Micro Frontends

with Module Federation

Presented by

pcomanici

About

Comanici Paul

🔒 Home Security & Automation

📡 Telecommunications

🛒 E-Commerce

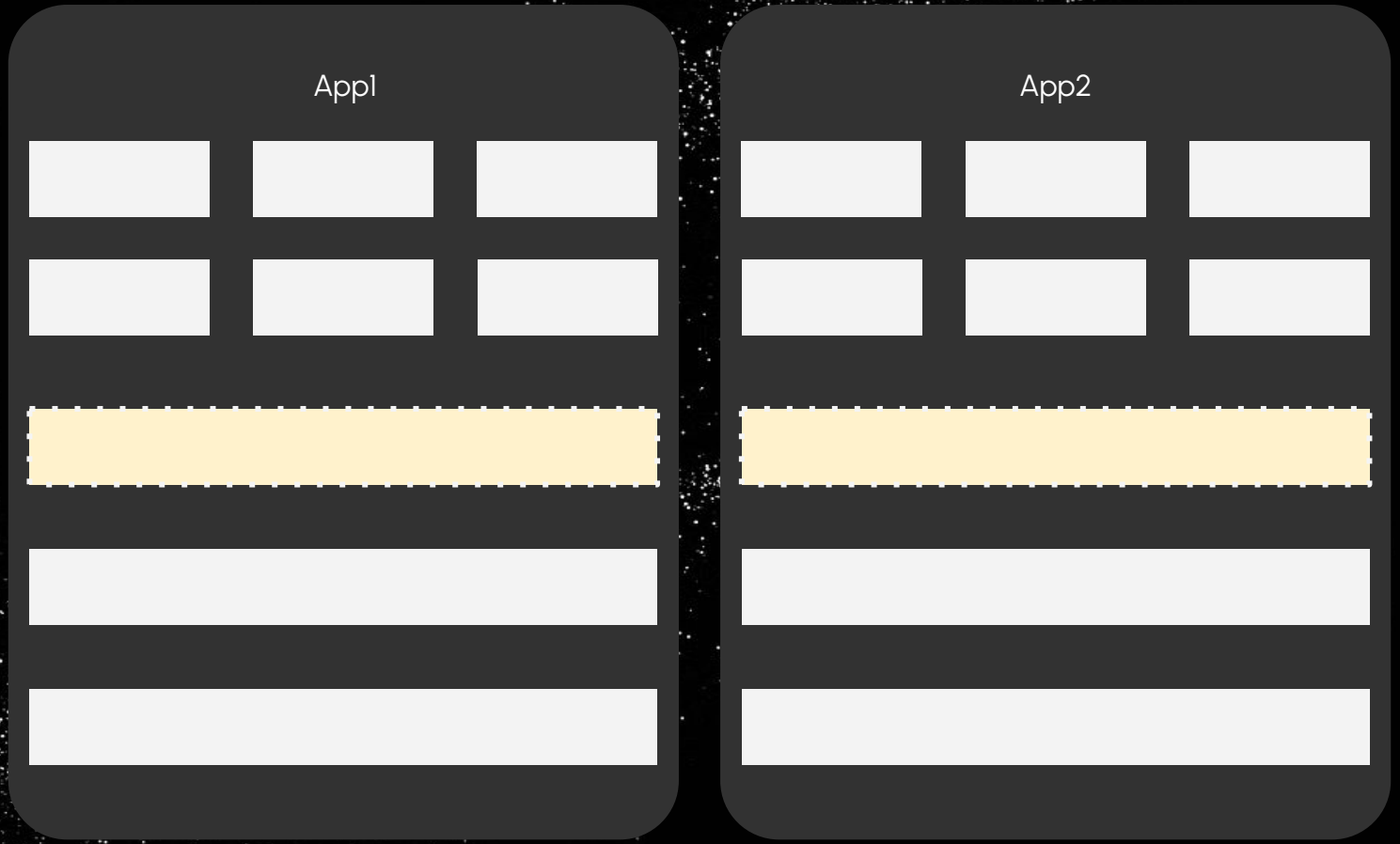
💳 Banking & Financial Services

Content

Context - sharing code

- inconvenient
- npm is slow
- grows in complexity
- sharing is usually primitive
- inefficient and unproductive to share feature code

Context - reusability



Context - existing options - native ESM

no build for linking parts

3rd party code natively supported

no tree shaking

ESM only (no css)

performance issues

Context - existing options - single build

everything is built together

external modules/components are available during the build

any changes require full deploy

slow builds, bottlenecked teams & workflows

performance issues

Context - existing options - externals

parts are built separately & exposed globally

apps are built to depend on externals

no on-demand loading

additional libraries must be created

highly dependent on external code, no fail safes

Context - existing options - DLL plugin

parts built as DLL

apps are built to depend on DLLs

additional DLLs must be created

extra infrastructure for compile time dependency

highly dependent on external code, no fail safes

Context - existing options - conclusion

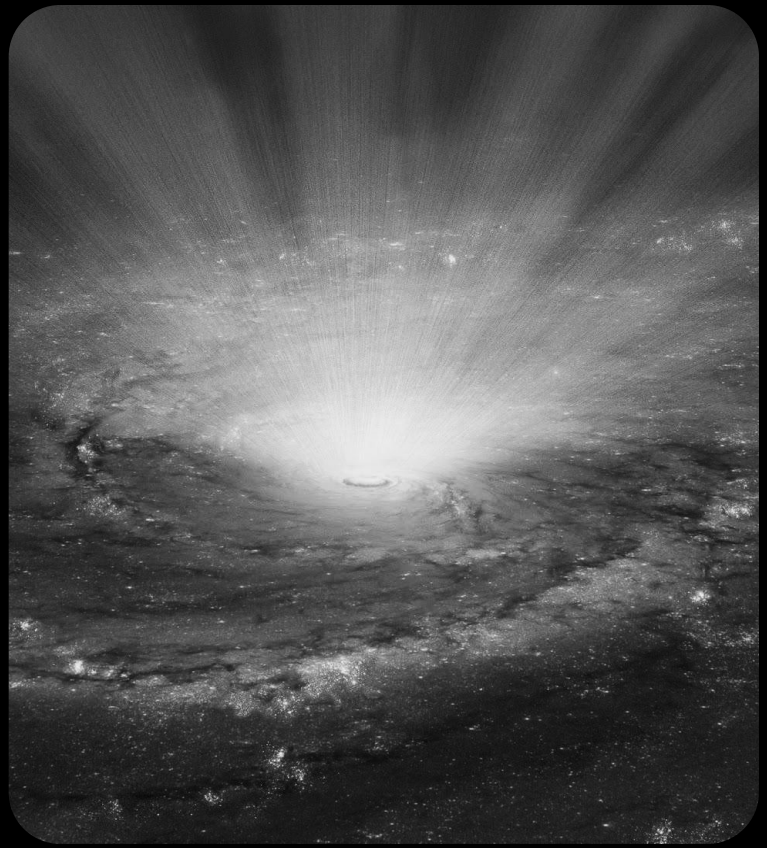
- native ESM ➢ web performance
- single build ➢ build performance
- externals ➢ manual work
- DLL plugin ➢ manual work

good build performance

good web performance

a solution for shared dependencies

Module federation allows a JavaScript application to dynamically load code from another application — in the process, sharing dependencies, if an application consuming a federated module does not have a dependency needed by the federated code — Webpack will download the missing dependency from that federated build origin.



- ★ import code from other builds, at runtime
- ★ share vendor code dynamically, at runtime
- ★ deploy independent SPAs, without needing to re-deploy consumers
- ★ redundancy and self-healing capabilities

- ★ micro-frontends work like a monolith
- ★ developer experience improved without compromising user experience
- ★ evergreen code, directly from a separate build
- ★ work in any JavaScript environment (node, browser, electron)

Terminology

1

Host

consumer

2

Remote

consumed

3

Bidirectional host

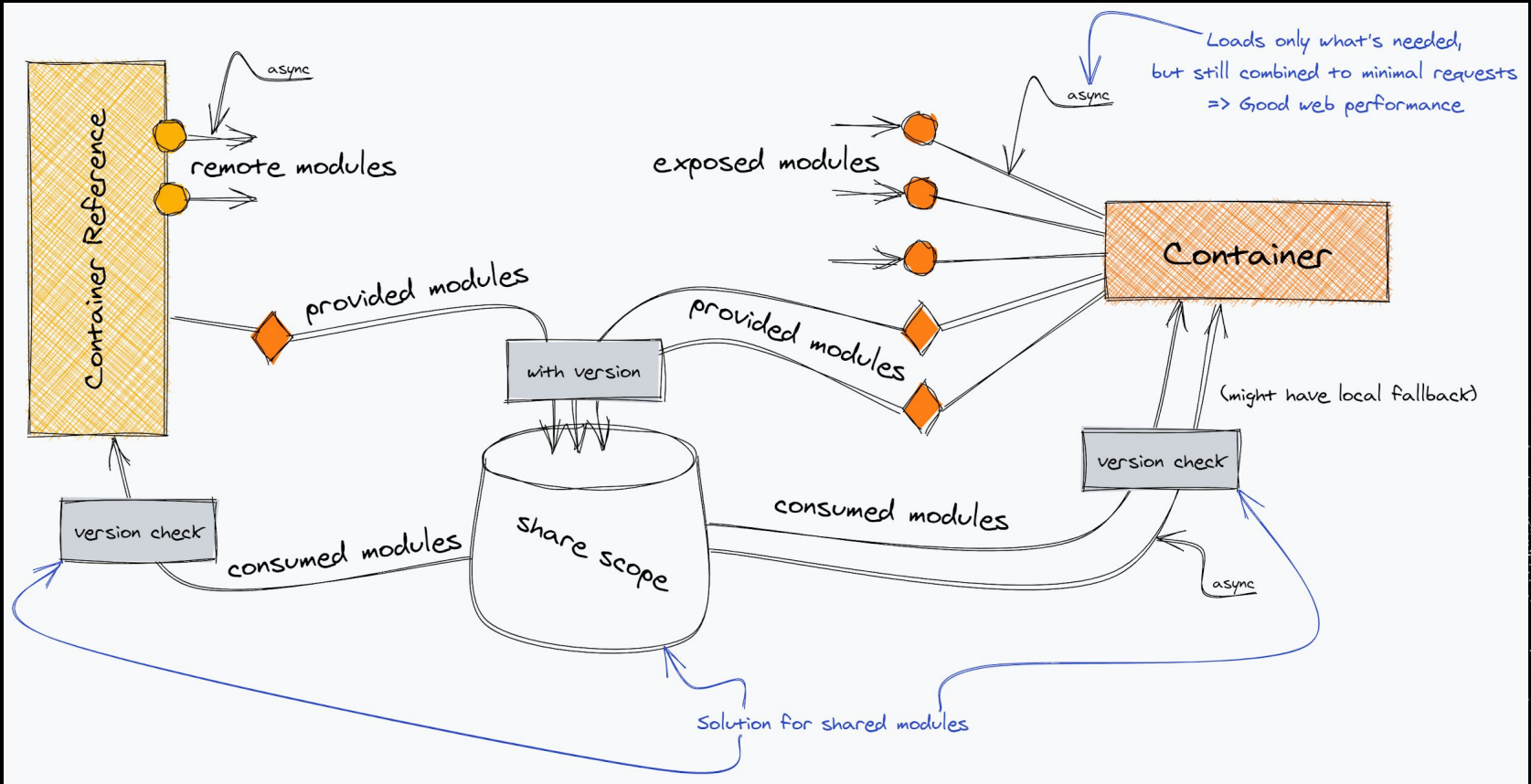
consume and consumed

4

Omnidirectional host

all three & negotiate dependencies

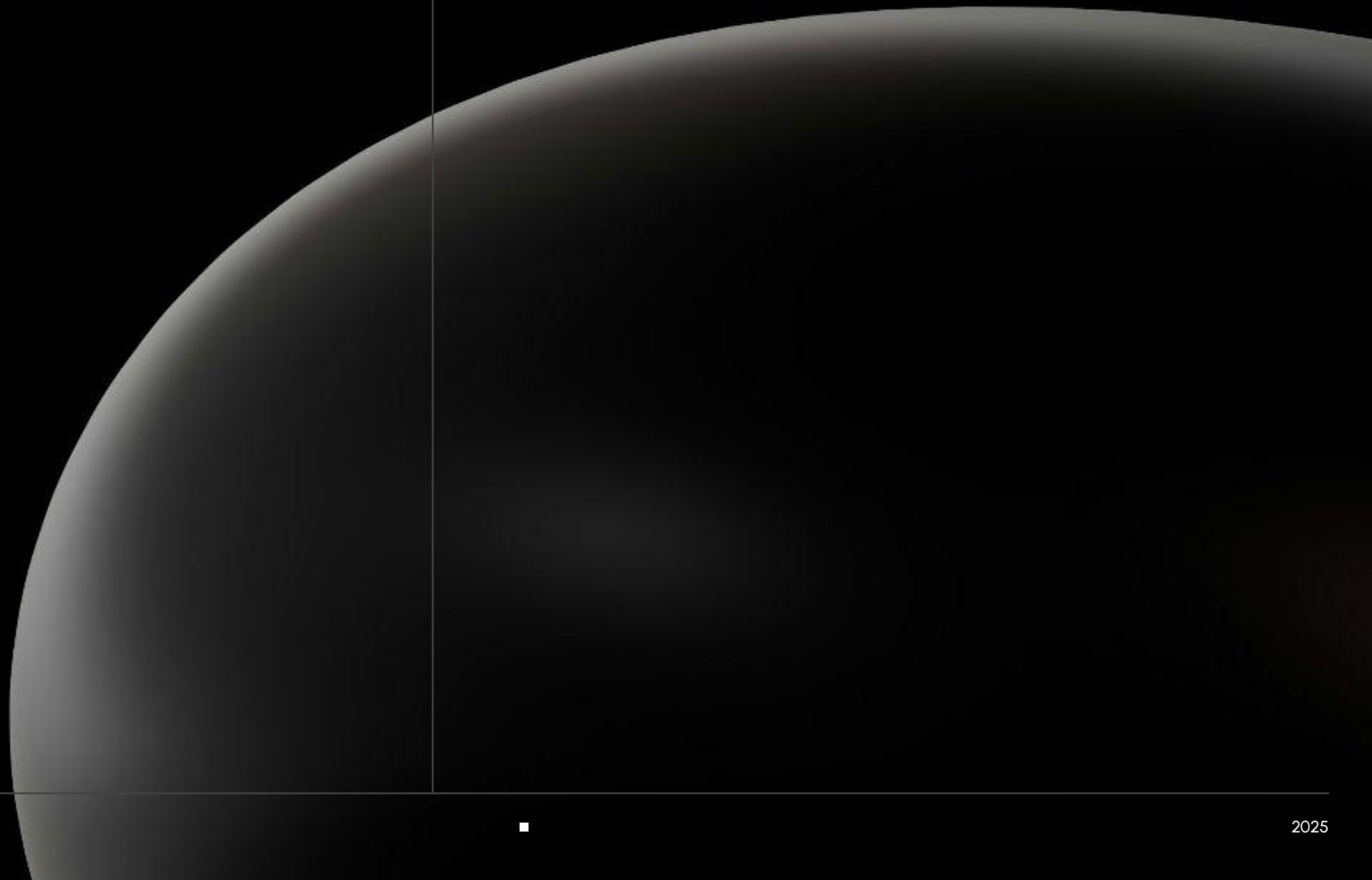
Overview


Loads only what's needed,
but still combined to minimal requests
=> Good web performance
async
Container Reference
remote modules
exposed modules
Container
provided modules
with version
provided modules
(might have local fallback)
version check
version check
consumed modules
share scope
consumed modules
async
Solution for shared modules

Why

- ➔ designed for large scale applications
- ➔ deploy multiple applications, or parts of an application
- ➔ multiple teams, autonomous workflows
- ➔ avoid learning curves
- ➔ efficient and productive to share feature code
- ➔ avoid multiple copies of same library
- ➔ share vendor code, stay flexible
- ➔ no UX drawbacks (page reloads)
- ➔ avoid complicated CI

Demo



Thank you