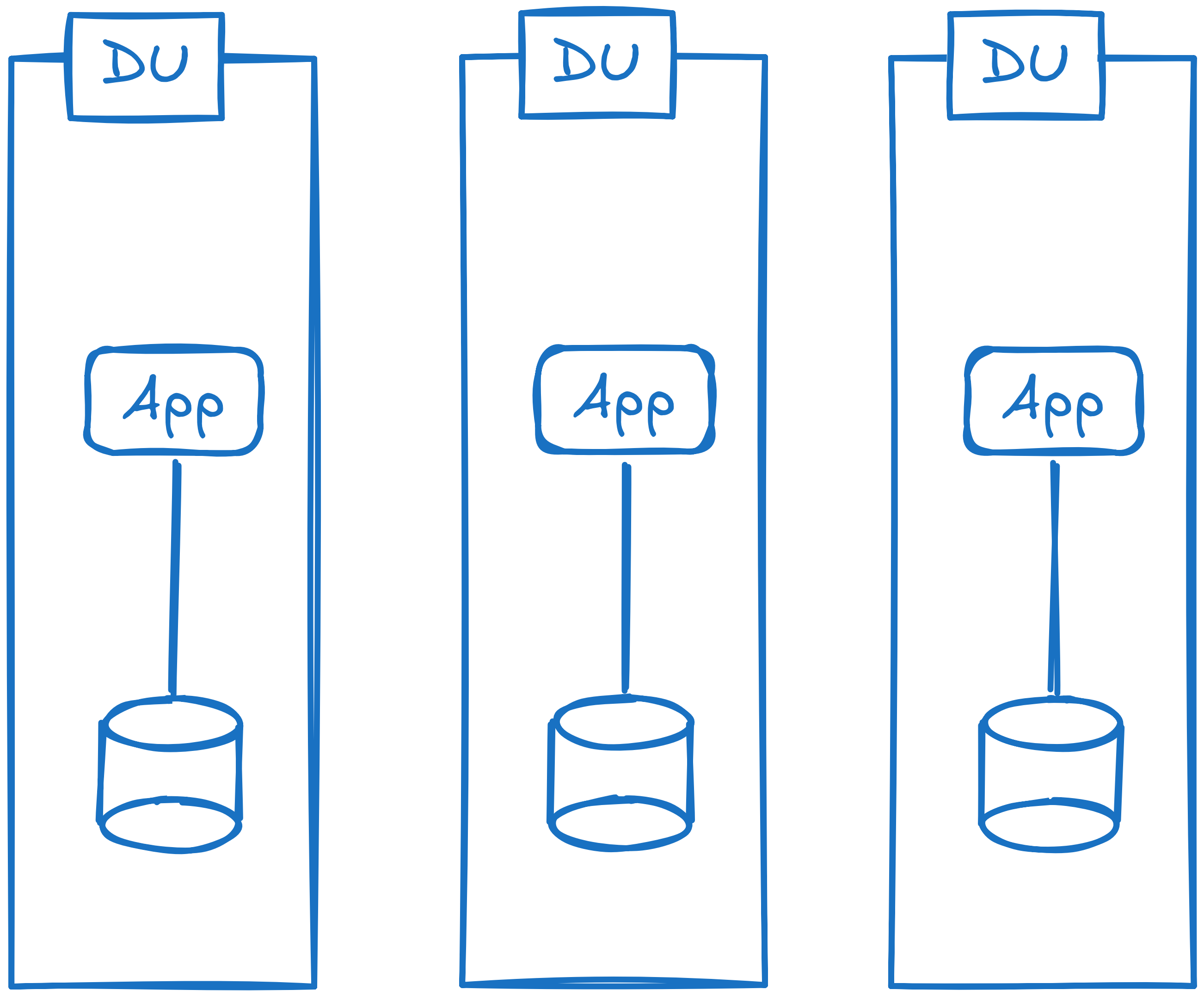
A monolith is a single deployment unit with a single application. This is very convenient, because every part of the app is accessible in memory.



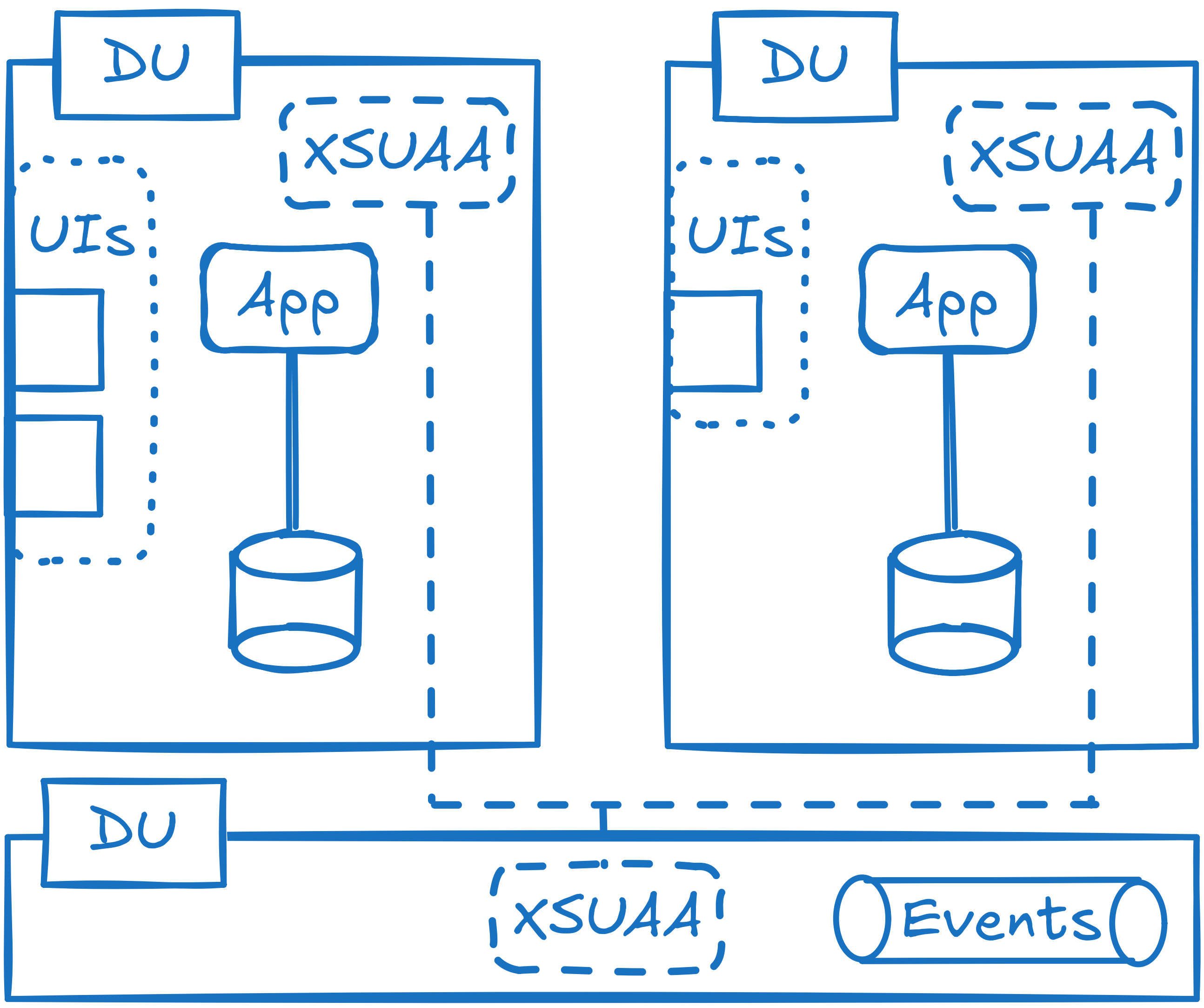
A modulith, even though the app is separated into multiple CAP services inside multiple modules, can still be deployed as a single monolithic application. This combines the benefit of a clear structure and distributed development while keeping a simple deployment.



True microservices each consist of their own deployment unit with their own application and their own database. Meaning that they're truly independent of each other. And it works well if they are actually independent.



What was mentioned earlier is a simplified view. In an actual microservice deployment, there are typically shared service instances and wiring needs to be provided so that apps can talk to each other, directly or via events. If the microservices are not cut well, the communication overhead leads to high performance losses and often the need for data replication or caching.



Application Instances

Having only a single virtual machine or container, the application can only be scaled vertically by increasing the CPU and memory resources. This typically has an upper limit and requires a restart when scaling.

To improve scalability, we can start multiple instances of the same application.

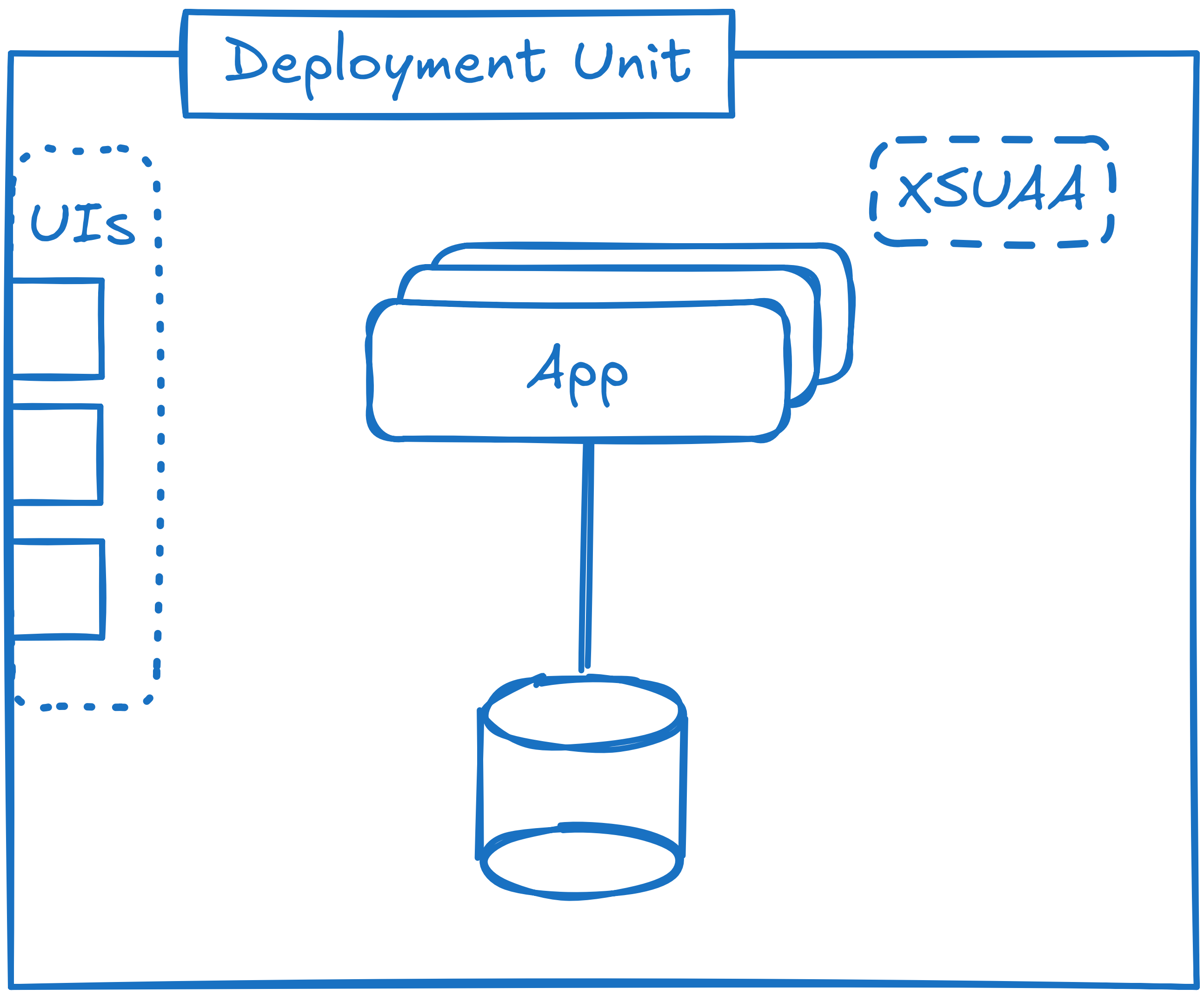
Benefits:

- Near unlimited scaling
- No downtimes when scaling
- Better resilience against failures in single app instances

Requirement:

- The app needs to be stateless, state needs to be persisted

Multiple app instances can be used for both monoliths and microservices.



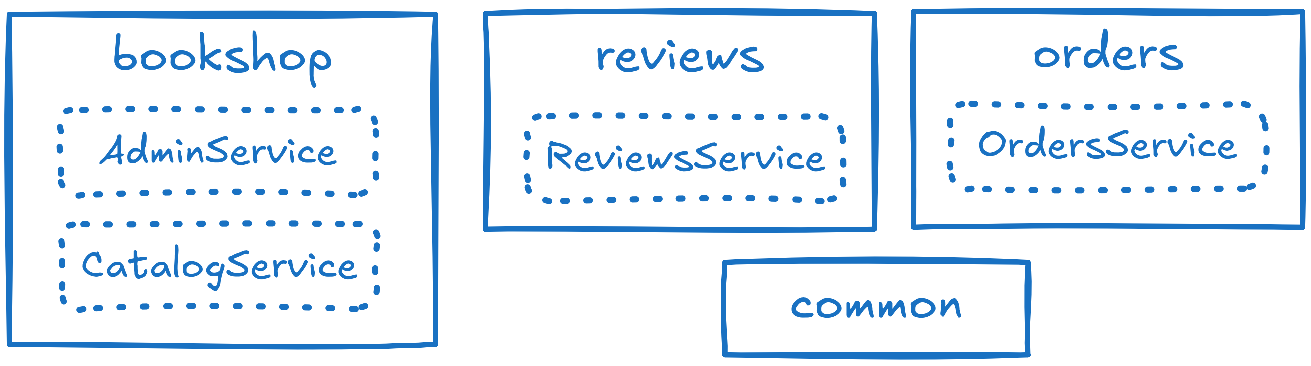
Modules

When many developers work on an app, a distribution of work is necessary. Nowadays this distribution is often reached by each team working on one or multiple microservices. Also, microservices are potentially cut by which team is developing them.

Instead, developers can work on single modules, which are later deployed and run as a single app... or as multiple apps. But this choice is then independent of who is developing the module.

Benefits:

- Distributed Development
- Clear Structure



Multiple Applications

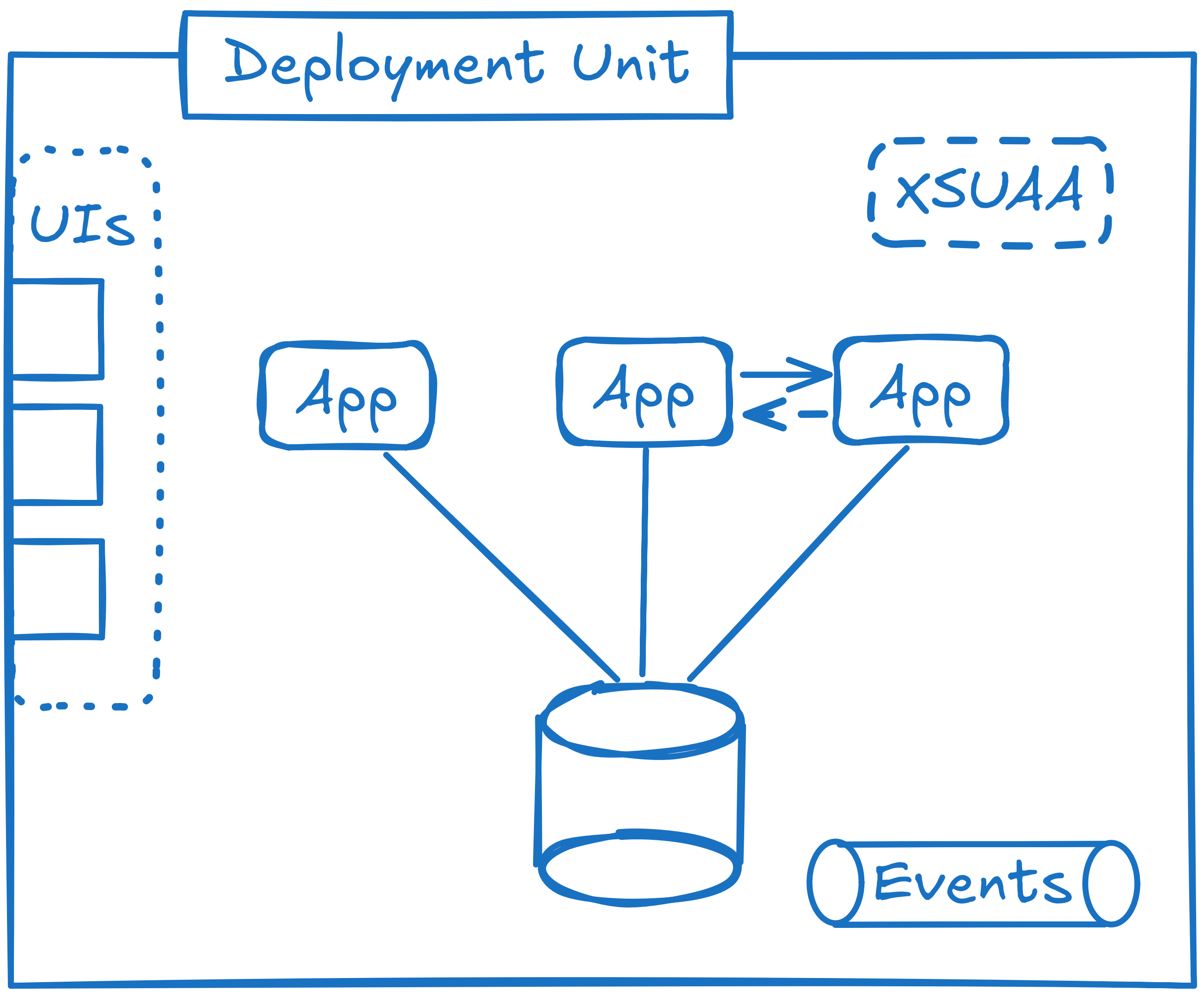
As described above, application instances already have near unlimited scaling, even for a monolith. So why would you want multiple apps?

Benefits:

- Resource Separation
- Independent Scaling
- Fault Tolerance

Drawbacks:

- Latency for synchronous calls between dependent apps



Resource Separation

One part of an application may do highly critical background processing, while another handles incoming requests. The incoming requests take CPU cycles and consume memory, which should rather be used for the background processing. To make sure that there are always enough resources for specific tasks, they can be split into their own app.

Independent Scaling

Similar to resource separation, different parts of the app may have different requirements and profiles for scaling. For some parts, a 100% CPU utilization over an extended period is accepted for efficiency, while request handling apps need spare resources to handle user requests with low latency.

Fault Tolerance

While app instances already provide some resilience, there are failure classes (for example, bugs) which affect each app instance.

Separating functionality into different apps means that when one app experiences issues, the functionality of the other apps is still available. In the bookstore example, while reviews may be down, orders may still be possible.

This benefit is null for apps with synchronous dependencies on each other. If A depends on synchronous calls to B, then if B is down, A is down as well.

Multiple Deployment Units

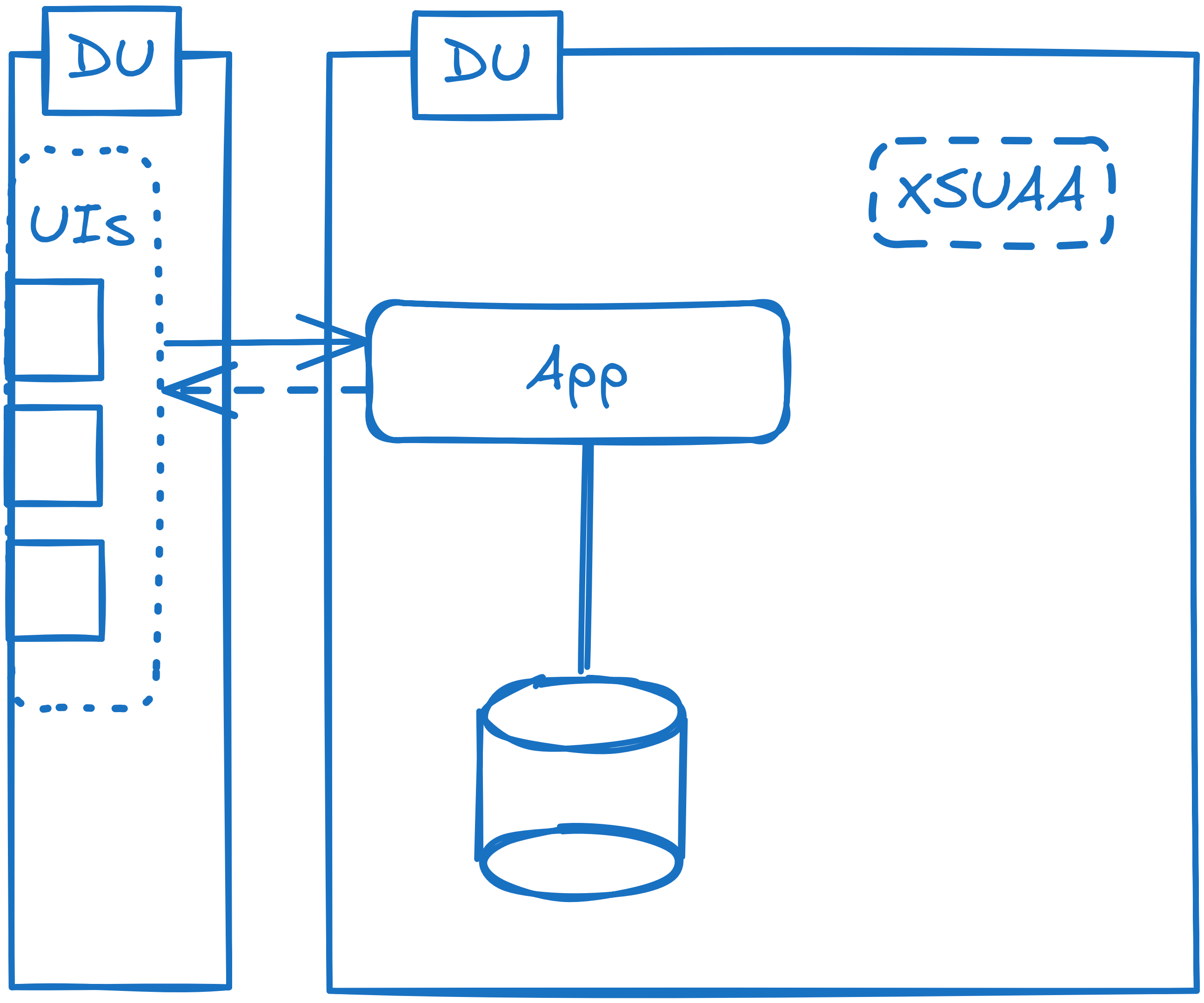
With multiple apps, you can still deploy them together as one unit, for example as part of a multitarget application archive. Once an application grows bigger, this takes a significant amount of time. Deployments can then be split up either by type (for example, deploying UIs separately) or horizontally (for example, deploying each app via its own deployment unit).

Benefits:

- Faster individual deploy times
- Independent deployments

Drawbacks:

- Coordination between deployment units for updates with dependencies
- Configuration wiring to connect systems across deployment units



With a single deployment unit, when a fix for one part needs to be deployed, the risk of redeploying the rest of the application needs to be considered. For example, there may already been changes to other parts of the app in the same code line. A restart / rolling restart may also lead to higher resource consumption due to startup activities and thus slightly degrade the performance during this time.

Being able to deploy apps or other resources independently reduces the risk when a single part of the system needs to be updated. The update decision needs less coordination and can be made by the team responsible for this part of the system.

Coordination is still necessary when deploying changes that affect the whole system, for example when a feature needs implementations in multiple apps.

Multiple Databases

Here we need to differentiate between two scenarios:

- Using multiple types of databases

- Using multiple databases of the same type

A polyglot persistence can be used when the app has different requirements for the types of data it needs to store. For example, there may be a large number of large files that can be stored in a document store, while corresponding administrative data is stored in a relational database.

Benefits:

- Use suitable technology for different use cases

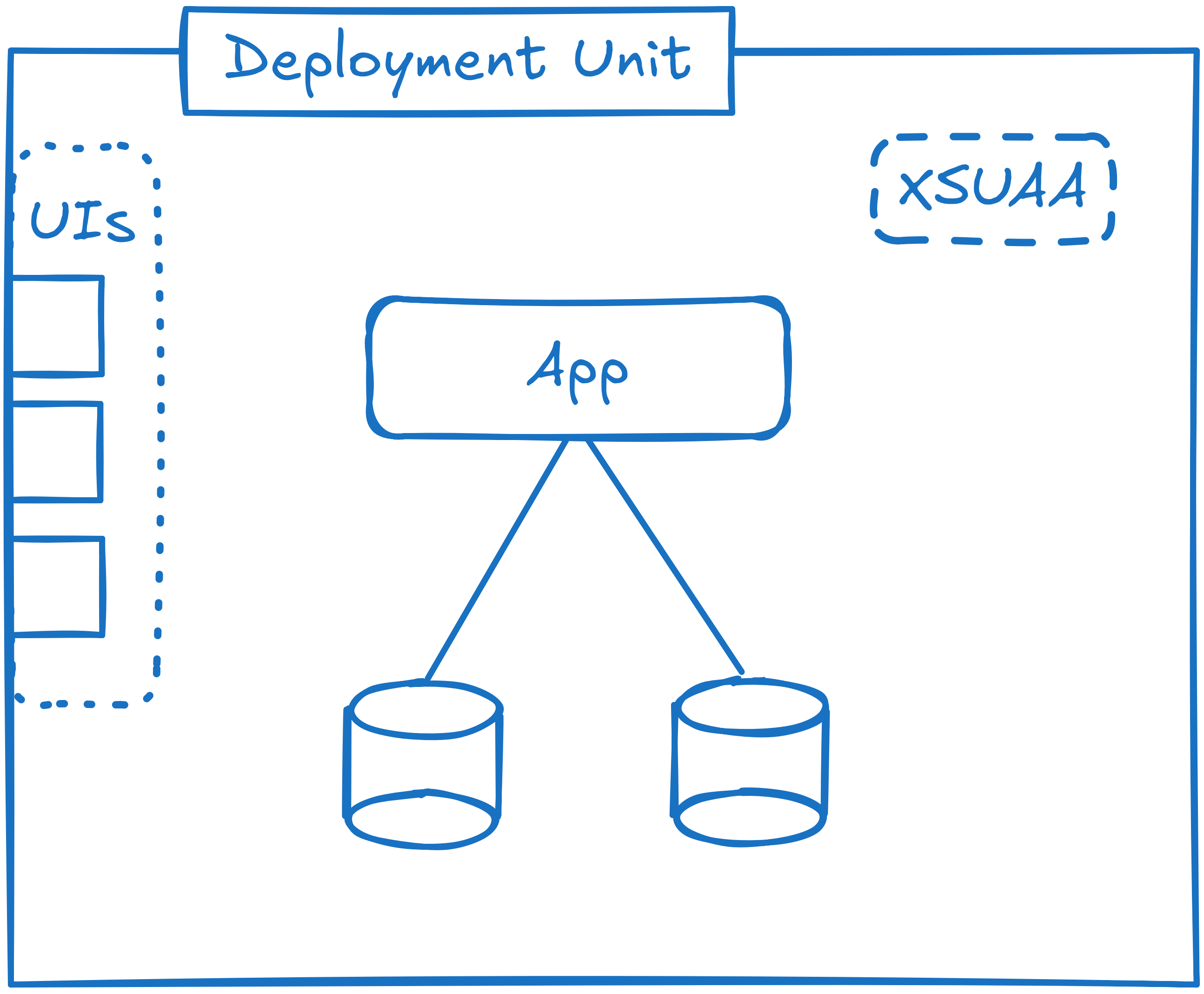
In contrast, using multiple databases of the same type may be suggested for

- Scalability
- Resource Separation
- Tenant Isolation
- Semantic Separation

Scalability and resource separation need multiple database instances. Tenant isolation and semantic separation could also be achieved through multiple schemas or containers inside the same database instance.

Drawbacks:

- Data consistency across databases
- Collecting data across databases



Data Federation

When data is distributed across multiple databases, strategies may be necessary to combine data from multiple sources.

- Fetching on demand
  - Caching
- HANA synonyms
- Data Replication