Best Practices

Based on common issues

Content

- Dependency management
- Error handling
- Transaction handling
- Database pool configuration
- Logging
- Generic handlers
- Environment
- Testing
- REPL

Issue

Suddenly my app stopped working in production because of some updates!

```json
{
  "dependencies": {
    "@sap/cds": "latest"
  }
}
```

(no `package-lock.json`)

Reason

`npm install` **determines version based on npm registry**

Manage Your Dependencies

Stay up to date - but don't break your productive app

- Use `package-lock.json` (freezes versions)
- Manually update

```
npm outdated
npm update
```

```
{
  "dependencies": {
    "@sap/cds": "^5.4.0"
  }
}
```

Semantic versioning: `major.minor.patch`

```
^5.4.0 (caret)
   ^ ^


~5.4.0 (tilde)
     ^
```

💻 Example 1

Issue

Sometimes, my app behaves in a very unexpected way!

```
const payCustomers = customers => {
  try {
    customers.forEach(c => { c.payments.paid = true })
  } catch (e) {
    console.error("Some error happened:", e)
  }
}
```

Reason

Also programming errors are caught

Do not catch programming errors

- They need to be fixed
- For unknown programming errors, the app must crash, fail loudly, then fix

```
const payCustomers = customers =>
  customers.forEach(c => {
    if (c.payments) c.payments.paid = true
  })
```

Do not program in a defensive way

```
const payCustomers = customers =>
  Array.isArray(customers) &&
  customers.forEach(c => {
    if (c && c.payments) c.payments.paid = true
  })
```

Always handle operational errors

```javascript
app.get('/Books', async (_, res) => {
  try {
    const httpResponse = await executeHttpRequest(...)
    res.send(httpResponse)
  } catch (e) {
    console.error(e)
    res.sendStatus(502) // Bad Gateway
  }
})
```

CAP's remote service API does this automatically

```
srv.on('READ', 'Books', req => extSrv.run(req.query))
```

Issue

After some point in time, my app stops working properly.

```
app.use((err, req, res, next) => {
  console.error(err)
  res.sendStatus(500) // Internal Server Error
})
```

Reason

Unexpected errors are always caught

Let your app crash when there are unexpected errors

- Restart the app automatically
- Do not leave it in a zombie state
- There might be side effects

CAP server crashes when there are programming errors

```
err instanceof TypeError ||
err instanceof ReferenceError ||
err instanceof SyntaxError ||
err instanceof RangeError ||
err instanceof URIError
```

Issue

There's an error in my app, but I can't find the root cause!

```
Error: Some error happened
    at main (/irrelevant/location.js:3:9)
```

```
try {
  mightThrow()
} catch {
  throw new Error('Some error happened')
}
```

Reason

Error information lost when re-thrown

Don't lose information when re-throwing errors

```
try {
  mightThrow()
} catch (e) {
  throw Object.assign(
    new Error('Some error happened'),
    { cause: e })
}
```

```
Error: Some error happened
    at main (/irrelevant/location.js:3:9)
  cause: Error: This is the original error
      at: importantFunction (/important/location.js:3:9)
```

There's also a new feature in V8 v9.3:

```
throw new Error('Some error happened', { cause: e })
```

Issue

After the first request, my app doesn't respond anymore!

```
srv.on('READ', 'Books', req => cds.tx()
    .run(SELECT.from('Books')))
```

Reason

`cds.tx()` **starts a new (unmanaged) transaction**

Bind your transaction to the request

```
srv.on('READ', 'Books', req => cds.tx(req)
    .run(SELECT.from('Books')))
```

- Our database operations start with `BEGIN` and must end with `COMMIT`/`ROLLBACK`
- In `SQLite`, there are no parallel transactions
- `cds.tx(req)` automatically performs a `COMMIT` once the request is succeeded or `ROLLBACK` if it fails

Under the hood (simplified)

```
start of request
       |
       |-- database interaction
       |   BEGIN
       |   on('succeeded', () => tx.commit())
       |   on('failed', () => tx.rollback())
       |   SELECT FROM BOOKS
       |
       |
       |
succeeded/failed -- emit('succeded'/'failed')
```

💻 Example 2

You can also use `AsyncLocalStorage`

```
srv.on('READ', 'Books', () => SELECT.from('Books'))
```

- Information about the current transaction is saved in `cds.context`
- It's not a global variable - it's local w.r.t. the async context

💻 Example 3

Issue

After the first request to my express handler, my app doesn't respond anymore!

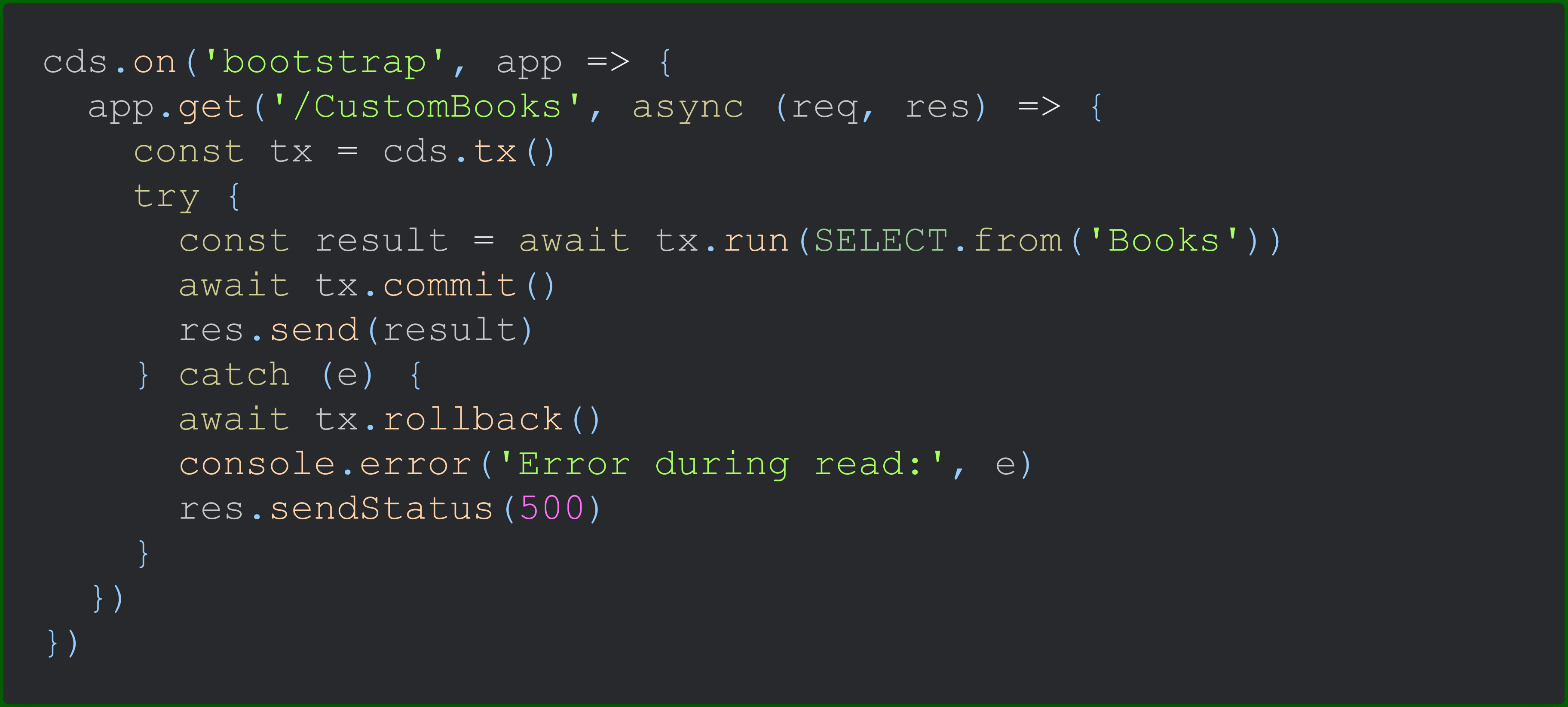
```
cds.on('bootstrap', app => {
  app.get('/CustomBooks', async (req, res) => {
    const result = await cds.tx(req).run(SELECT.from('Books'))
    res.send(result)
  })
})
```

Reason

`req` **of Express is not** `req` **of CAP**

Do not use them interchangeably

You need to use own transactions



```
cds.on('bootstrap', app => {
  app.get('/CustomBooks', async (req, res) => {
    const tx = cds.tx()
    try {
      const result = await tx.run(SELECT.from('Books'))
      await tx.commit()
      res.send(result)
    } catch (e) {
      await tx.rollback()
      console.error('Error during read:', e)
      res.sendStatus(500)
    }
  })
})
```

Alternative

```
cds.on('bootstrap', app => {
  app.get('/CustomBooks', async (req, res) => {
    try {
      const result = await cds.tx(tx =>
        tx.run(SELECT.from('Books')))
      res.send(result)
    } catch (e) {
      res.sendStatus(500) // Internal Server Error
    }
  })
})
```

Issue

Background database operations have a strange behavior!

```
const backgroundTask = async () => {
  await UPDATE("ReadCounter")
    .set({ count: { "+=": 1 } })
    .where({ ID: 'Books' })
}

srv.on("READ", "Books", (req, next) => {
  backgroundTask() // no await
  return next()
})
```

Reason

Race conditions in transaction handling

Use `cds.spawn`

```
const backgroundTask = async () => {
  await UPDATE("ReadCounter")
    .set({ count: { "+=": 1 } })
    .where({ ID: 'Books' })
}

srv.on("READ", "Books", (req, next) => {
  cds.spawn(backgroundTask)
  return next()
})
```

💻 Example 4

Issue

Some requests fail during high load!

Possible Reason

Database pool misconfiguration

Most important options:

- acquireTimeoutMillis
- max

```json
{
  "cds": {
    "requires": {
      "db": {
        "kind": "hana",
        "pool": {
          "acquireTimeoutMillis": 5000,
          "max": 1000
        }
      }
    }
  }
}
```

Issue

I can't use Kibana to analyze the logs!

```
console.log("My custom log output")
```

Use LOG with the Kibana formatter

```
const LOG = cds.log('custom')
LOG.info("My custom log output")
```

```
cds.env.features.kibana_formatter
```

💻 Example 5

Issue

How can I register generic handlers for all services?

Define an own implementation of the app service

```json
{
  "cds": {
    "requires": {
      "app-service": {
        "impl": "lib/MyAppService.js"
      }
    }
  }
}
```

```
const cds = require('@sap/cds')
const LOG = cds.log('generic')

class MyAppService extends cds.ApplicationService {
  async init() {
    await super.init()
    this.before('*', '*', req => {
      LOG.info('generic before handler is called')
    })
  }
}

module.exports = MyAppService
```

Issue

How can I easily switch my environment?

Use profiles

```json
{
  "cds": {
    "requires": {
      "[production]": {
        "auth": {
          "strategy": "JWT"
        }
      }
    }
  }
}
```

- `cds env --profile <profile>`
- `cds run --profile <profile>`

💻 Example 6

Testing

Use `cds.test` for testing

```
const project = require('path').join(__dirname, '..')
const t = cds.test(project)
```

```
test('simple test', async () => {
  const { data } = await t.GET('/catalog/Books')
  expect(data.value).toContainEqual({
    ID: 1,
    stock: 100,
    title: 'Wuthering Heights'
  })
})
```

💻 Example 7

REPL

Use `cds repl` **to play around**

💻 Example 8

Thank you

cap.cloud.sap