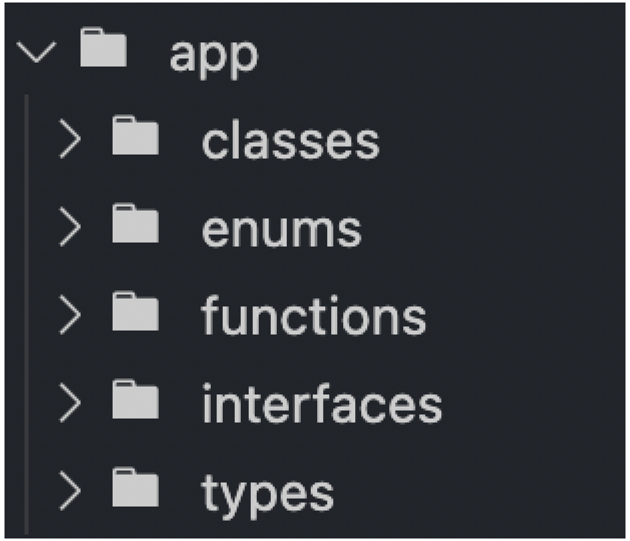
Scalable Frontend Architecture that meets Your Business

Thomas Gossmann - gos.si - @unistyler

Architecture

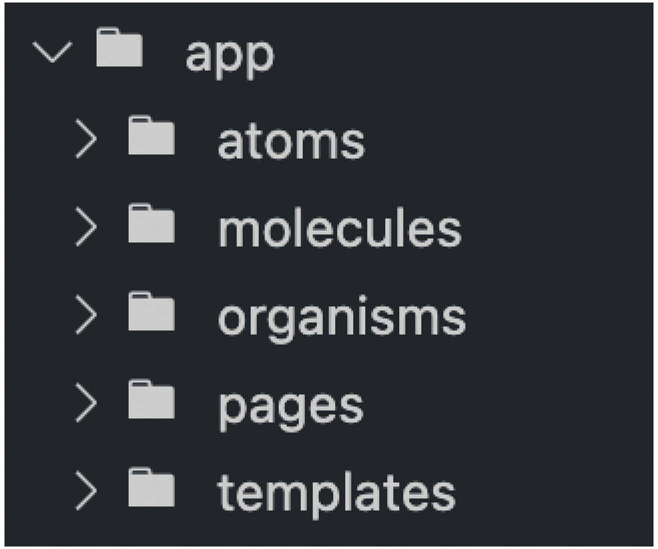
Architects: Draw the map and guide engineers to the treasure
Engineers: Read the map to reach the treasure

Quiz: What does this Product do? (1)



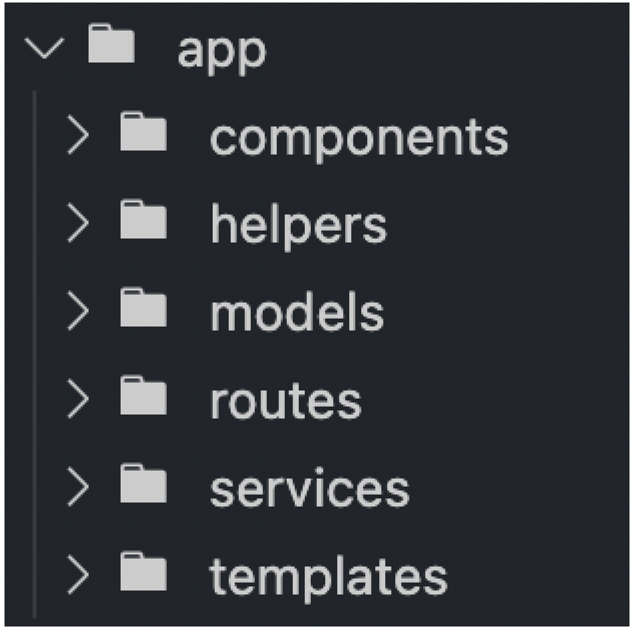
```
app
  classes
  enums
  functions
  interfaces
  types
```

Quiz: What does this Product do? (2)



```
app
  atoms
  molecules
  organisms
  pages
  templates
```

Quiz: What does this Product do? (3)



```
app
  components
  helpers
  models
  routes
  services
  templates
```

Default Directory Structure - Why ?

- Good onboarding to the framework
- Explains technical aspects of the framework

- Good for hobby and weekend projects
- Hardly scalable beyond that

PART 1

Meet Your Business

Tactical Design

Technical Objects

- Components
- Services
- Routes

↰ *not* aspects of your product

Domain Objects

- Contract
- Appointment
- Risk Audit
- Saloon
- Calendar

↰ they ***are*** aspects of your product

Why there is no Domain-Driven Development ?

It is hard to do. Some observed reasons:

1. Education: Data Structures, Algorithms, Design Patterns, Performance, …
    - Missing: Linguistic Course, Domain-Driven Design Pratices
2. We design development workflows for technical aspects
3. No visibility for the domain in our code
    - Lack of feedback from product people or designers
    - No reward to engineers for their contributing impact

Can we (Re)Design our Development Workflow with the Business in Mind?

1. Identify Technical Aspects that Encode Business Logic

Queries

```
function query(...args: unknown[]): NonNullable<unknown>;
```

- *Read*
- Questions: Ask facts about the system
- Abilities/Authorization/Guards/Conditions/Criteria: Control acces

Commands

```
function command(...args: unknown[]): void;
```

- *Write*
- Fire & Forget
- May/should cause side effects

Command-Query-Separation (CQS)

Functions to either be commands that perform an action or queries that respond data, but neither both!

Queries: Presentation Logic / Control Flow

↙ Two Times Business Logic. Two Times Anti-Patterns ↘

Helper

```
{{#if (feature-flag 'PROPLUS')}}
  Special Feature here
{{/if}}
```

- What's the name of the feature?

  hint: it is not "Pro Plus", that's only the feature flag currently used for its condition
- Not unit testable :(

Components

```
import Component from '@glimmmer/component';
import { service } from '@ember/service';
import type FeaturesService from 'whereever/features-infra-sits';

class Search extends Component {
  @service declare features: FeaturesService;

  get isProPlus() {
    return this.features.has('PROPLUS');
  }

  <template>
    {{#if this.isPropPlus}}
      Special Feature here
    {{/if}}
  </template>
}
```

Queries: Data Fetching

- Fetching data from your API
- Business logic part:
  - Endpoint
  - Parameters
  - Payload structure

Commands: Actions

Components

```
import Component from '@glimmer/component';
import { action } from '@ember/object';
import { AnotherComponent } from 'your-ui';

class Expose extends Component {
  @action
  onClick() {
    // whatever happens here
  }

  <template>
    <AnotherComponent @onClick={{this.onClick}}>
      Something sits here
    </AnotherComponent>
  </template>
}
```

Services

```
import Service from '@ember/service';

class UserService extends Service {
  createUser(data) {
    // ...
  }

  deleteUser(userId: number) {
    // ...
  }
}
```

Services

Services is an overloaded Term

Infrastructure Services

- API client
- Messaging / Message Broker

Application Services

- Session
- Features
- A/B Testing

Domain Services

- Domain Objects (CRUD)
- e.g. UsersService

We host Business logic in Components, Services, Routes, Controllers, Models merely to use Ember’s DI system.

We created a strong coupling of business logic to Ember’s DI system 🤔

What is the correct Statement?

(A)
Make a Framework a Dependency of your Business?

[✓ (B)
Your Business drives Implementation within a Framework?]

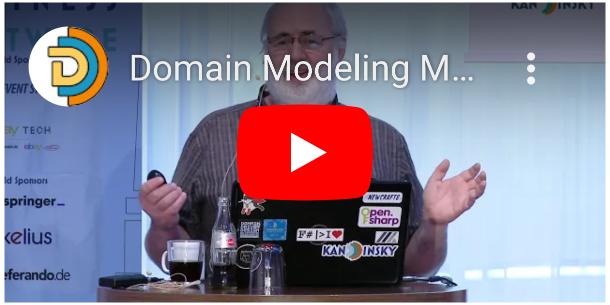
2. (Re)Design our Development Workflow

Rideshare Example

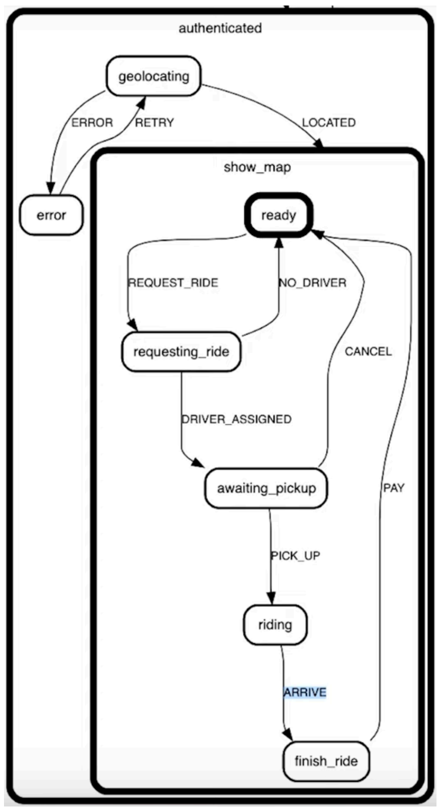
On the Development of Reactive Systems with Ember.js

by Clemens Müller and Michael Klein

Domain Modeling Made Functional



by Scott Wlaschin



```
interface User {
  id: string;
  name: string;
  type: 'rider' | 'driver';
}

type RideState =
  | 'requested'
  | 'declined'
  | 'awaiting_pickup'
  | 'driving'
  | 'arrived'
  | 'payed'
  | 'canceled';

interface Ride {
  id: string;
  from: string;
  to: string;
  riderId: string;
  driverId: string;
  state: RideState;
}
```

```typescript
interface User {
  id: string;
  name: string;
  type: 'rider' | 'driver';
}

type RideState =
  | 'requested'
  | 'declined'
  | 'awaiting_pickup'
  | 'driving'
  | 'arrived'
  | 'payed'
  | 'canceled';

interface Ride {
  id: string;
  from: string;
  to: string;
  riderId: string;
  driverId: string;
  state: RideState;
}
```

```typescript
// actions
function request(ride: Ride, rider: User): void;
function accept(ride: Ride, driver: User): void;
function drive(ride: Ride, driver: User): void;
function arrive(ride: Ride, driver: User): void;
function pay(ride: Ride, rider: User): void;
function cancel(ride: Ride, user?: User): void;

// guards rsp. abilities
function canRequest(ride: Ride, user: User): boolean;
function canAccept(ride: Ride, user: User): boolean;
function canDrive(ride: Ride, user: User): boolean;
function canDecline(ride: Ride, user: User): boolean;
function canArrive(ride: Ride, user: User): boolean;
function mustPay(ride: Ride, rider: User): boolean;

// questions
function isDriver(user: User): boolean;
function isRider(user: User): boolean;
function isDriverFor(ride: Ride, driver: User): boolean;
function calculateTravelDistance(ride: Ride): number;
```

Implementation

Goal

- Ride Details Page
- Task Based UI
- Domain Code in plain TS
- Thin layer in Ember for DI integration

Given

- `User` is given as part of `SessionService`
- `APIClient` is our `APIService`

```
import {
  canAccept, mustPay,
  accept, pay
} from 'your-domain';
import { Button } from '@hokulea/ember';

import type { TOC } from '@ember/component/template-only';
import type { Ride } from 'ember-domain';

interface RideActionsSignature {
  Args: {
    ride: Ride;
  }
}

const RideActions: TOC<RideActionsSignature> = <template>
  {{#if (canAccept @ride)}}
    <Button @push={{fn (accept) @ride}}>Accept</Button>
  {{/if}}

  {{#if (mustPay @ride)}}
    <Button @push={{fn (pay) @ride}}>Pay</Button>
  {{/if}}
</template>

export { RideActions };
```

2.1. Actions

1. Bi-Directional API, Statechart, Event-Driven Architecture, CQRS/ES
2. Uni-Directional API, Statechart, CRUD
3. Uni-Directional API, CRUD

Implementing Scenario 1

Fire & Forget

```
import type { APIClient } from 'infra';

async function accept(ride: Ride, driver: User, { apiClient }: { apiClient: APIClient }): void {
  await apiClient.post(`/ride/${ride.id}/accept`, {
    driverId: driver.id
  });
}
```

Implementation to focus on:

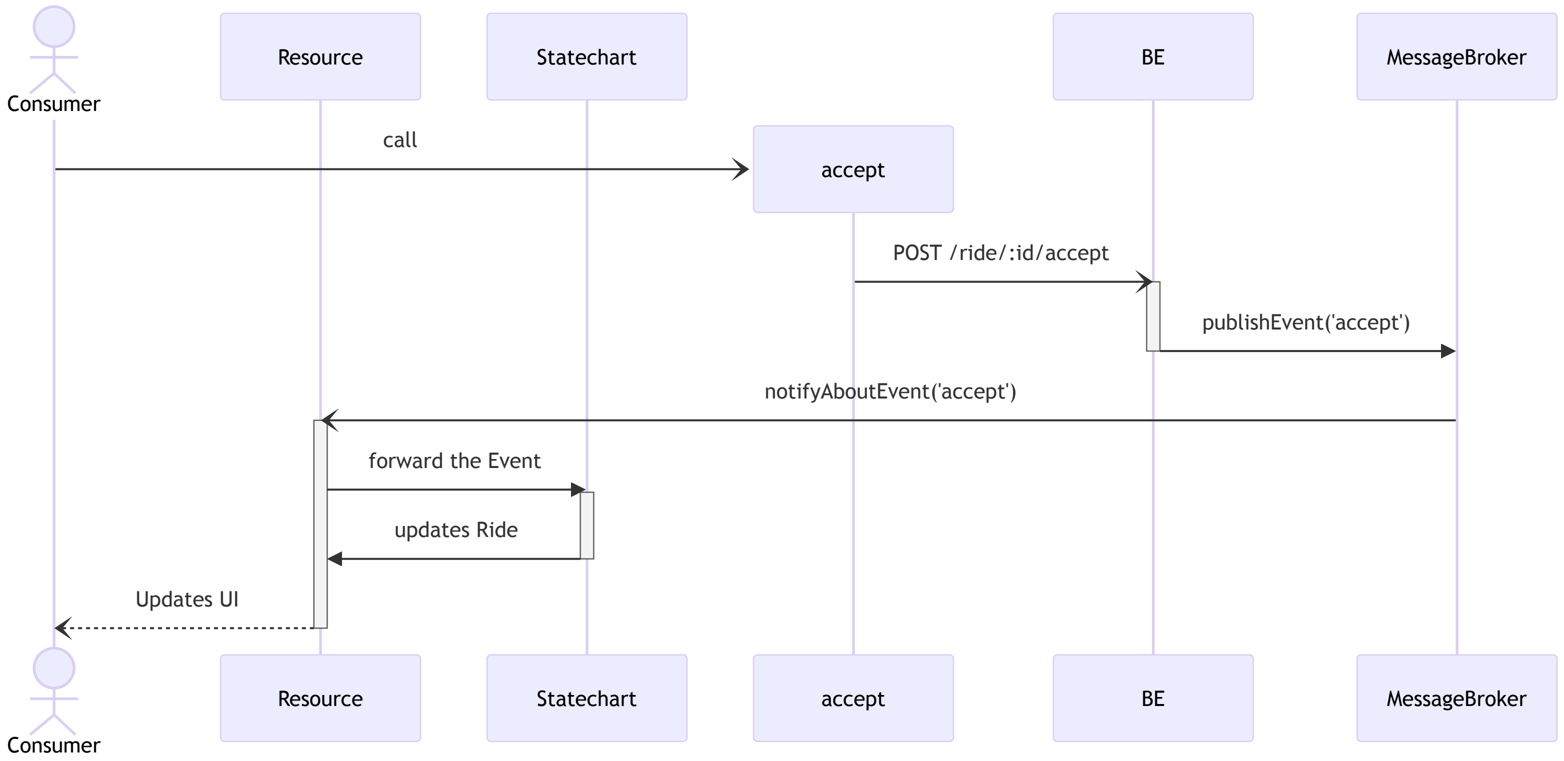
- Endpoint
- Parameters
- Payload Structure

Additionally to the Domain

- Infrastructure/technically relevant parameters
- Develop against interfaces
- Perfect to mock for testing

Scenario 1: Setup

Scenario 1: Action



Implementing Scenario 2

Fire & Play BE in FE

```
import type { APIClient } from 'infra';

async function accept(ride: Ride, driver: User, { apiClient }: { apiClient: APIClient }): void {
  await apiClient.post(`/ride/${ride.id}/accept`, {
    driverId: driver.id
  });
}
```

Scenario 2: Setup

Secnarion 2: Action

Implementing Scenario 3

Fire & Play BE in FE

```
import type { APIClient } from 'infra';

async function accept(ride: Ride, driver: User, { apiClient }: { apiClient: APIClient }): void {
  await apiClient.post(`/ride/${ride.id}/accept`, {
    driverId: driver.id
  });
}
```

2.2. Abilities

```
function canAccept(ride: Ride, user: User) {
  // when...
  return (
    // ride is in state requested...
    ride.state === RideState.Requested &&
    // AND user is a driver
    isDriver(user)
  );
}
```

- use single exit functions

  no guards with early exits, we are only interested when something can be done, not when it can't be done
- readability: use positive statements (non negated statements)
- annotate with comments to explain tricky non-readable code for non-tech people (when necessary)

2.3. Integration with Ember

Abilities

```
function canAccept(ride: Ride, user: User) {
  return ride.state === 'requested' && isDriver(user);
}
```

↓

?

↓

```
{{#if (canAccept @ride)}}
  ...
{{/if}}
```

Actions

```
import type { APIClient } from 'infra';

async function accept(ride: Ride, driver: User, { apiClient
  await apiClient.post(`/ride/${ride.id}/accept`, {
    driverId: driver.id
  });
}
```

↓

?

↓

```
<Button @push={{fn (accept) @ride}}>Accept</Button>
```

Abilities: `ability()` from `ember-ability`

```
import { canAccept as upstreamCanAccept } from 'your-plain-ts-domain';
import { ability } from 'ember-ability';

const canAccept = ability((owner) => (ride: Ride) => {
  const session = owner.lookup('service:session');
  const { user } = session;

  return upstreamCanAccept(ride, user);
});

export { canAccept };
```

ember-sweet-owner

```
import { sweetenOwner } from 'ember-sweet-owner';

const { services } = sweetenOwner(owner);
const { session } = services;
```

↓

```
{{#if (canAccept @ride)}}
  ...
{{/if}}
```

Actions: `action()` from `ember-command`

```
import { accept as upstreamAccept } from 'your-plain-ts-domain';
import { action } from 'ember-command';

const accept = action(({ services }) => (ride: Ride) => {
  const { session, api } = services;
  const { user } = session;

  upstreamAccept(ride, user, { apiClient: api });
});

export { canAccept };
```

↓

```
<Button @push={{fn (accept) @ride}}>Accept</Button>
```

Domain Code

- is actually tiny
- many tiny functions
- easy unit testing
- Plain TS can be integrated into multiple systems:
  - thin integration layer into frameworks
  - statecharts

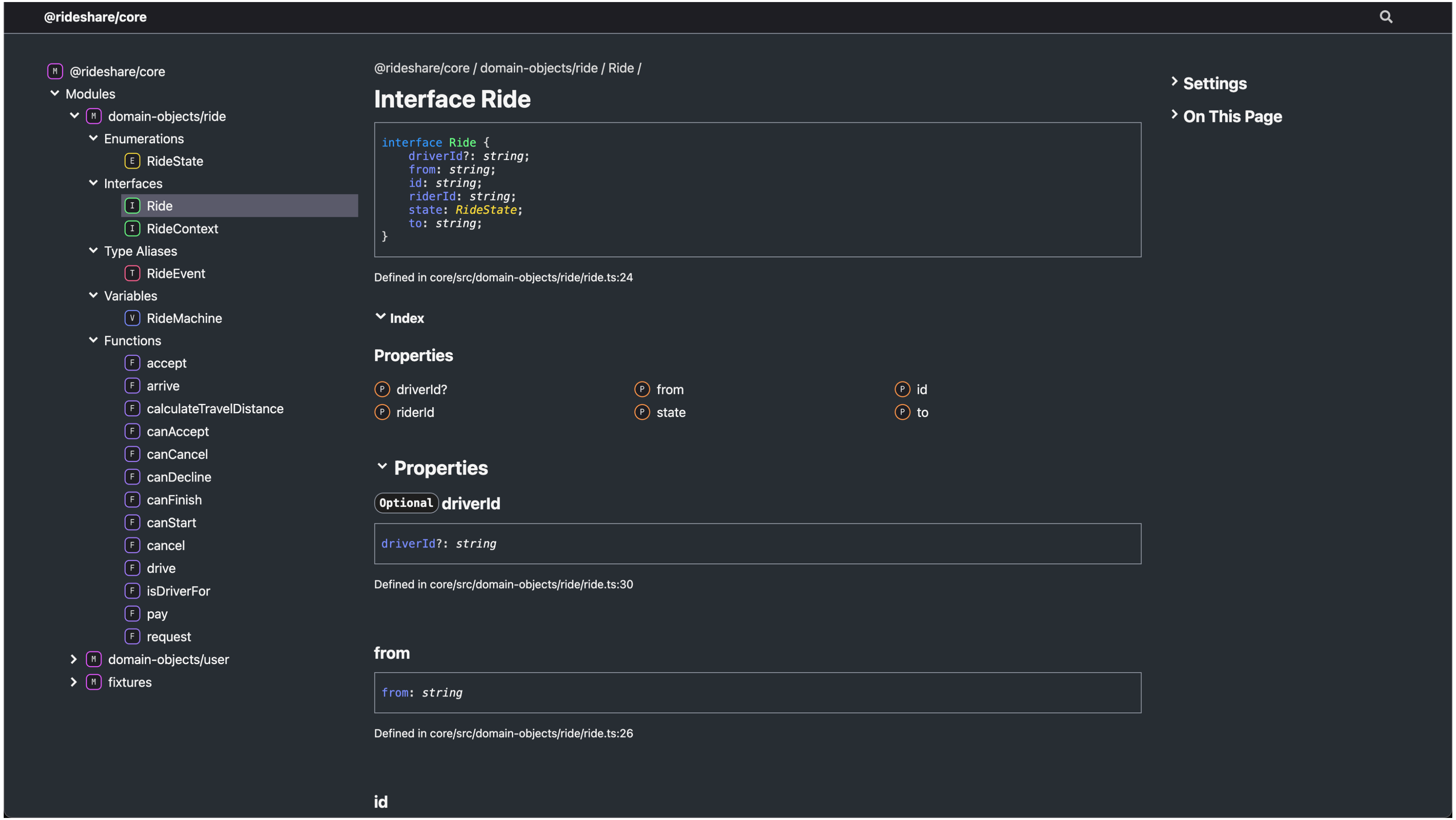
but:

- is still hard to write code like that
- that’s a naive design
- needs visibility
- a way to reward engineers

Finish the Development Workflow Design

- can we have a "magic number" (similar to code-coverage), that signals:

  "good code quality that follows our architecture design"
- I haven't found one… (yet?)
- Follow nature: Indicator Species
- Bridge between engineers and non-tech-people
- Use: `typedoc`


@rideshare/core

@rideshare/core / domain-objects/ride / Ride /

Interface Ride

```
interface Ride {
    driverId?: string;
    from: string;
    id: string;
    riderId: string;
    state: RideState;
    to: string;
}
```

Defined in core/src/domain-objects/ride/ride.ts:24

Index

Properties

driverId?	from	id
riderId	state	to

Properties

Optional driverId

```
driverId?: string
```

Defined in core/src/domain-objects/ride/ride.ts:30

from

```
from: string
```

Defined in core/src/domain-objects/ride/ride.ts:26

id

Settings

On This Page

Configure `typedoc`

- Organize our domain aspects:

```
/**
 * @group Domain Objects
 * @module Ride
 */
```

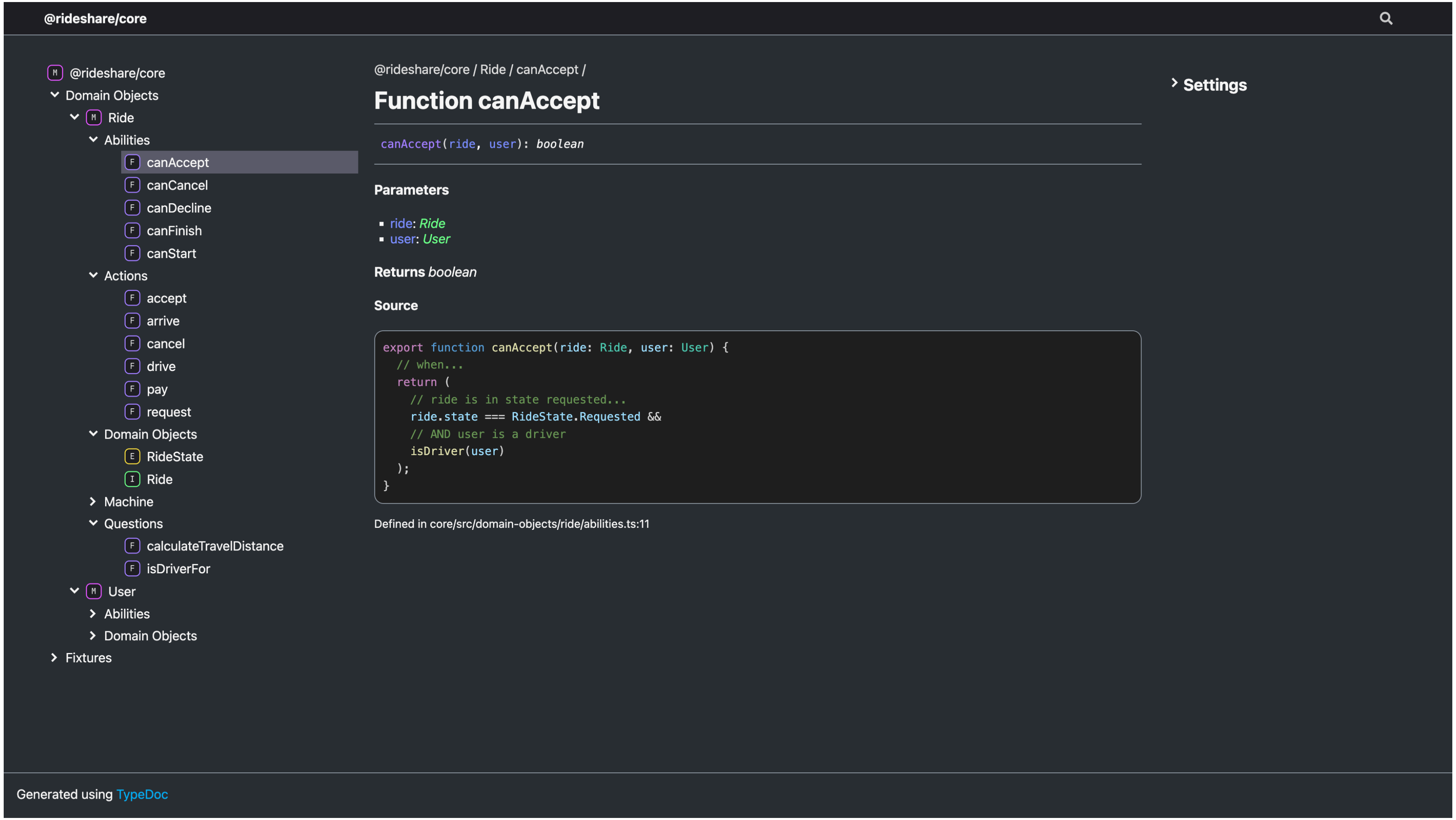
- Give meaning to our code:

```
/**
 * @category Abilities
 * @source
 */
```

Plugin: `typedoc-plugin-inline-sources`

- Configure typedoc:

```
"navigation": {
  "includeCategories": true,
  "includeGroups": true,
  "includeFolders": false
},
"categorizeByGroup": false
```



@rideshare/core

@rideshare/core / Ride / canAccept /

Function canAccept

```
canAccept(ride, user): boolean
```

Parameters

- ride: *Ride*
- user: *User*

Returns *boolean*

Source

```
export function canAccept(ride: Ride, user: User) {
  // when...
  return (
    // ride is in state requested...
    ride.state === RideState.Requested &&
    // AND user is a driver
    isDriver(user)
  );
}
```

Defined in core/src/domain-objects/ride/abilities.ts:11

Settings

Generated using TypeDoc

Benefits

- Make complexity visible
- Significant reduction in bugs
- Feature devlivery improved by factor 2-3x
- Increased developer velocity
- Business logic Lego

PART 2

Organizing Code and Scale it Up

Strategic Design

Naive Approach

- Use Ember Addons
- Use Ember Engines
- Move things from app into addons/engines

⇨ "False" Scalability

Example: A Zoo

The technical goal is to keep animals and visitors separated

Technical

💬 Let's make a compound for animals and a compound for visitors

⇨ Missing accomplished

Domain

💬 Who put herbivores and carnivores in the same compound ?

⇨ Short term attraction

✗ No long term, sustainable solution

Frontend Architecture: How to Bu...
webinale
SESSION
Frontend Architecture: How to Build a Zoo?
Thomas Gossmann

Domain

Understanding subdomains

① Core Subdomain

Unique/Core part of your product.

② Supporting Subdomain

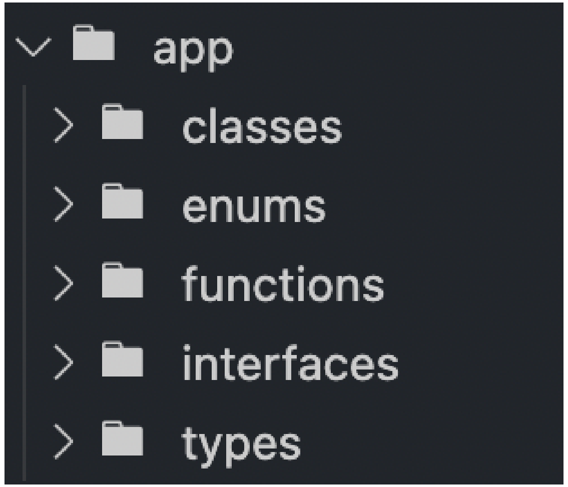
Ancillary parts that support your core.

③ Generic Subdomain

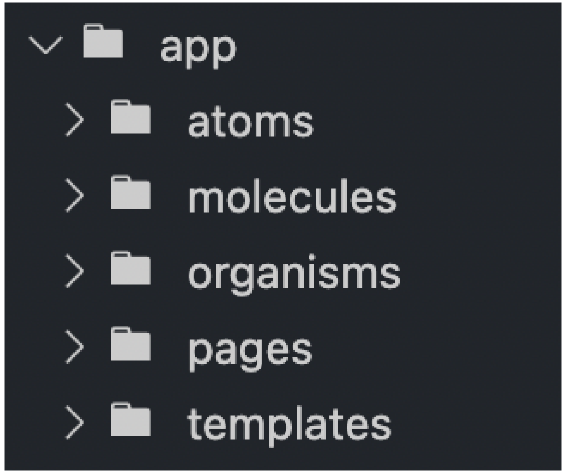
We’ll find these parts in many applications (e.g. user management).

Subdomains help you distill your product into manageable pieces.

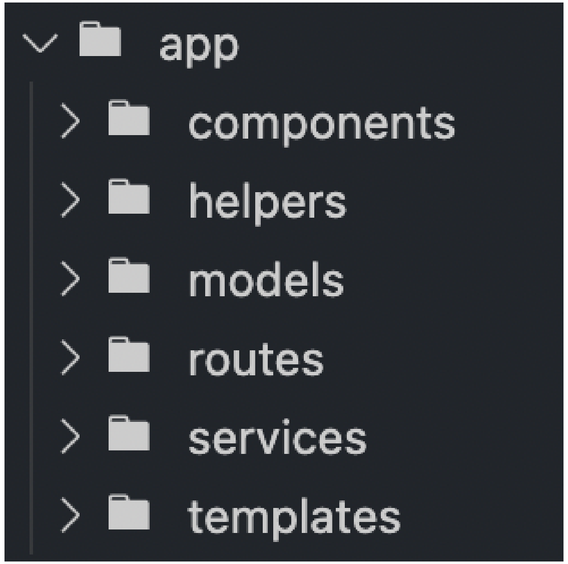
Time to Solve that Puzzle



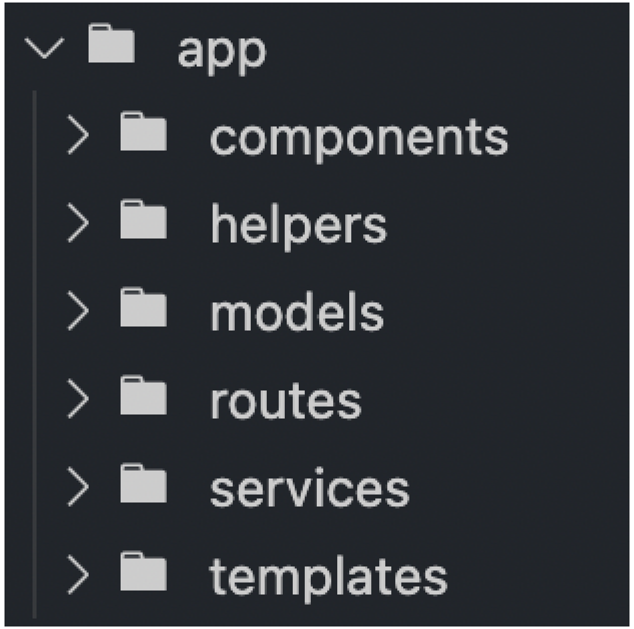
```
app
  classes
  enums
  functions
  interfaces
  types
```



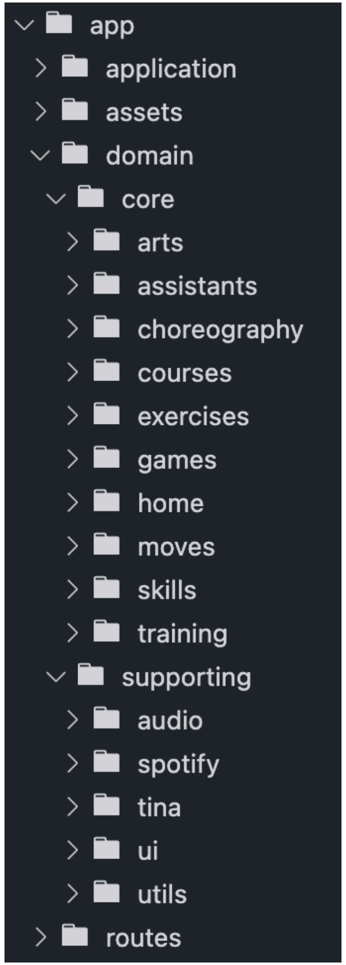
```
app
  atoms
  molecules
  organisms
  pages
  templates
```



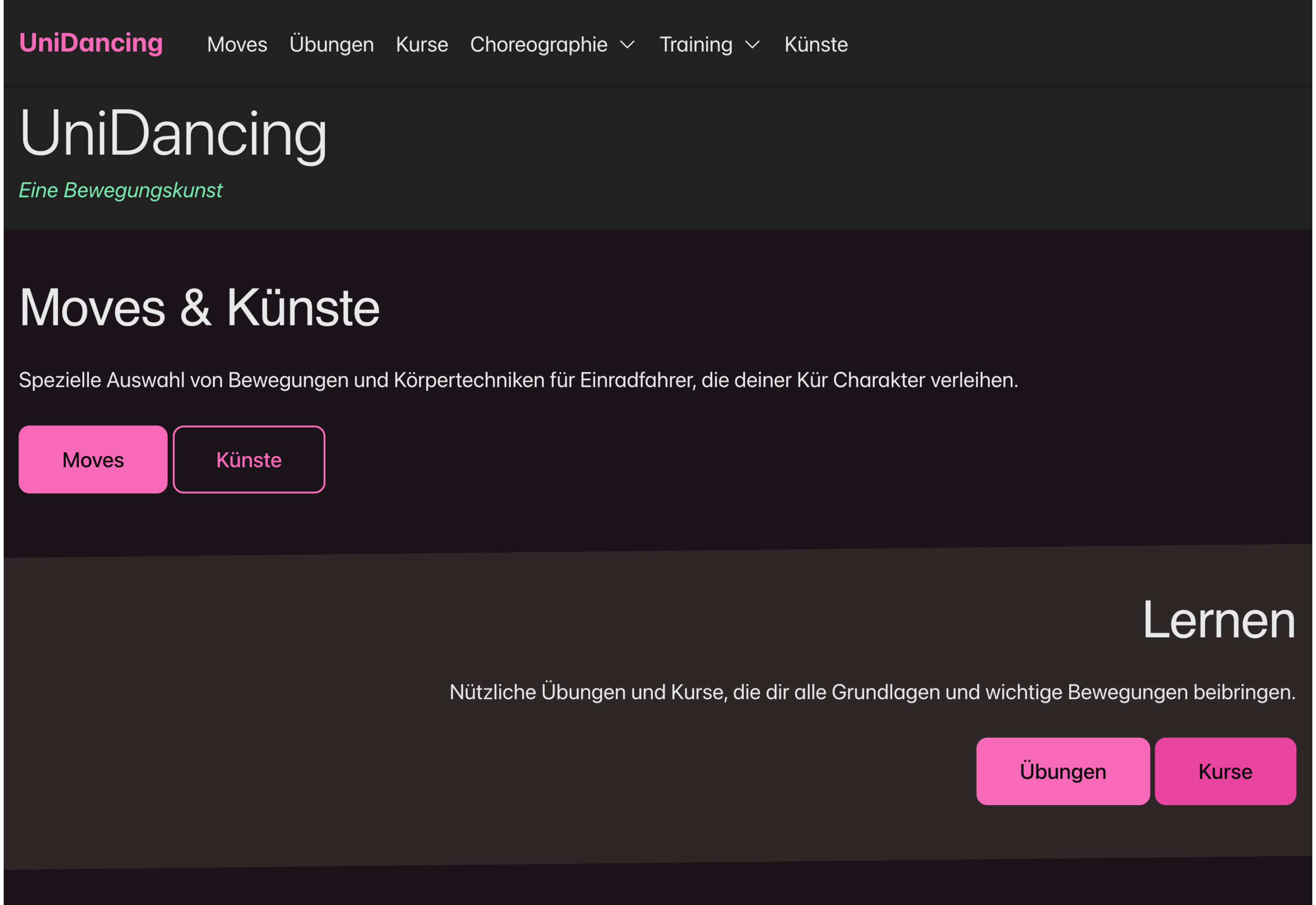
```
app
  components
  helpers
  models
  routes
  services
  templates
```



```
app
  components
  helpers
  models
  routes
  services
  templates
```



```
app
  application
  assets
  domain
    core
      arts
      assistants
      choreography
      courses
      exercises
      games
      home
      moves
      skills
      training
    supporting
      audio
      spotify
      tina
      ui
      utils
  routes
```



UniDancing

Eine Bewegungskunst

Moves & Künste

Spezielle Auswahl von Bewegungen und Körpertechniken für Einradfahrer, die deiner Kür Charakter verleihen.

Lernen

Nützliche Übungen und Kurse, die dir alle Grundlagen und wichtige Bewegungen beibringen.

github.com/gossi/unidancing

Colophon UniDancing.art

- **Each domain directory has an `index.gts` which contains the public API**

- Routes are exported as part of each domains public API

  ```
  // routes/exercises/index.gts
  export { IndexRoute as default } from '../../domain/core/exercises';
  ```

- `ember-polaris-routing`: for defining routes (there is also `ember-route-template`)

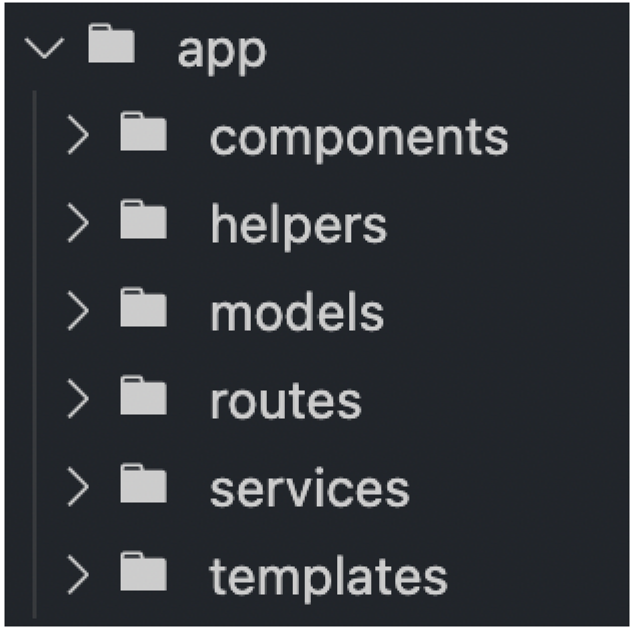
- `ember-polaris-service`: Infrastructure located in their respective domain (no root level `services/` directory)

What’s Inside a Subdomain?

- Domain Objects
- Actions
- Abilities
- Questions
- Components
- Routes
- Services / Resources

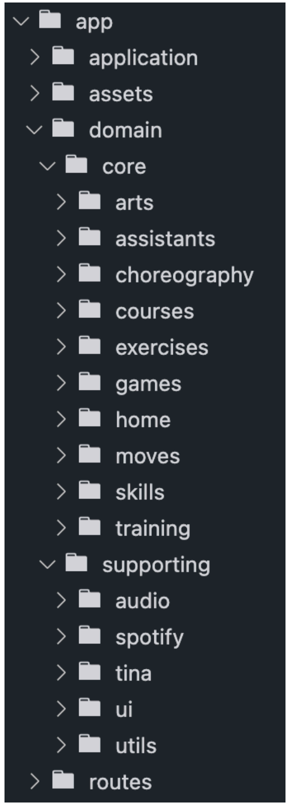
Public API as gateway to export what is accessible from the outside

Monolith



```
app
  components
  helpers
  models
  routes
  services
  templates
```

Modular Monolith

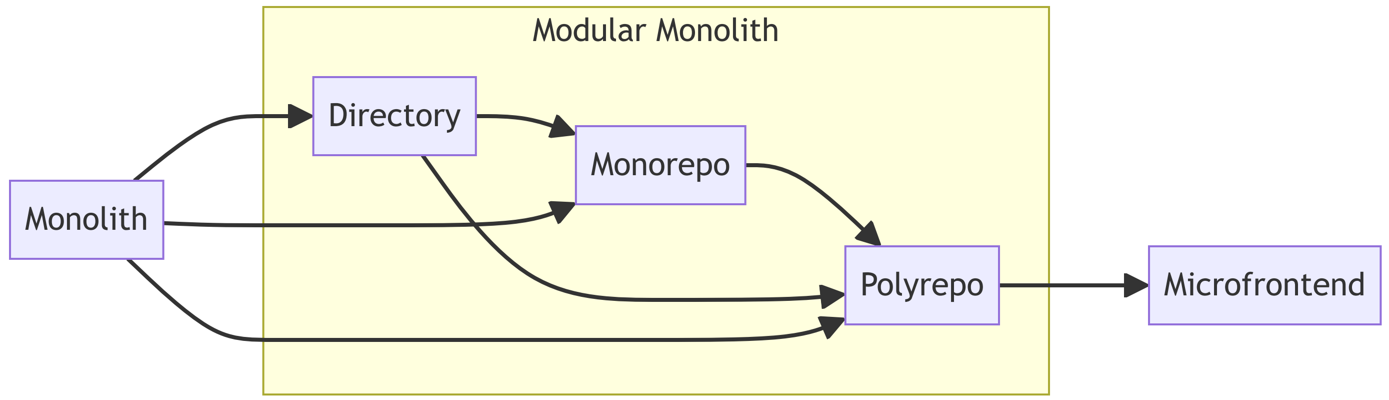


```
app
  application
  assets
  domain
    core
      arts
      assistants
      choreography
      courses
      exercises
      games
      home
      moves
      skills
      training
    supporting
      audio
      spotify
      tina
      ui
      utils
  routes
```

Modular Monolith

1. Directory: `domain/`
2. Monorepo: Private and public packages packages per subdomain
3. Polyrepo: One repository per subdomain with private and public packages

Scaling Up



Modular Monolith: Polyrepo

One repo per subdomain

Pro

- Use the physical boundaries of a repo for internal/public API
- Everything public API is published to your registry

Contra

- You need the publish/update dance
- Use `release-plan`
- Use `renovate` / `dependabot` to automate updates

Tip

@unidancing/training

- core
- public-api
- ember-core (addon)
- ember (addon)
- main (engine)

Legend: Internal Public

Modular Monolith: Monorepo

One repo for all subdomains

Pro

- No need to for publishing/updating
- Faster development time

Contra

- Needs to mimic the boundaries of a polyrepo
- ⚠️ Linting is required!
- ⚠️ Extra tooling for linting against internal/public APIs

Tip

- `domain/core/`
  - `choreography/`
  - `training/`
    - 📦 TS core
    - 📦 e ember (addon)
  - `exercises/`

Legend: 📦 Internal 📦 Public

Microfrontend

- Subdomain independently deployable
- Ember engines would be the technological choice
- Currently not possible

`ember-engines`

- Use them for isolated context
- Do NOT use them for route/chunk splitting (use embroider for that)
- Similar to "composable components", Ember will have "composable apps" - and I think that is beatiful
- The technical solution for this is unclear as of now (apps and engines might merge)

Takeaways

- Focus on the domain
- Make your domain/complexity visible
- Reward your engineers for their contribution impact
- Your domain tells you how to scale up

Thank You

:)

Thomas Gossmann - gos.si - @unistyler

CLARK