Hexagonal Architecture

QWAN - Quality Without a Name

ZiuZ Visual Intelligence, 23 maart

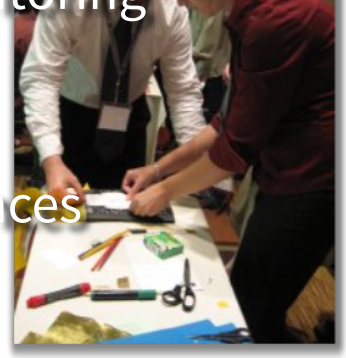
Quality Without A Name

marc@qwan.eu | willem@qwan.eu | rob@qwan.eu | christina@qwan.eu

What we do

increase business value from software development

- Consultancy en Mentoring
- Training
- Development
- Organizing conferences



Our learning vision

Learning by doing

Goals

to understand how application architecture can be malleable through hexagonal architecture

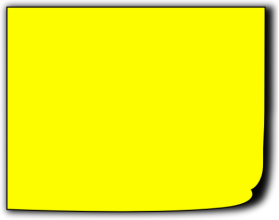
Deliberate Practice

We are here to learn, reflect, try and try over again.

Expectations

- what would you like to learn?
- what questions do you have?

1 item / keyword per post-it



Agenda

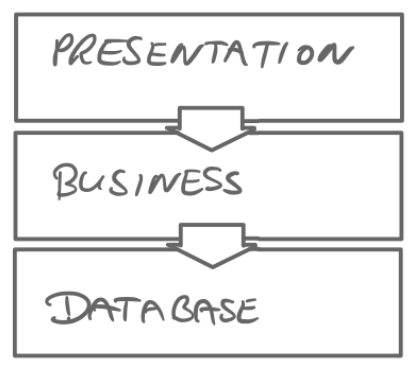
- Hexagonal architecture - introduction
- Design walkthrough: embedded vending machine
- A hexagonal perspective on automated tests
- Hexagonal architecture in code, hands on exercise

Working agreements & practical stuff

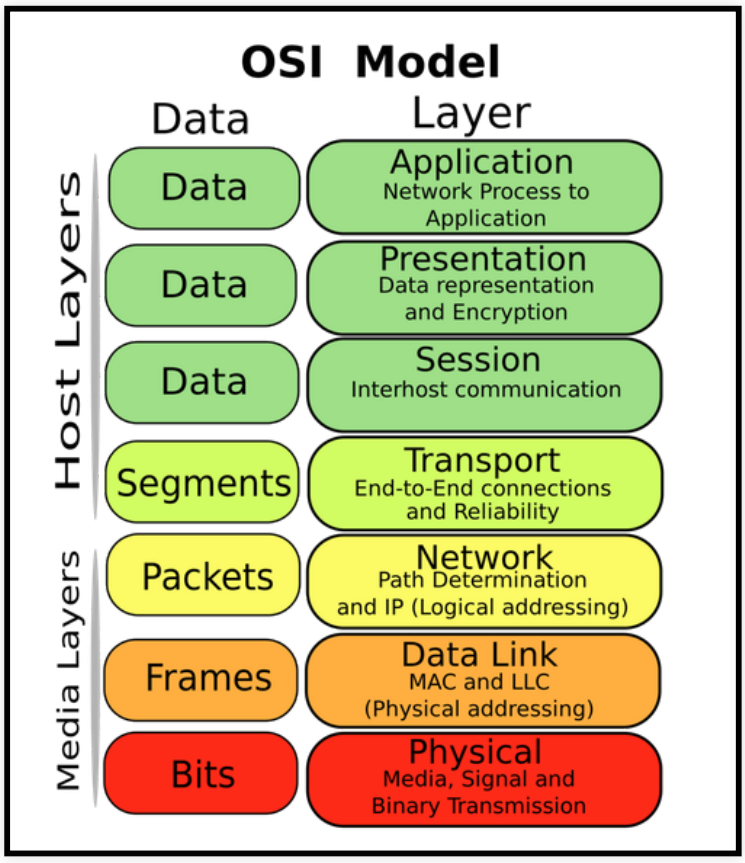
- Questions & remarks - ✋
- Breaks - ☕
- **This is *your* party!**
  Please keep your work outside

Where do we come from?

Layered architecture



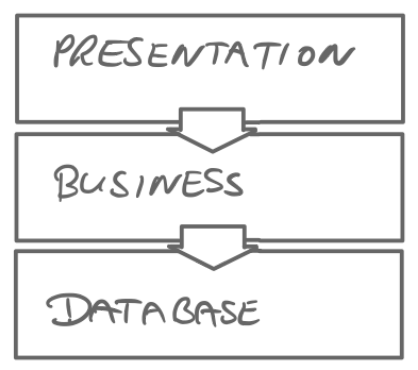
But not everything is...



And we all know...

gravity

And we all know...



gravity

Tight coupling!

everything depends on the whole model

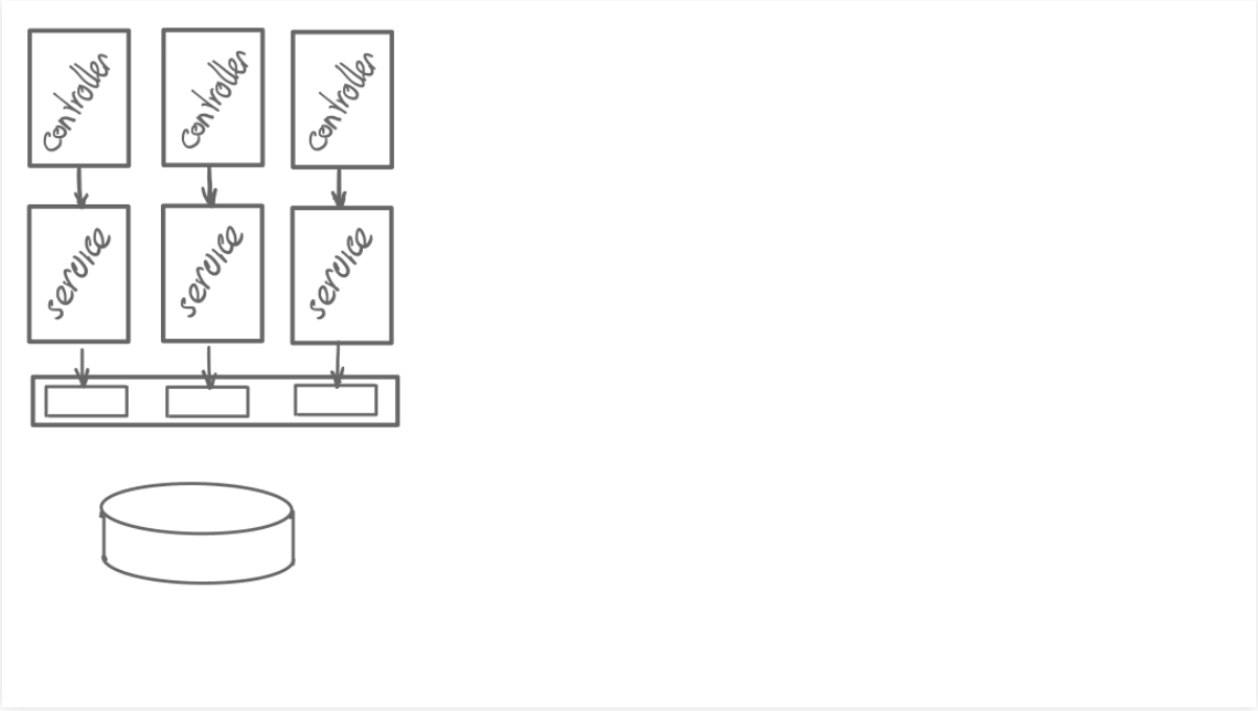
or

one object to rule them all

everything depends on the whole model

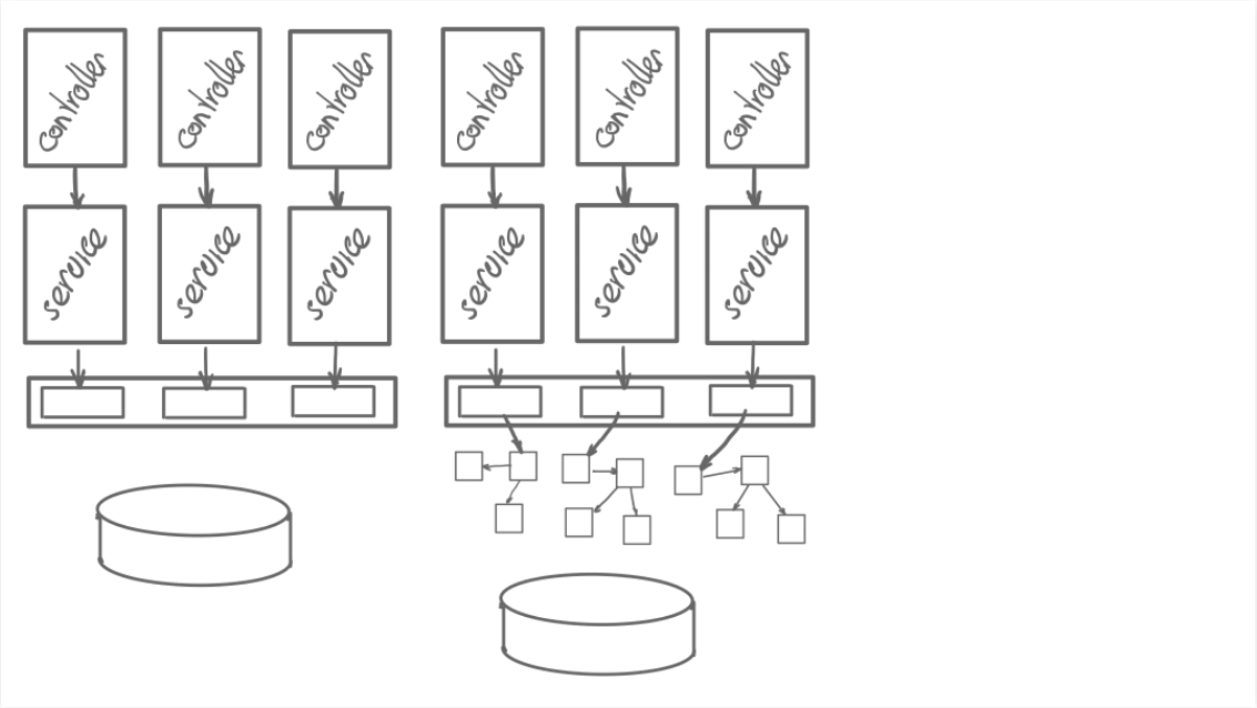
but how ?

A typical application architecture



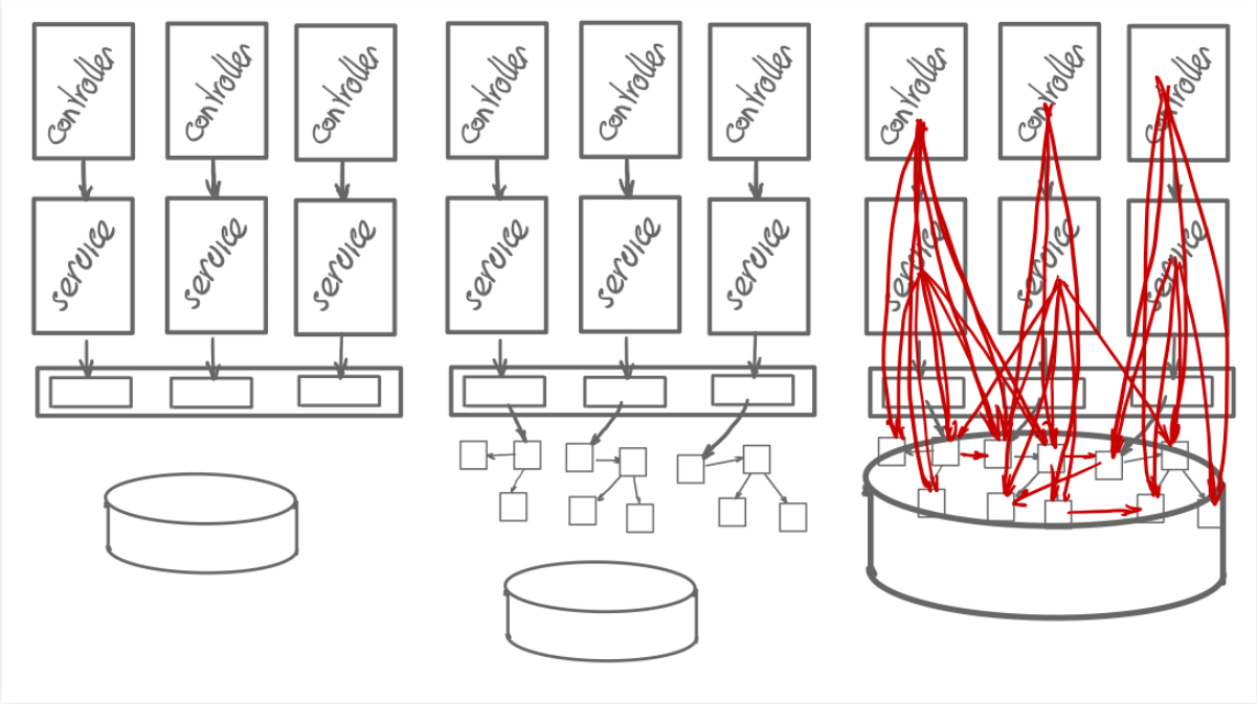
the theory

A typical application architecture

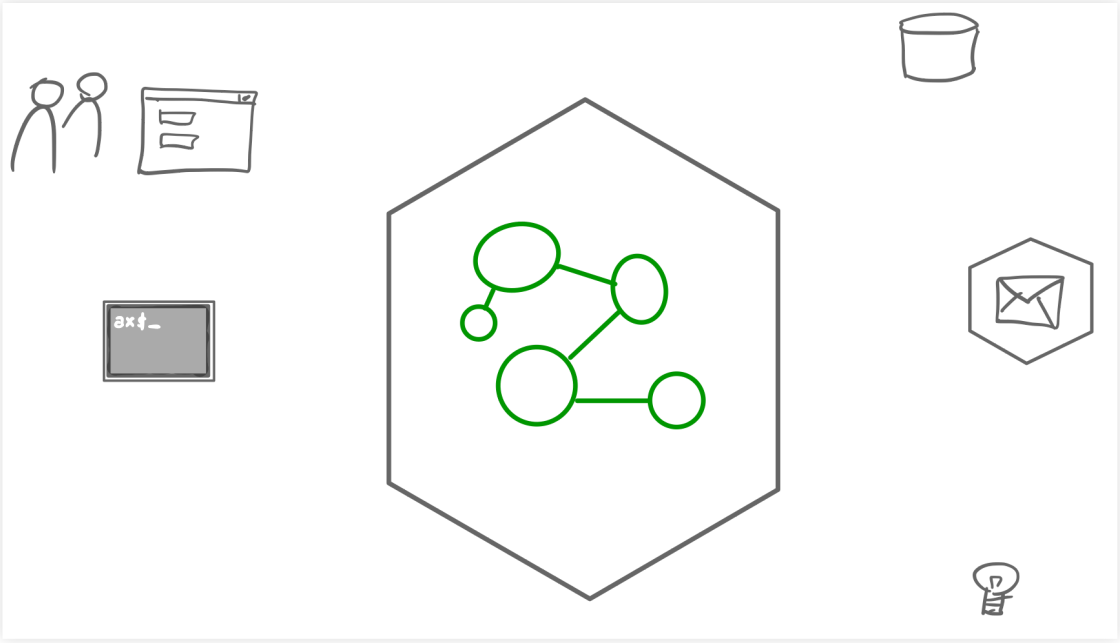


a bit more realistic

But what really happens

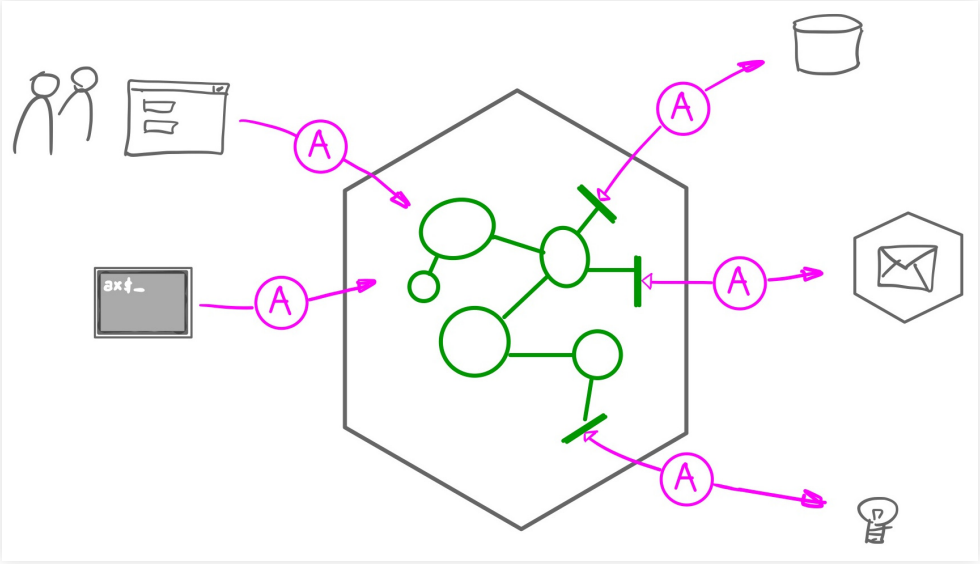


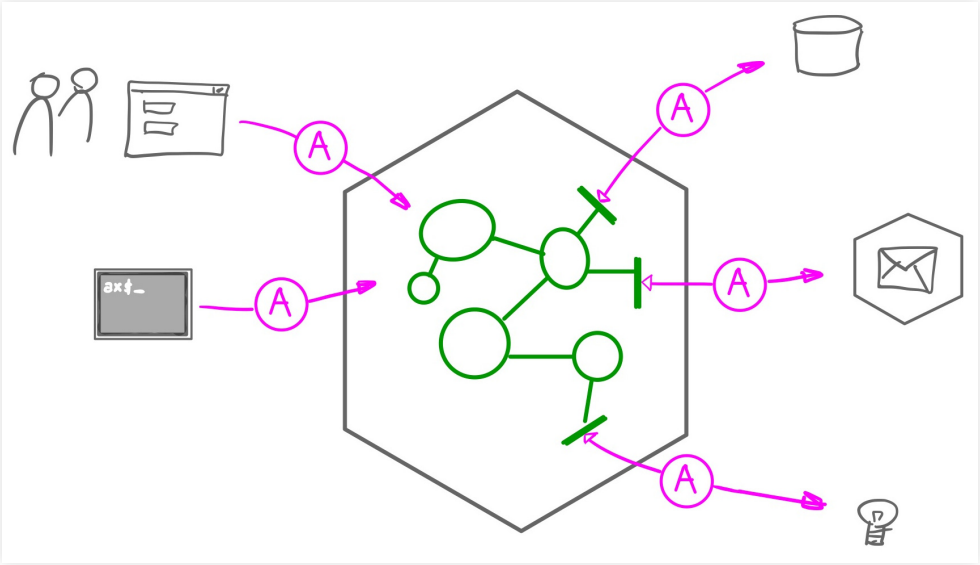
The Hexagon



Hexagonal architecture by Alistair Cockburn, aka Ports & Adapters / Clean Architecture

Domain in the center

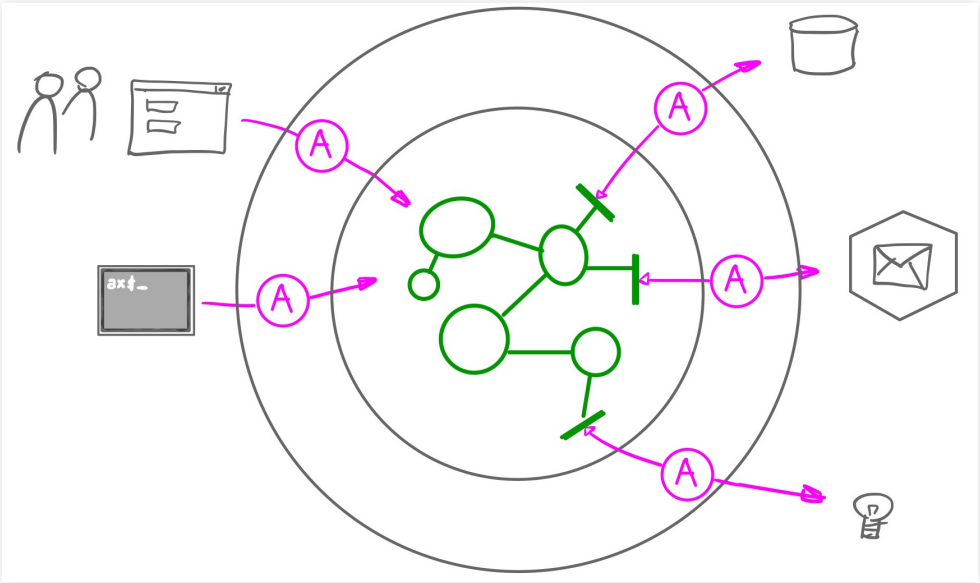




Port: set of interactions with outside world with same intent

Adapter: map domain ⇔ outside world

Primary vs secondary ports

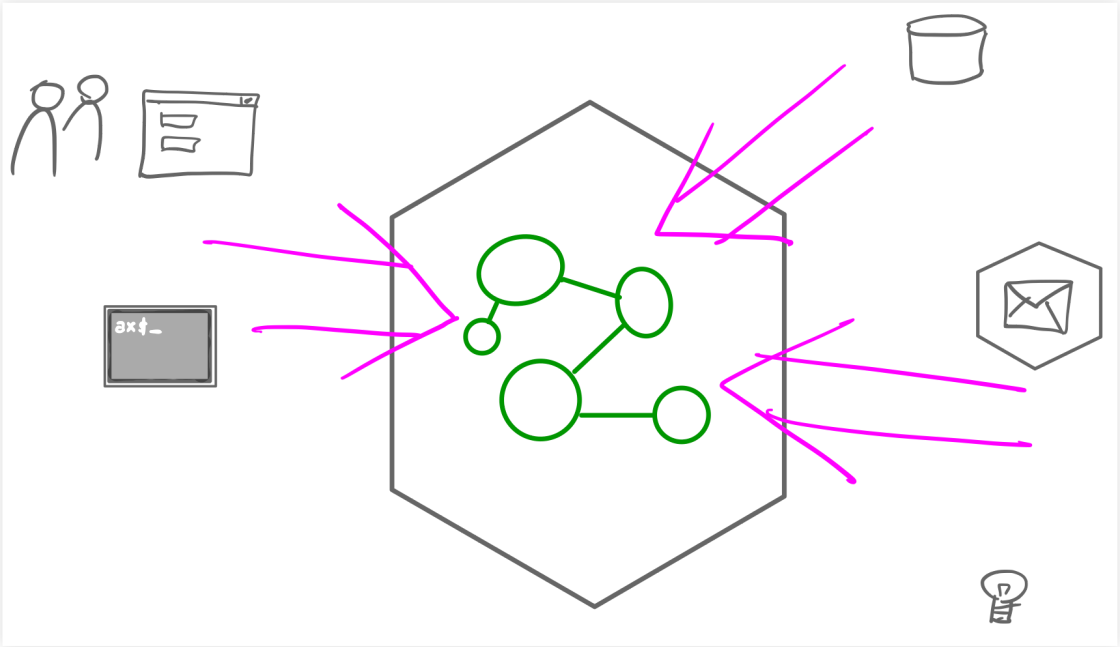


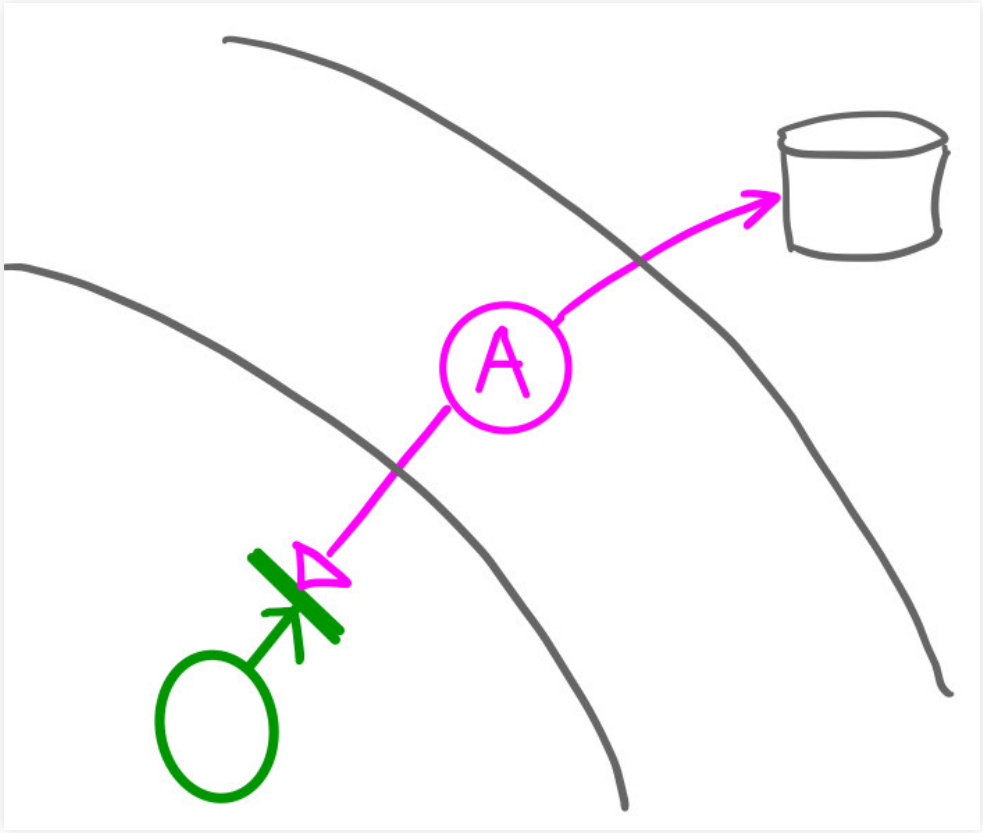
Port: set of interactions with outside world with same intent

Adapter: map domain ⇔ outside world

Primary vs secondary ports

Dependencies go outside-in





Principles

- Dependency inversion
- Framework independence
- Interface segregation

Code example

Finishing an order

updates existing order and sends out email

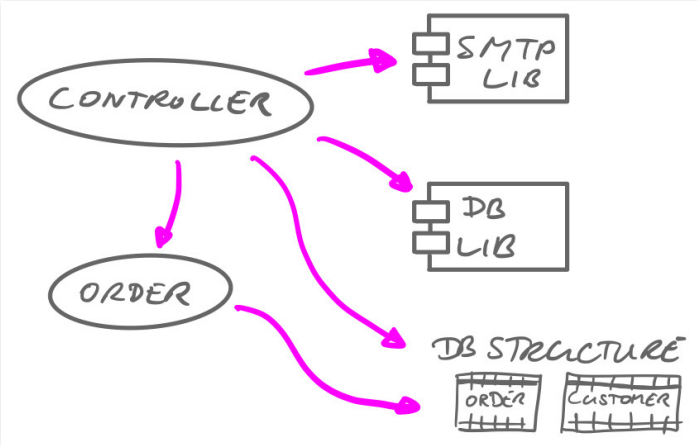
The ugly

```
module Ugly
  class OrderController
    attr_reader :db, :mailer

    def confirm(order_id)
      order = db.execute("select o.id order_id, o.description order_
          o.status order_status, c.id customer_id, c.name customer_n
          c.address customer_address, c.email customer_email
        from orders o join customers c on o.customer_id = c.id")
          .map { |e| Order.from_query_result(e) }.first
      db.execute("update orders set status='order' where id='#{order
      mailer.smtp(order.customer.email, "Your order with id: #{order
    end
  end
```

The ugly

```
module Ugly
  class Order < Struct.new(:id, :description, :status, :customer)
    def self.from_query_result(rs)
      Order.new(rs['order_id'], rs['order_description'], rs['order_s
    end
  end

  class Customer < Struct.new(:id, :name, :address, :email)
    def self.from_query_result(rs)
      Customer.new(rs['customer_id'], rs['customer_name'], rs['custo
    end
  end
end
```

The ugly



All concerns are everywhere

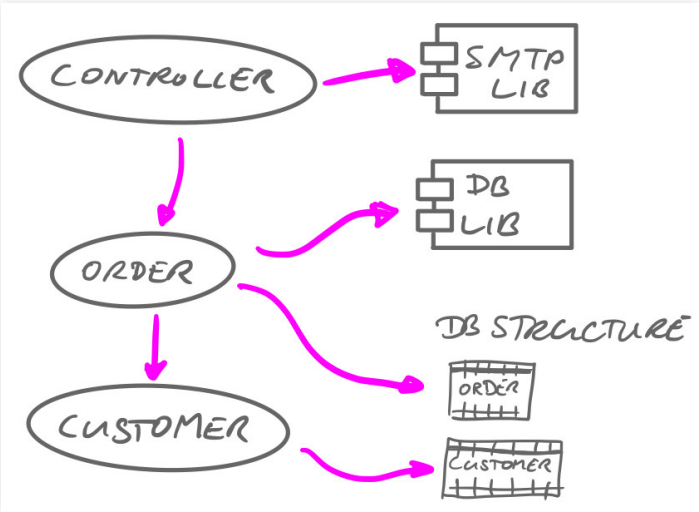
The slightly better but...

```
module SlightlyBetter
  class OrderController
    attr_reader :db
    attr_reader :mailer
    def confirm(order_id)
      order = Order.find_by_id(order_id)
      order.status = 'order'
      mailer.smtp(order.customer.email, "Your order with id: #{order
      order.save
    end
  end
```

The slightly better but...

```
class Order < Struct.new(:id, :description, :status, :customer)
  def self.from_query_result(rs)
    Order.new(rs['order_id'], rs['order_description'], rs['order_s
  end
  def self.find_by_id(id)
    order = DbConfig::DB.execute("select ..... ").map { |e| Order.
  end
  def save
    DbConfig::DB.execute("update orders set description='#{descrip
  end
end
class Customer < Struct.new(:id, :name, :address, :email)
  def self.from_query_result(rs)
    Customer.new(rs['customer_id'], rs['customer_name'], rs['custo
  end
end
```

The slightly better but...



Still quite some harsh db and mailer dependencies

The hexagonal

```
module Domain
  class ConfirmCommand < Commands::Command
    attr_reader :order_repository, :messaging
    def execute(order_id)
      order = order_repository.find_by_id(order_id)
      order.confirm(messaging)
      order_repository.save(order)
    end
  end
  class Order < Struct.new(:id, :description, :status, :customer)
    def confirm(messaging)
      status = 'order'
      messaging.send_order_confirmation(self, customer)
    end
  end
end
```

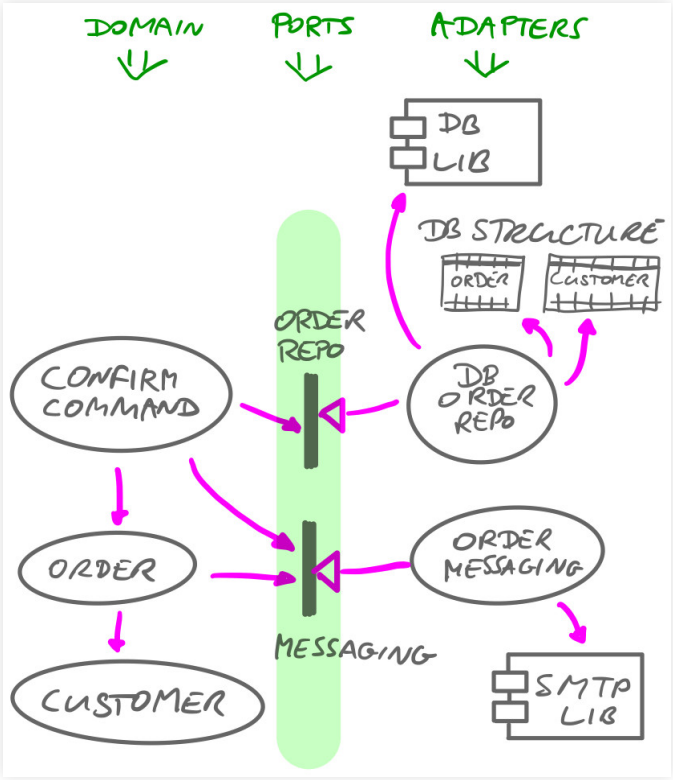
The hexagonal

```
module Adapters
  class OrderMessaging
    attr_reader :mailer
    def send_order_confirmation(order, customer)
      mailer.smtp(order.customer.email, render_message(order, custom
    end
    def render_message(order, customer)
      "Your order with id: #{order.id} is confirmed\nit will be sent
    end
  end
end
```

The hexagonal

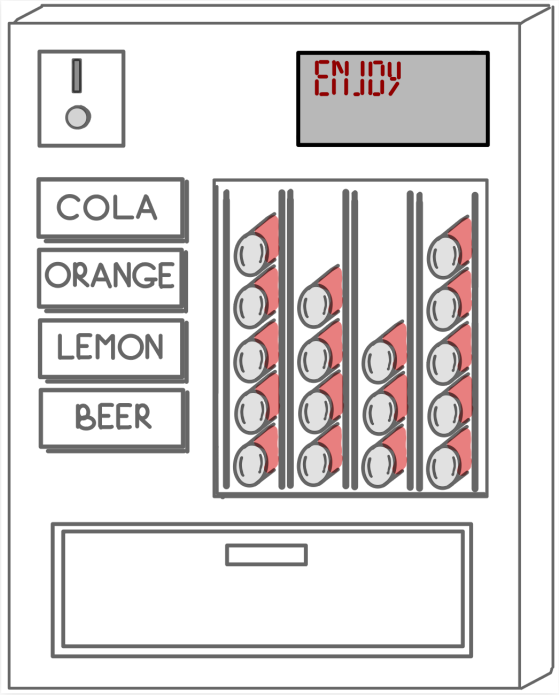
```ruby
module Adapters
  class OrderRepository
    attr_reader :db
    def find_by_id(order_id)
      db.execute("select ....").map { |e| as_order(e) }.first
    end
    def save(order)
      db.execute("update orders set description = '#{order.descripti
    end
    private
    def as_order(rs)
      Domain::Order.new(rs['order_id'], rs['order_description'], rs[
    end
    def as_customer(rs)
      Domain::Customer.new(rs['customer_id'], rs['customer_name'], r
    end
```

The hexagonal



Exercise:

Design controller software for a vending machine



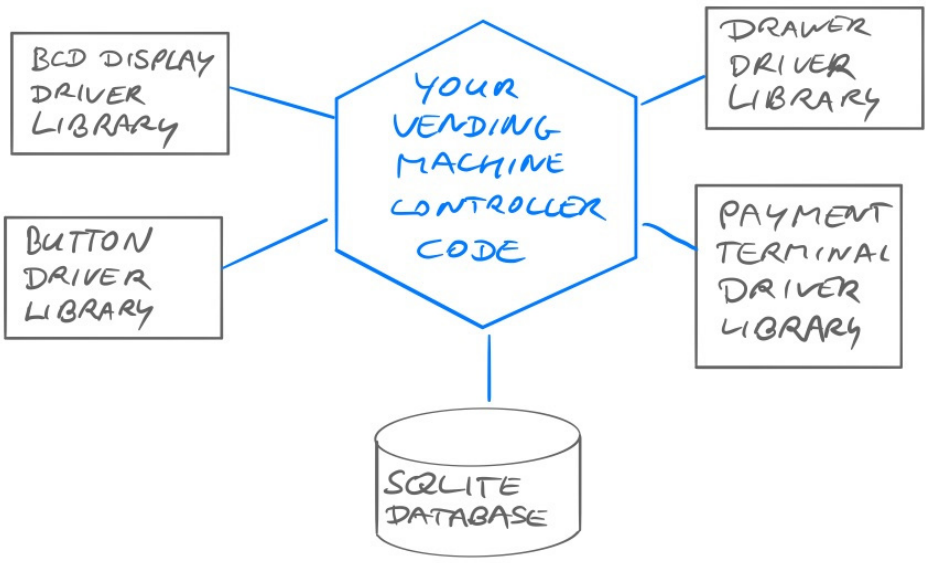
Scenarios

Get Drink: thirsty person inserts a euro in the machine, chooses a cola, pulls the drink from the bin & leaves happy.

Return Change: thirsty person inserts 2 euro, chooses a cola, presses the *return change* button and picks both the can and the change from the bin.

Out of Stock: thirsty person inserts a euro and chooses a cola; machine indicates it's out of stock; person asks her change back, and leaves unhappy.

Controller context



Driver APIs

```
public interface ButtonsLib
    int GetButtonState(int buttonNo); // -1 = down; 0 = up
public interface CoinMachineLib
    void OnCoinInserted(Action<int> callable);
    void OnChangeRequested(Action callable);
    void ReturnChange();
public interface DispenserLib
    void Release(int number);
    void Reset(int number); // reset 500ms after OnDropped event to
    void OnDropped(Action callable);
public interface DisplayLib
    void Show(int line, string message);
    // 2 lines of 16 characters; crashes if length > 16
public interface BinLib
    int GetState(); // -1 = full, 0 = empty
```

Exercise:

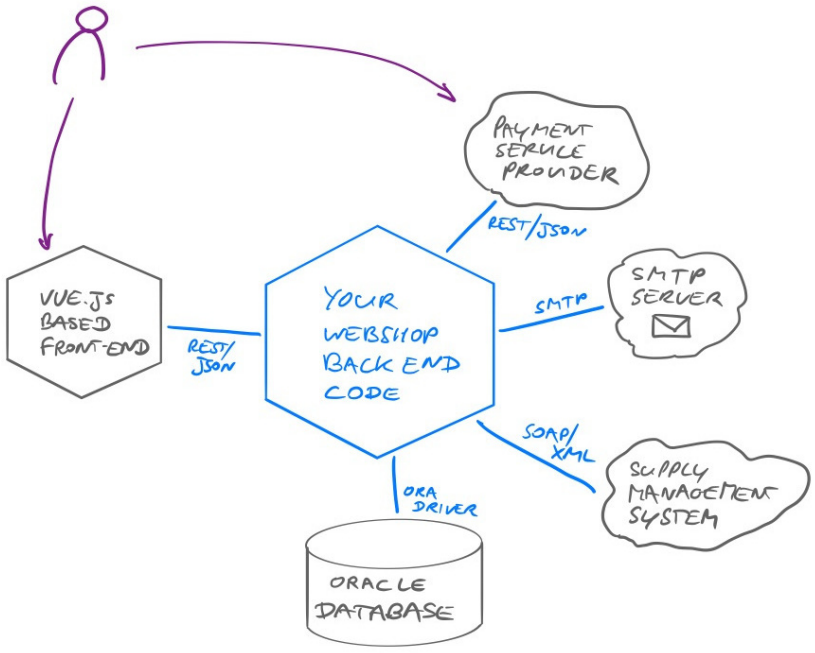
Design the backend for a web shop

Drinks Inc allows you to browse through different brands of drinks, search & filter drinks on name, description, price, and category. You can also see product details.

You can add a drink to your shopping basket, increase/decrease the number of products, or remove it. You can inspect the shopping basket, to see all selected products, with quantities and total price.

You can check out the contents of your shopping basket. You will be redirected to the payment provider, where you can pay by credit card and many other payment methods. When the payment is finished (successful or failed), you are redirected back to Drinks Inc.

Backend context



The back end component provides APIs for the front end:

- **Products API** - list products, get details, price & availability
- **Products query API** - offers rich query functionality; returns lists of matching products
- **Shopping basket API** - add, update, remove products, get details like total price
- **Checkout API** - checkout & get payment state information

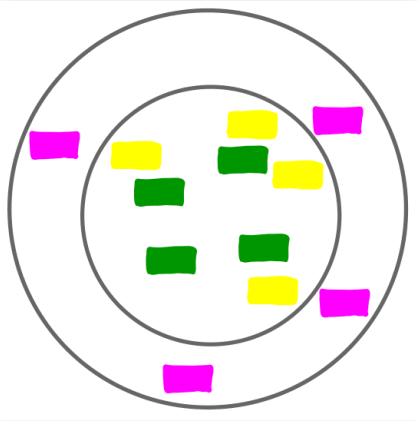
These are all HTTPS/REST/JSON based.

Assignment

Work in groups of 3 or 4, use flipchart

Create a conceptual design that realizes the scenarios

Think hexagonal



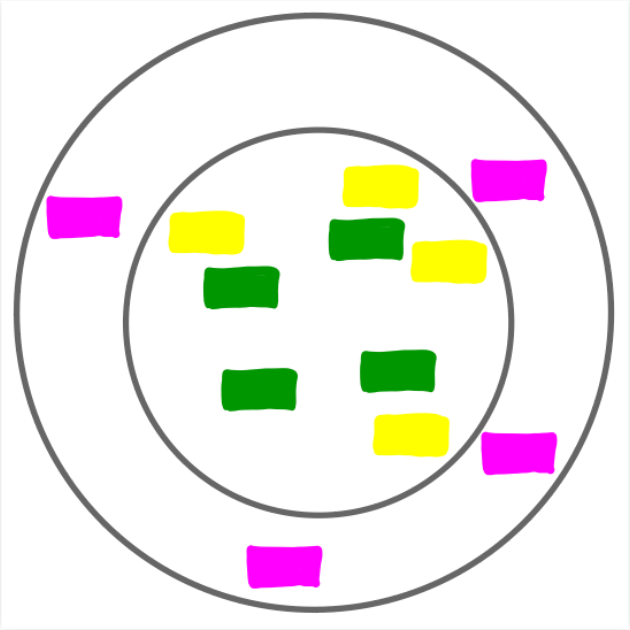
Different colors for different classes/components/modules:

What do the ports/interfaces look like?

Add *uses* & *implements* dependencies
use arrow shaped post-its

How does your design realize the 3 scenarios?

Debrief



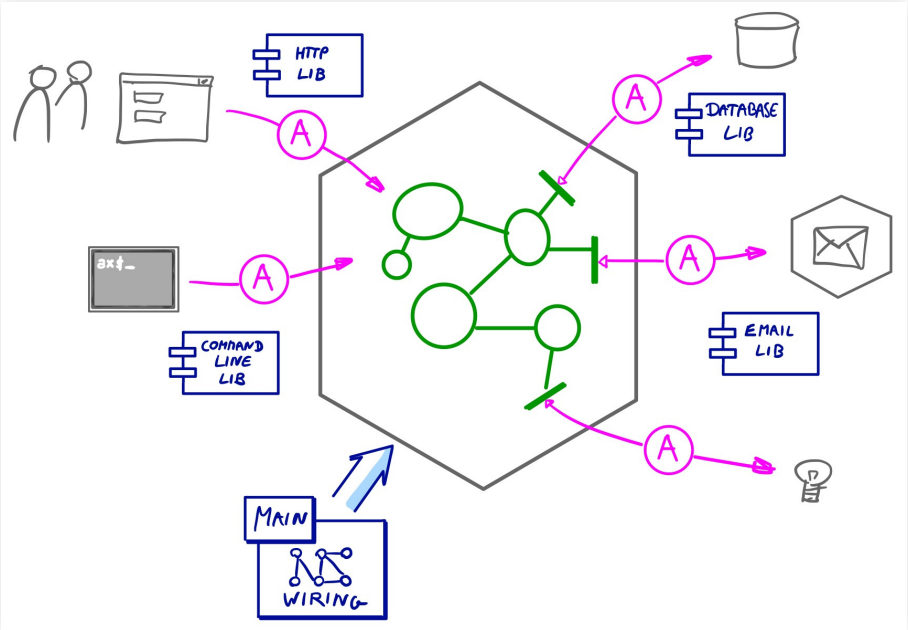
Show & tell!

What about

```
int main(void) {
  ...
}
```

?

Wiring code & frameworks live outside

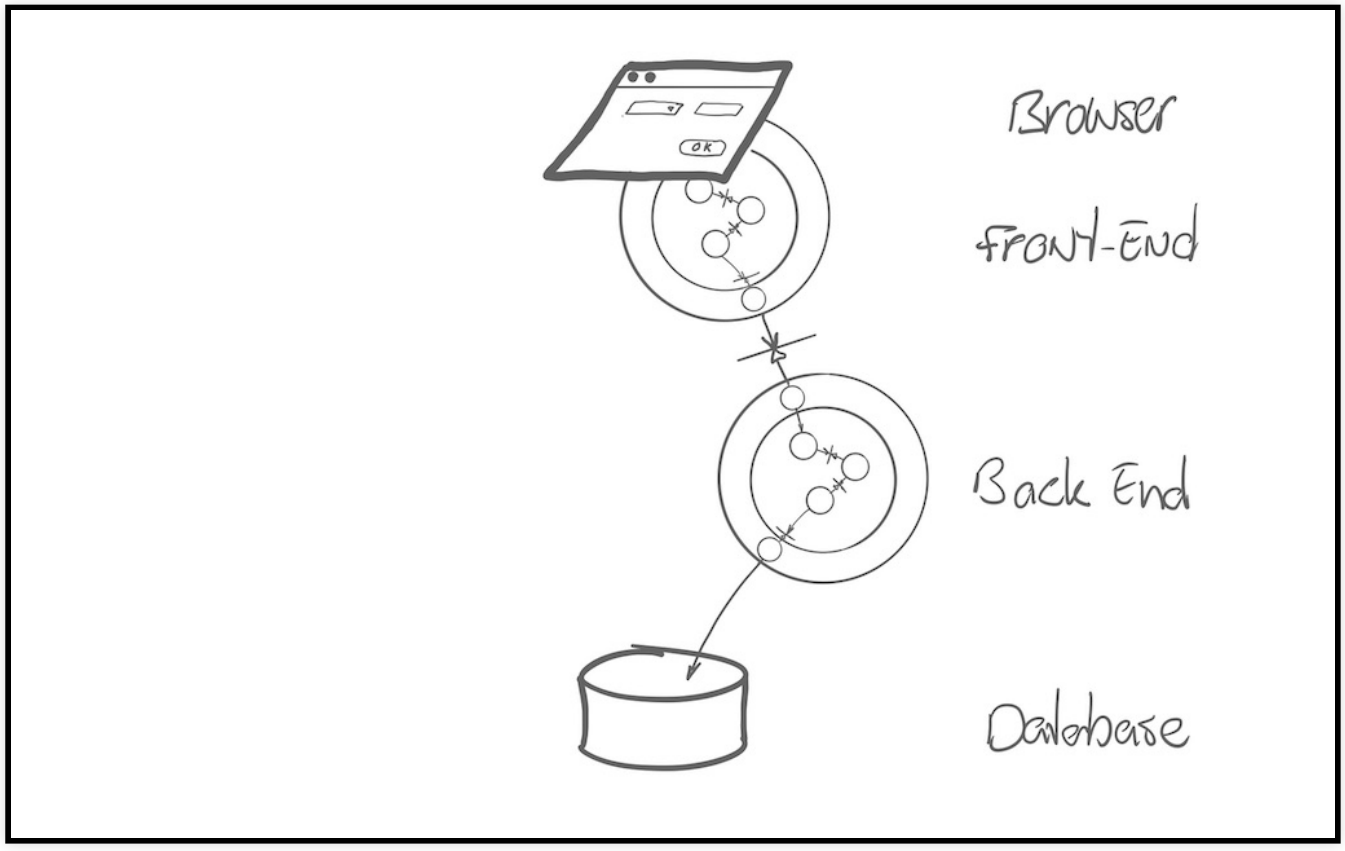


This includes dependency injection!

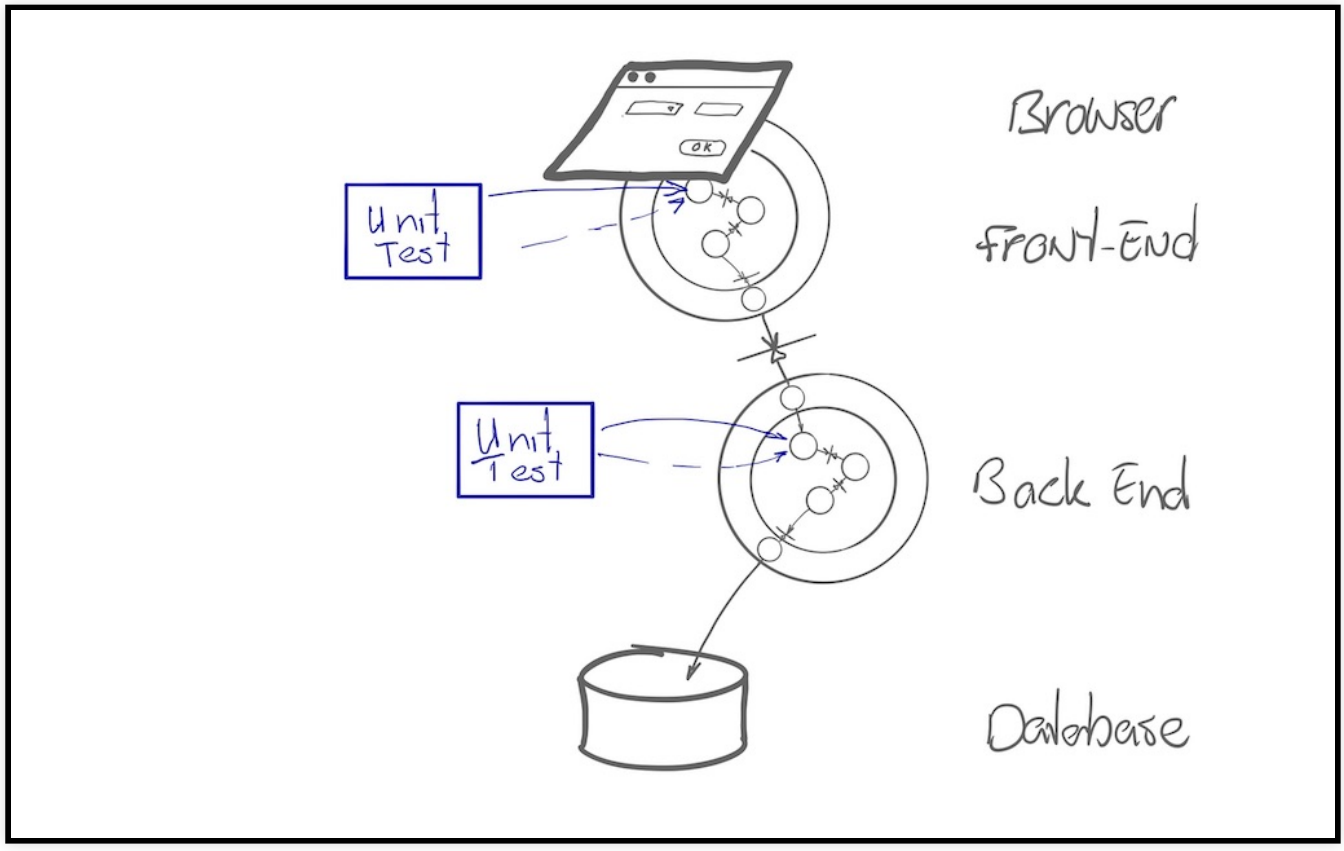
Test architecture

How do automated tests fit in?

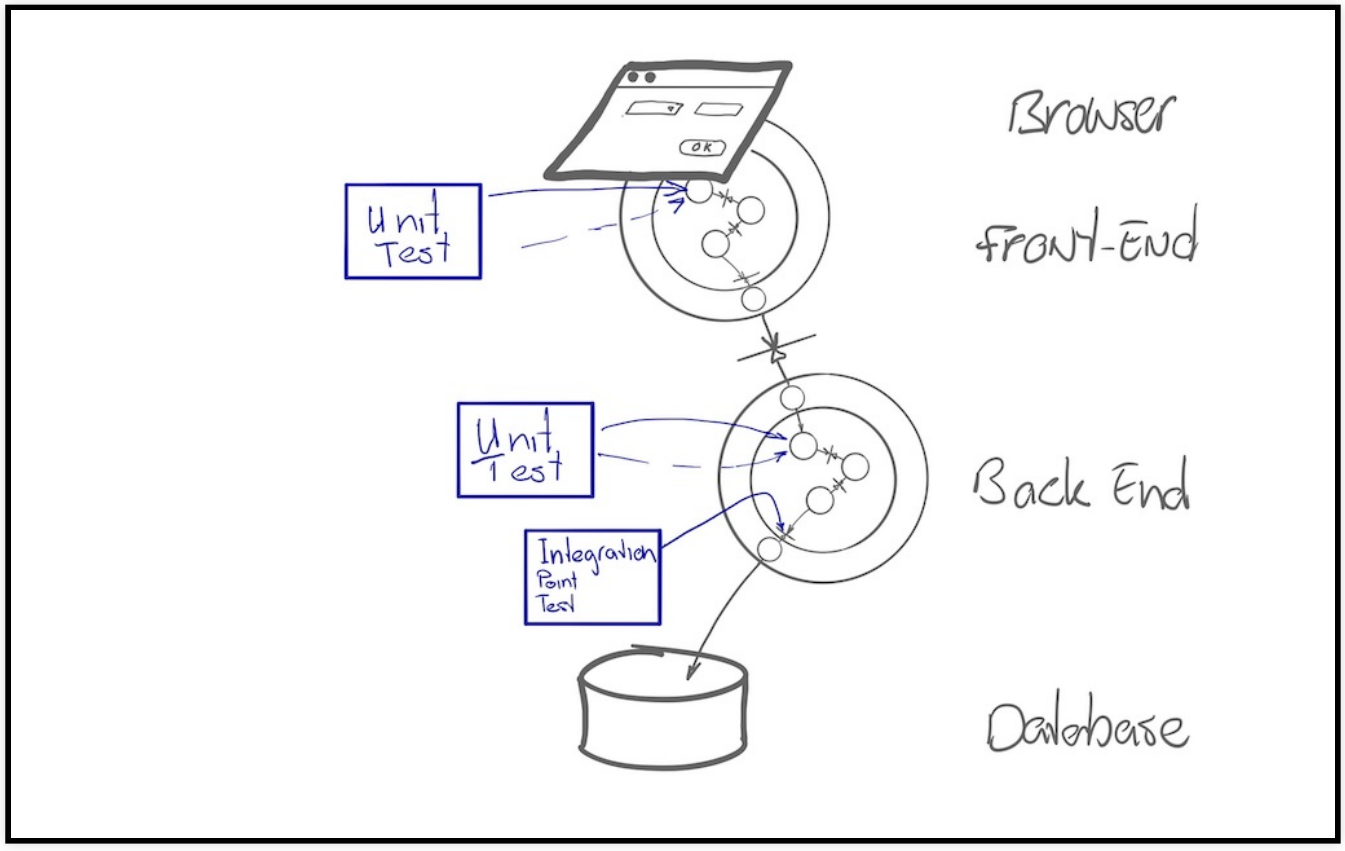
Our architecture



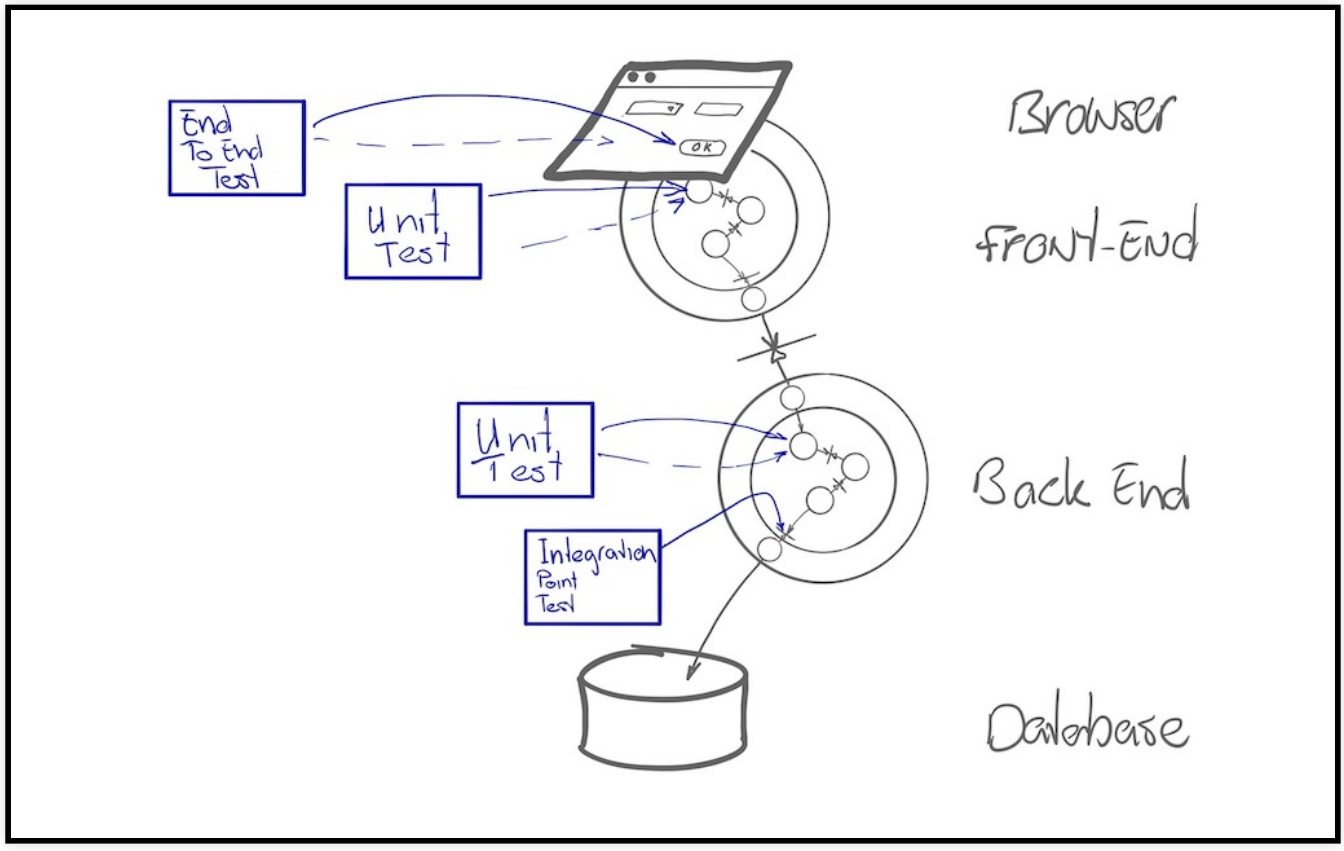
Unit tests



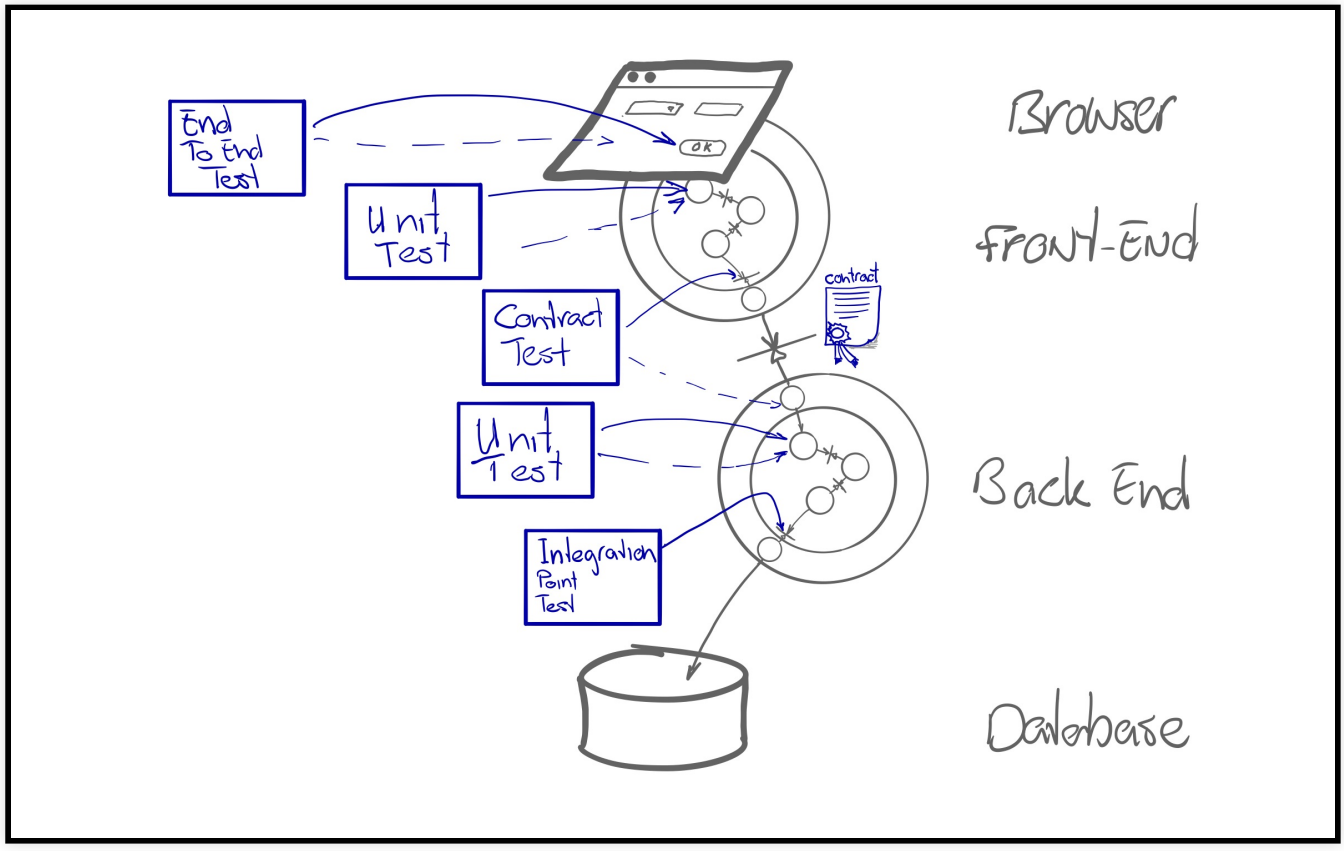
Integration tests



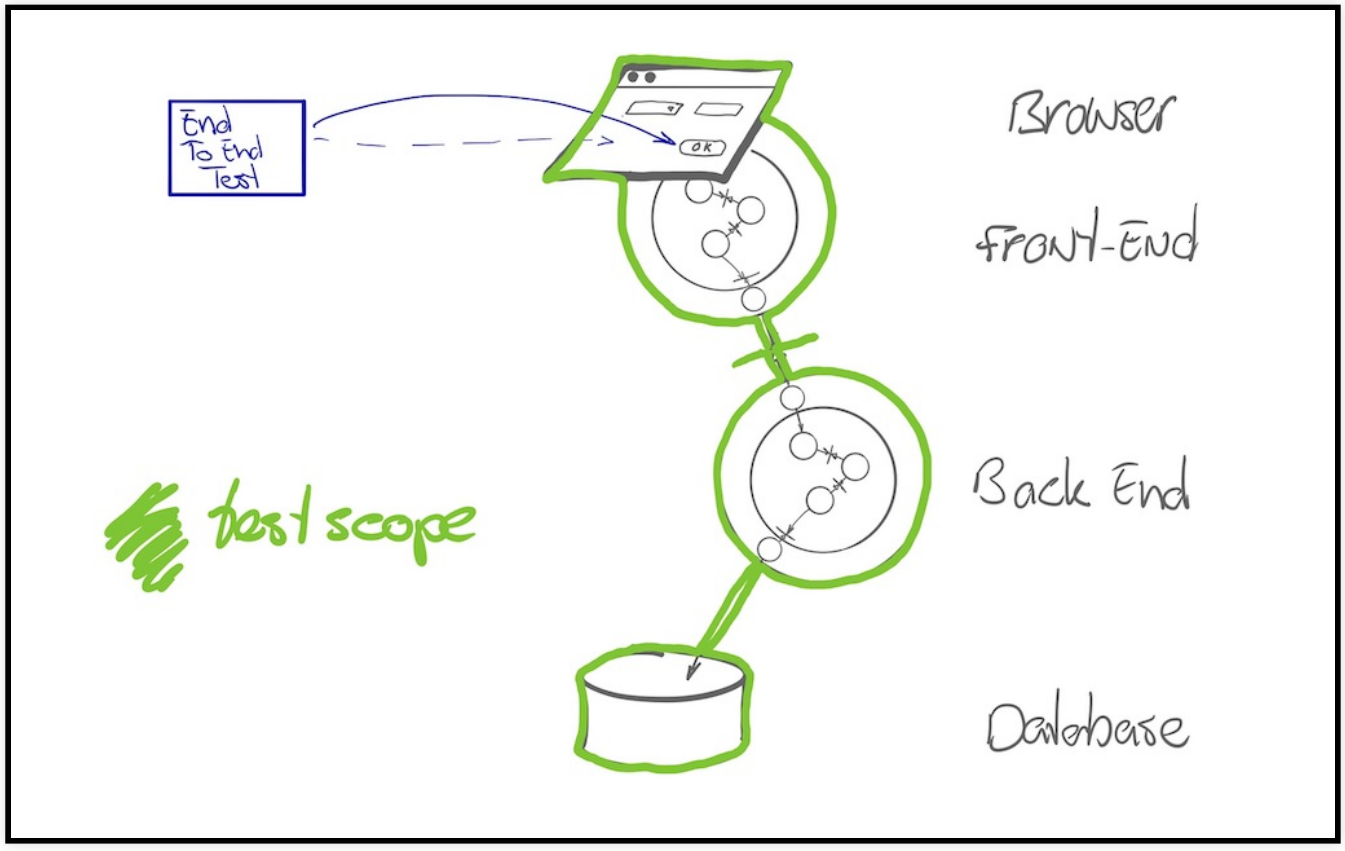
End to end tests



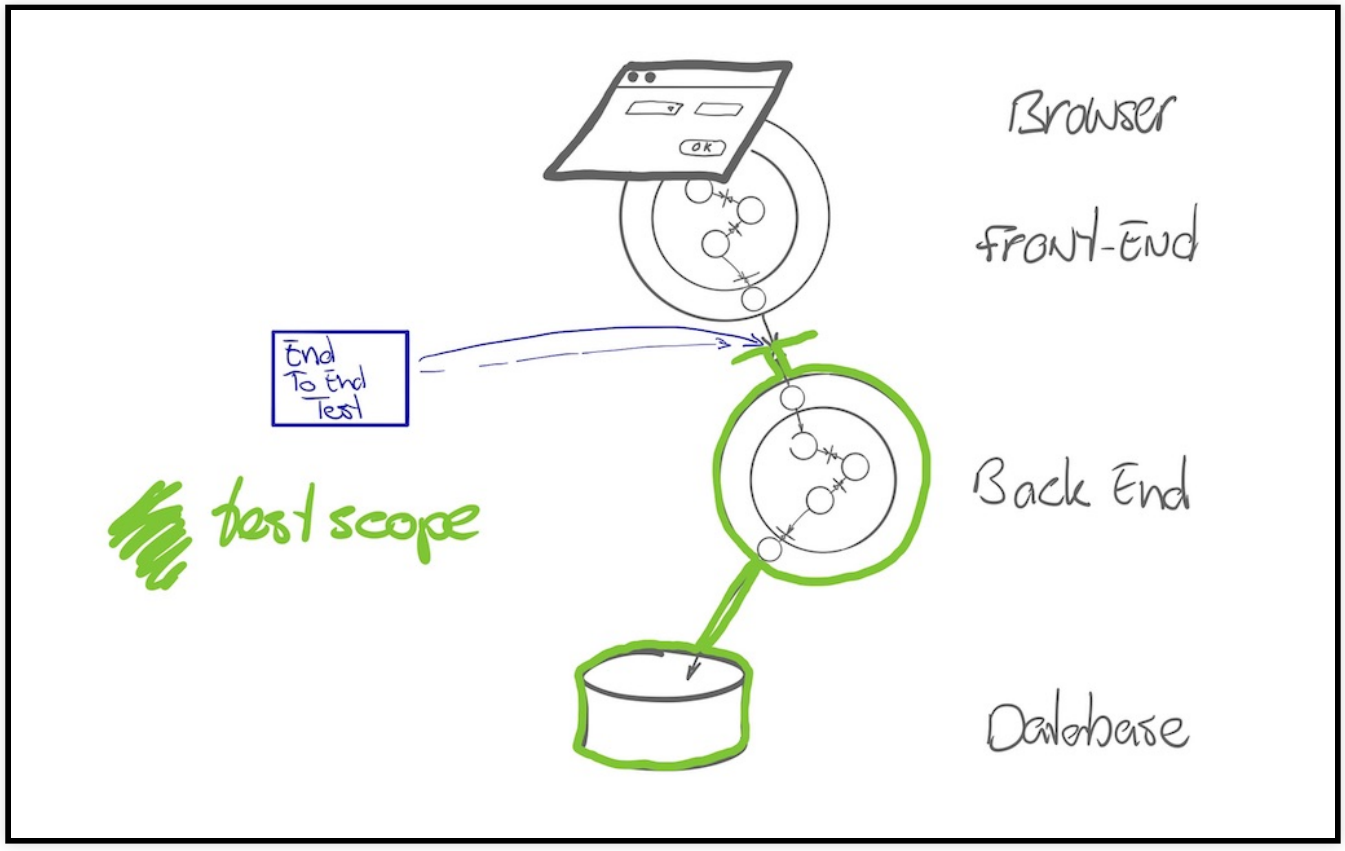
Contract tests



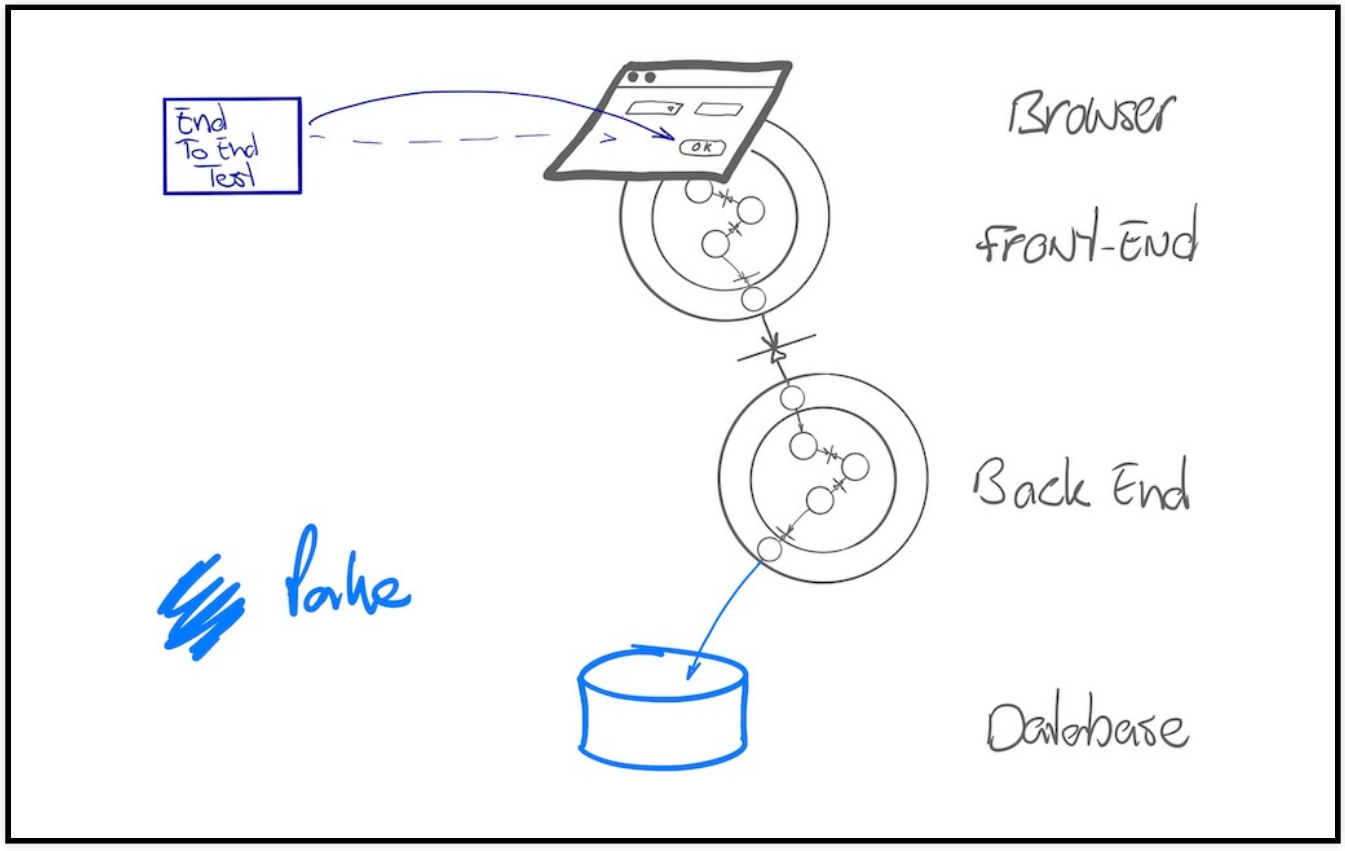
End to end tests



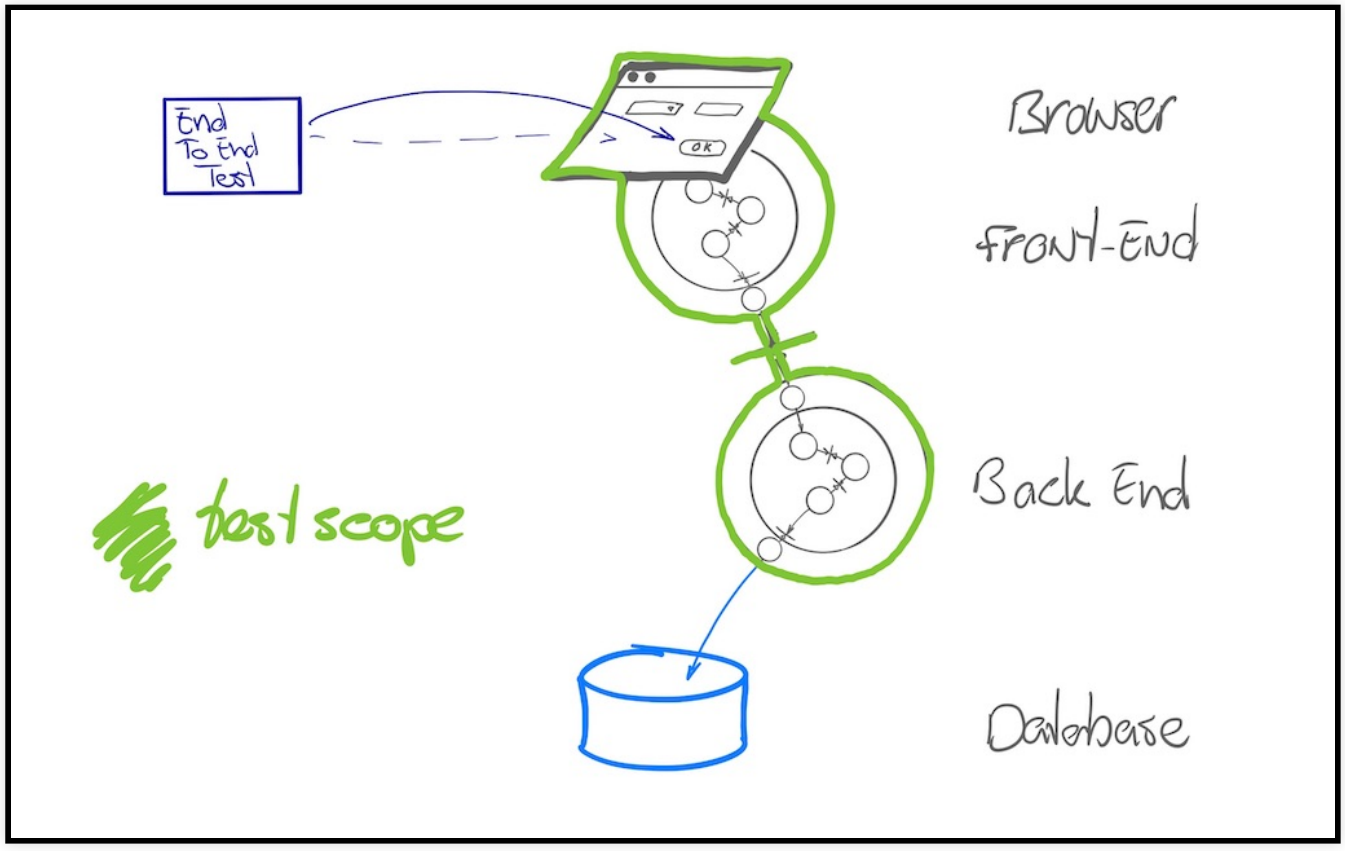
Choose end to end-ness



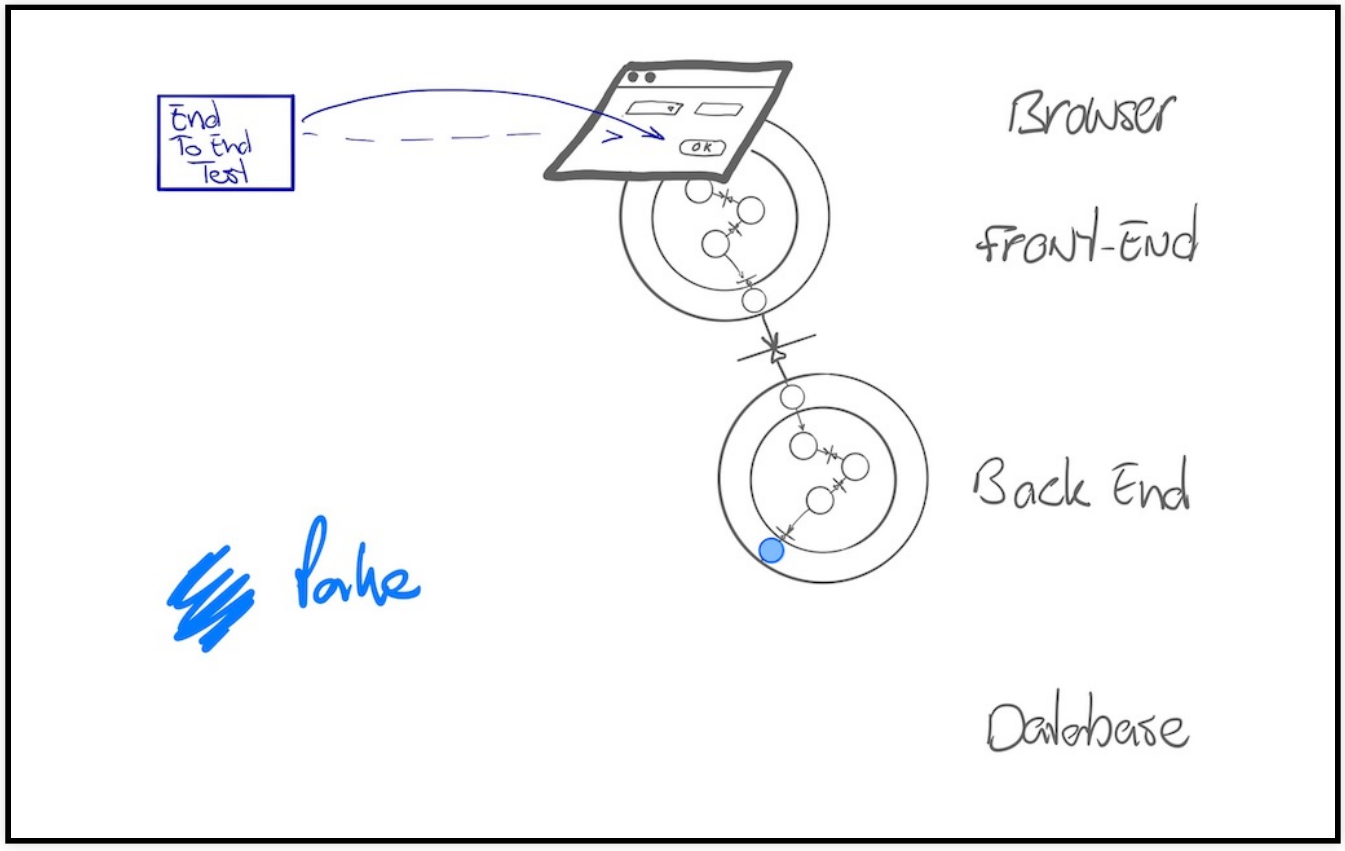
Use fake database



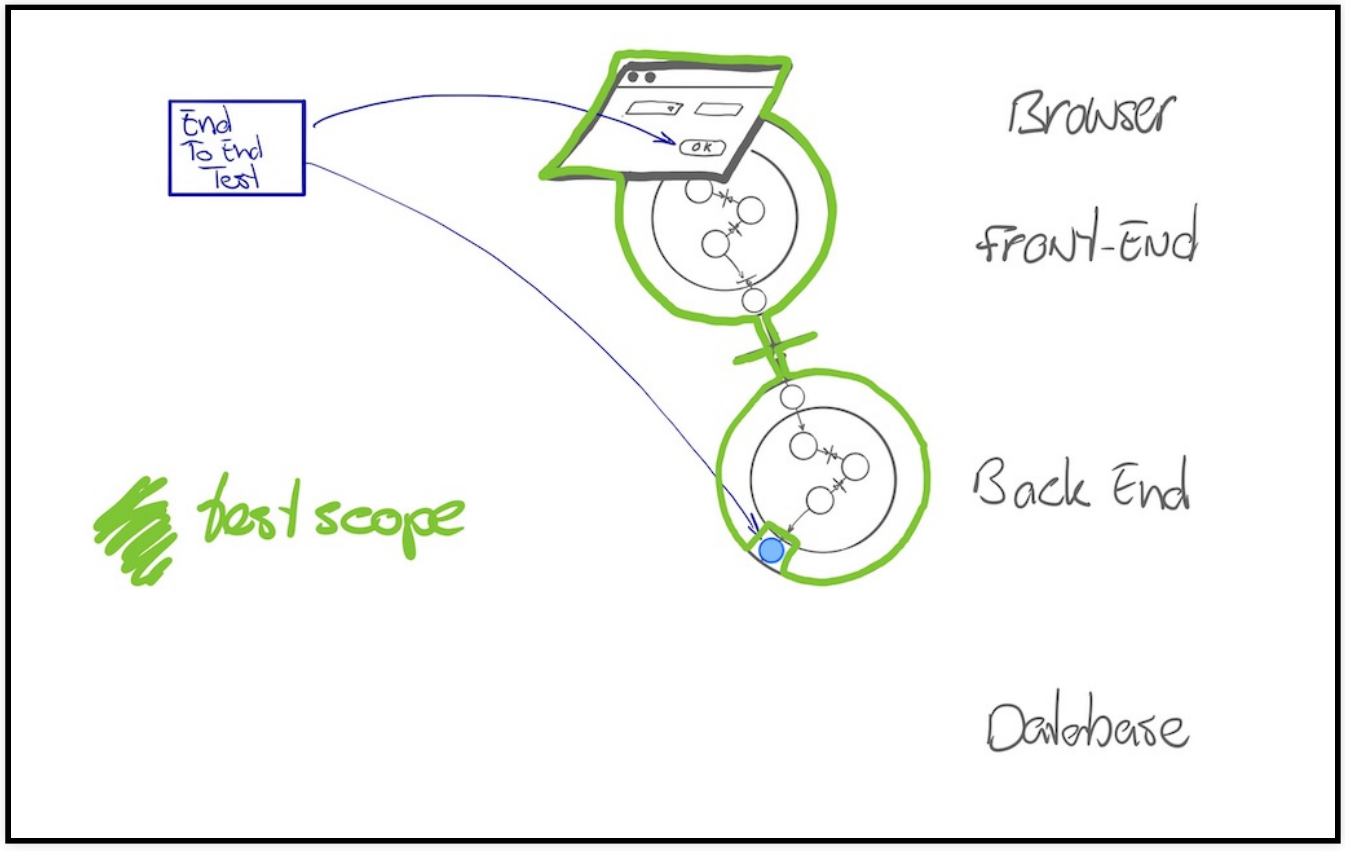
Use fake database



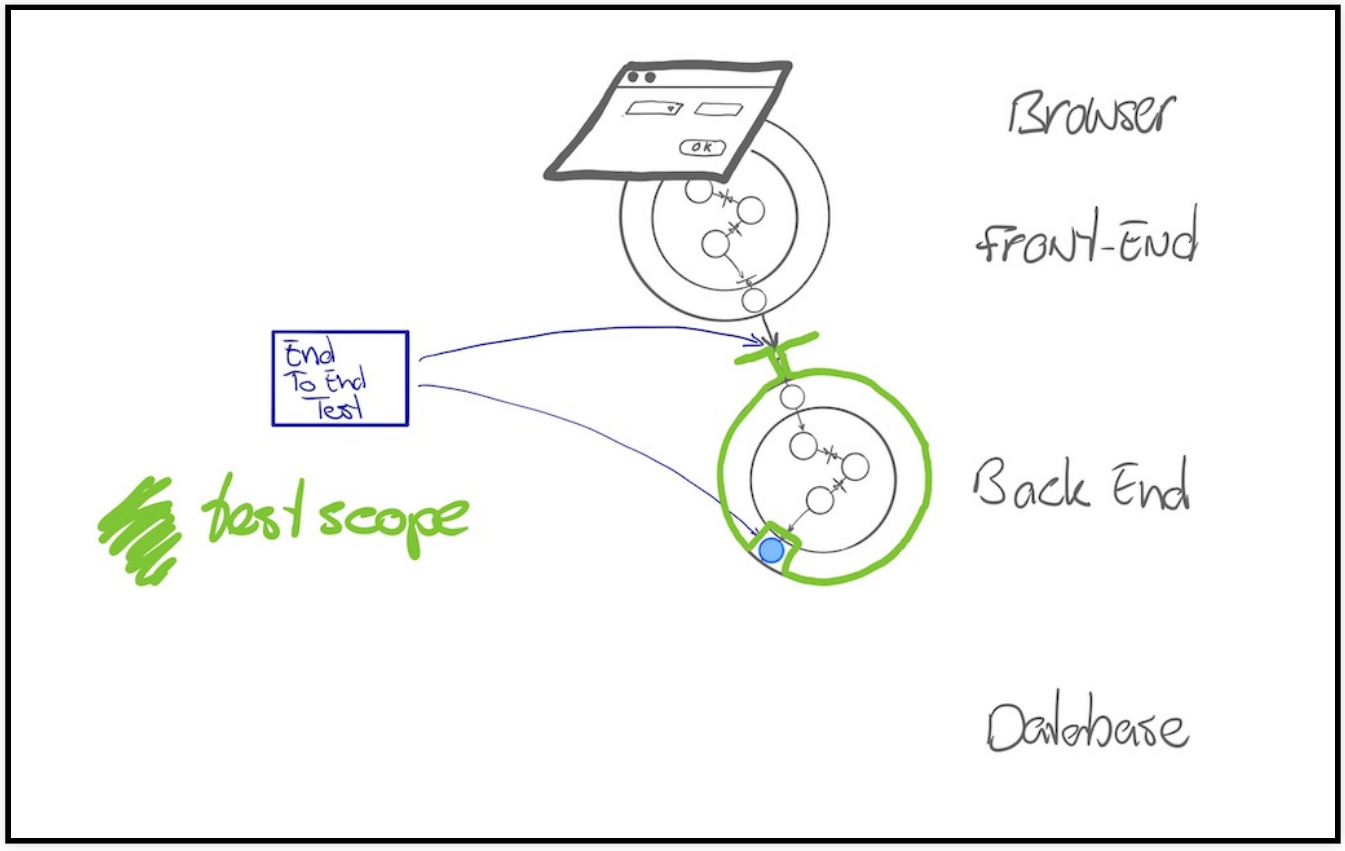
Or fake repository



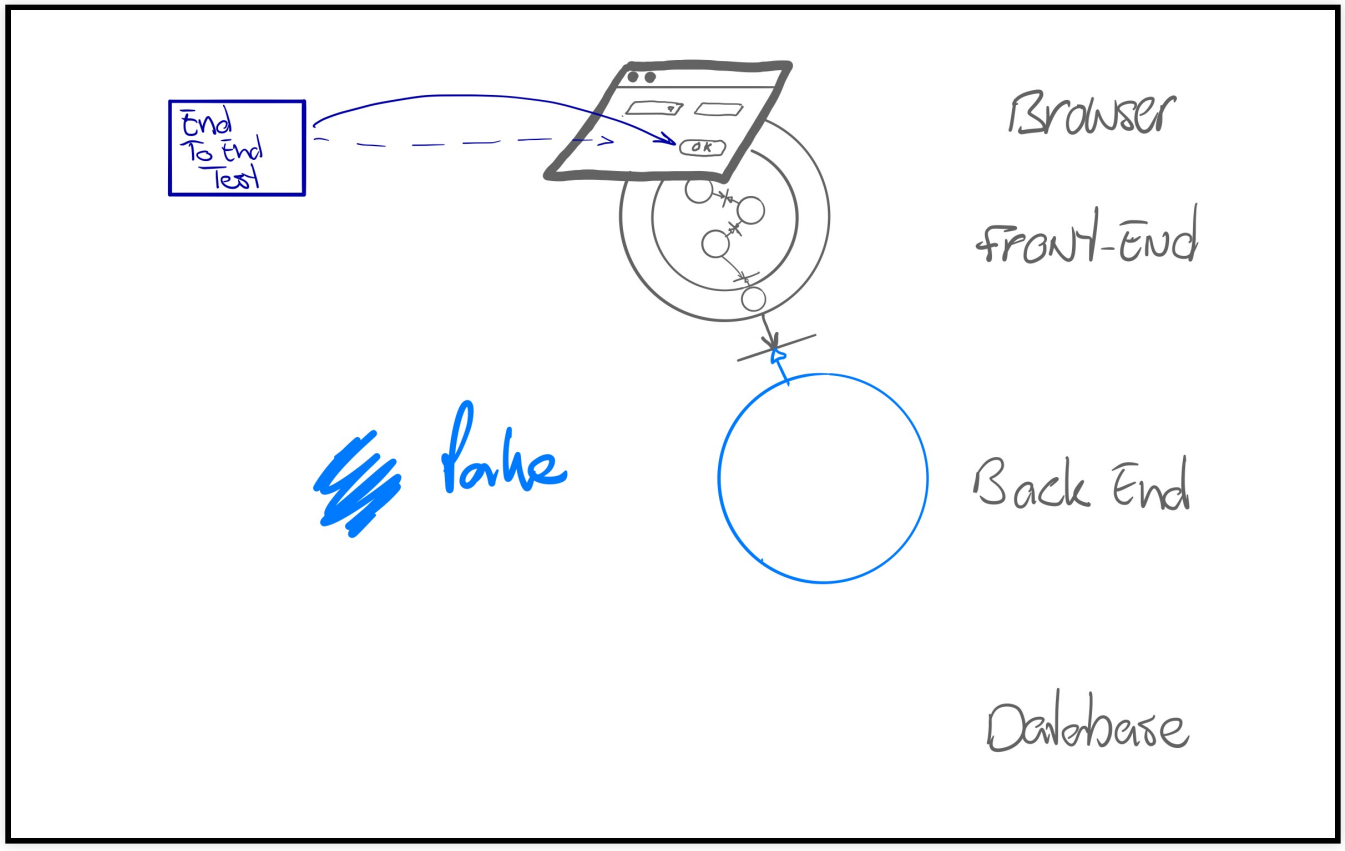
Or fake repository



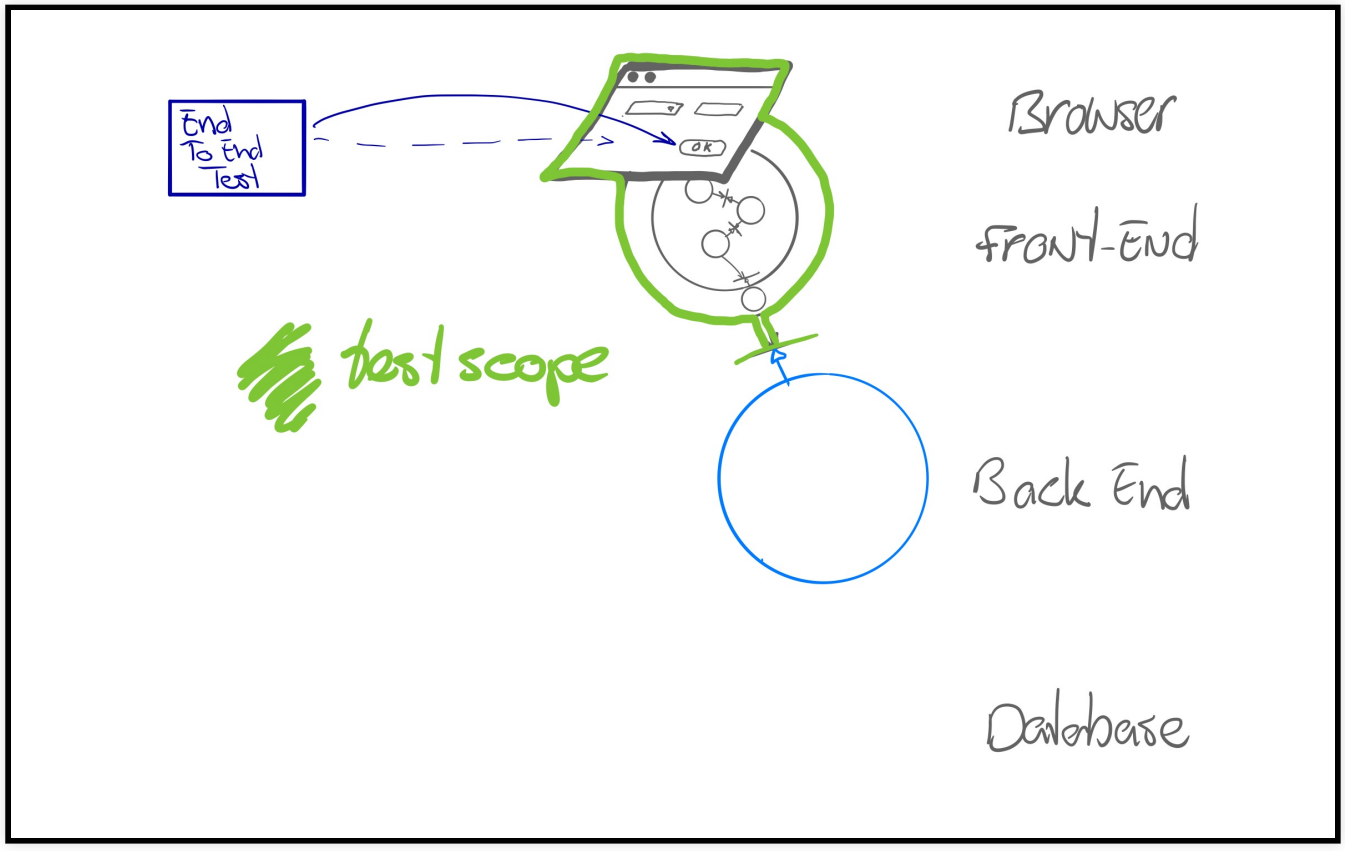
And choose end to end-ness



Or fake backend



Or fake backend



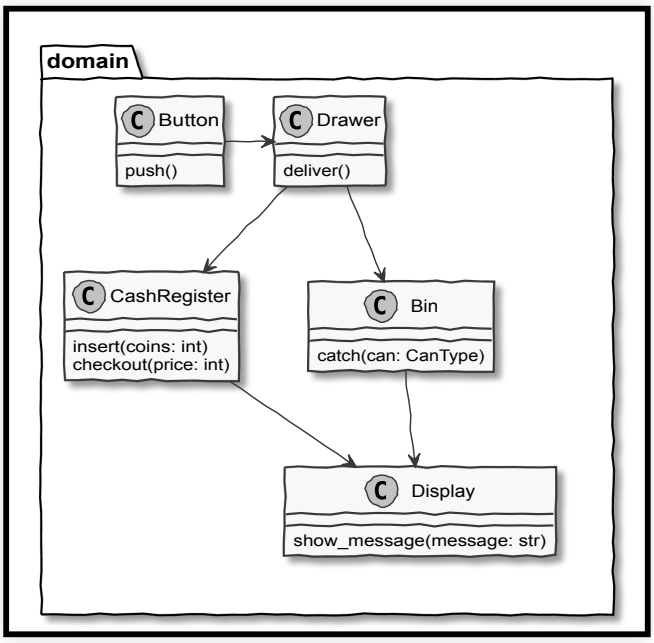
Code exercises

Analyse the code

- start VendingMachine and play around
- have a look at the code
- note what you recognise
- report back

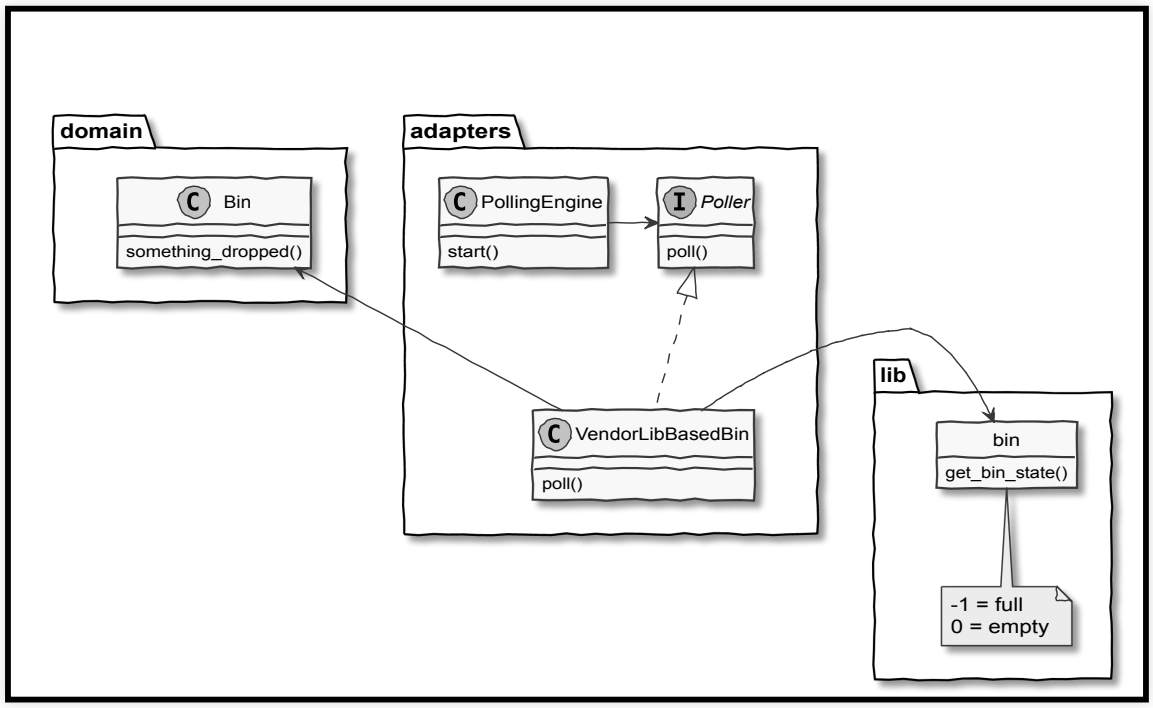
The Domain Package

@startuml embedded-machine skinparam { handwritten true monochrome true } package domain { together { class Bin { catch(can: CanType) } class Display { show_message(message: str) } } class Drawer { deliver() } class Button { push() } class CashRegister { insert(coins: int) checkout(price: int) } Button -> Drawer Drawer --> CashRegister Drawer --> Bin CashRegister --> Display Bin --> Display } @enduml



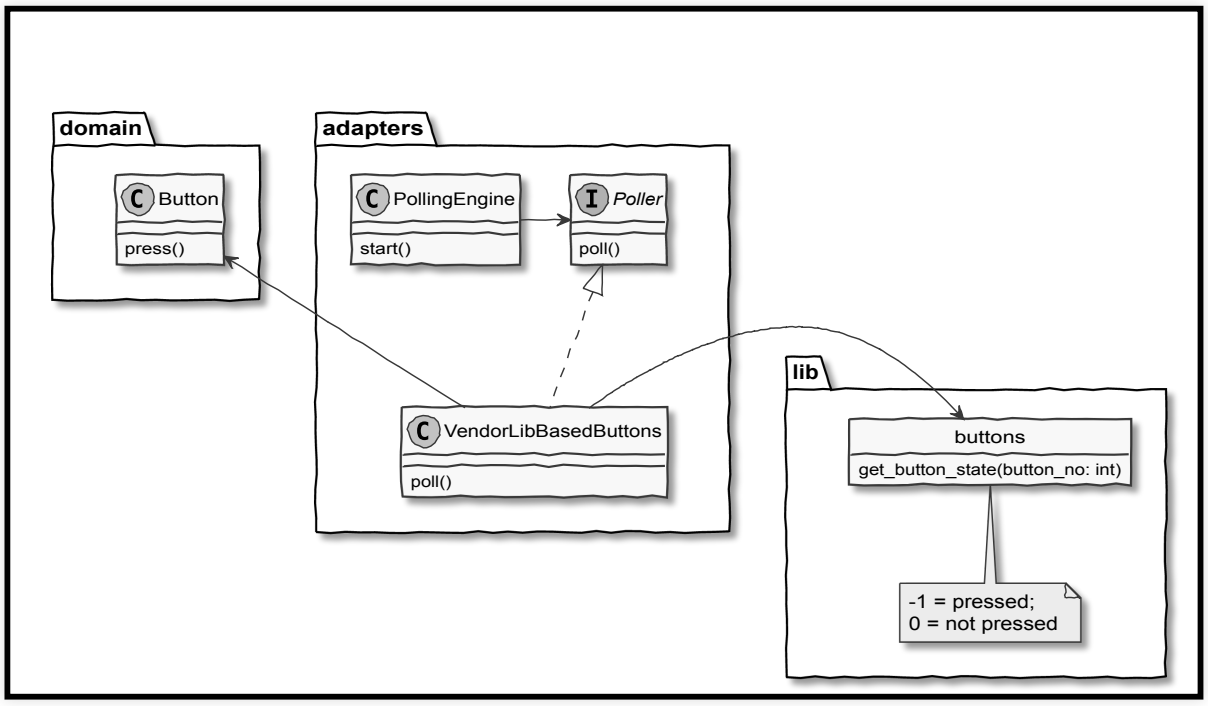
Bin Adapter

@startuml embedded-machine-bin-adapter skinparam { handwritten true monochrome true } package lib { class bin { get_bin_state() } note bottom: -1 = full\n0 = empty hide attributes hide circle } package domain { class Bin { something_dropped() } } package adapters { class PollingEngine { start() } interface Poller { poll() } class VendorLibBasedBin { poll() } } domain ..[hidden]right> adapters adapters ..[hidden]right> lib PollingEngine -> Poller Poller <|.. VendorLibBasedBin VendorLibBasedBin -> bin Bin <- VendorLibBasedBin @enduml



Buttons Adapter

@startuml embedded-machine-buttons-adapter skinparam { handwritten true monochrome true } package lib { class buttons { get_button_state(button_no: int) } note bottom: -1 = pressed;\n0 = not pressed hide attributes hide circle } package domain { class Button { press() } } package adapters { class PollingEngine { start() } interface Poller { poll() } class VendorLibBasedButtons { poll() } } domain ..[hidden]right> adapters adapters ..[hidden]right> lib PollingEngine -> Poller Poller <|.. VendorLibBasedButtons VendorLibBasedButtons -> buttons Button <- VendorLibBasedButtons @enduml



Coding exercises hands-on

Stories:

- Adding inserts (difficulty 1)
- Credits on display (difficulty 2)
- Feedback on checkout (difficulty 2)
- Empty Bin (difficulty 3)
- Request Change (difficulty 4)

Adding inserts (difficulty 1)

As a thirsty person
I want to use multiple times one coin
So that I can get rid of my small change

Acceptance

- can buy cola with multiple inserts of one coin

Credits on display (difficulty 2)

As a thirsty person
I want to see how much I have inserted
So that I know how many cans I can still buy

Acceptance

- credits on display

Feedback on checkout (difficulty 2)

As a thirsty person
I want to feedback on a failing checkout
So that I know how much money I need to add

Acceptance

- credits on display

Empty Bin (difficulty 3)

As a thirsty person
I want to get feedback on an emptied bin
So that I know that I can buy another can

Acceptance

- display emptied on 'take'ing a can from the bin

Request Change (difficulty 4)

As a thirsty person
I want to get change back
So that I buy other stuff

Acceptance

- get change in bin on return change request
- indicate change is coming (similar to indicating can is coming)
- "take change" on display

References (1)

- Alistair Cockburn, Hexagonal Architecture http://alistair.cockburn.us/Hexagonal+architecture
- Alistair Cockburn presenting the Hexagonal architecture:
  1: https://www.youtube.com/watch?v=th4AgBcrEHA&t=4s
  2: https://www.youtube.com/watch?v=iALcE8BPs94
  3: https://www.youtube.com/watch?v=DAe0Bmcyt-4
- Juan Manuel Garrido de Paz, Ports and Adapters Pattern (Hexagonal Architecture) https://softwarecampament.wordpress.com/portsadapters

References (2)

- Robert C. Martin, The Clean Architecture - similar to the hexagonal architecture, same intent - http://blog.8thlight.com/uncle-bob/2012/08/13/the-clean-architecture.html
- Jim Shore, The A-Frame Architecture - an alternative way of structuring your application/component and achieve decoupling - https://www.jamesshore.com/Blog/Testing-Without-Mocks.html#a-frame-arch

References (3)

- Hexagonal Architecture https://www.qwan.eu/2020/08/20/hexagonal-architecture.html
- How to decide on an architecture for automated tests https://www.qwan.eu/2020/09/17/test-architecture.html
- How to keep front end complexity in check with hexagonal architecture https://www.qwan.eu/2020/09/25/hexagonal-frontend-example.html
- A hexagonal vue.js front-end, by example https://www.qwan.eu/2020/09/09/how-to-keep-complexity-in-check-with-hexagonal-architecture.html

Books

- Robert C. Martin, Clean Architecture, 2017
- Steve Freeman & Nat Pryce, Growing Object Oriented Software, Guided by Tests, 2009