Look, no database

This talk is about the challenges, learnings, upsides, and downsides of building a completely custom E-Commerce experience with Symfony, while using only web services, like Commercetools Platform (E-Commerce) and Prismic.io (CMS).

We'll talk about what Multi Channel Solutions are, and what they are useful for. We'll also talk about how we integrated these services with Symfony and what the challenges were with project structure and abstractions.

No databases were harmed in the making of this project.

Hi, my name is Christoph Hochstrasser and I’m from a little city called Waidhofen an der Ybbs in Lower Austria.

I’m a web developer since over 10 years now, most of the time having worked with PHP. I’m the creator of several open source projects, like phpenv, php-build, the StackPHP project, and a Heroku Buildpack for PHP.

You can find me as @hochchristoph on Twitter and @CHH on Github.

I’m now freelancing since two years. I’m making websites, creating custom web apps with Symfony and PHP, and I accidentally got into making webshops, though I enjoy it.

You can find my company’s website on https://hochstrasser.io.

Open Source

- StackPHP
- phpenv
- php-build

Freelancer

- Fast, accessible, responsive Websites
- Customized Webshops
- Custom WordPress Plugins and Themes
- Product Development Services

About Keynet

- Main partner in this project
- Based in Amstetten in Lower Austria
- Agency specialized on E-Commerce
- keynet.at

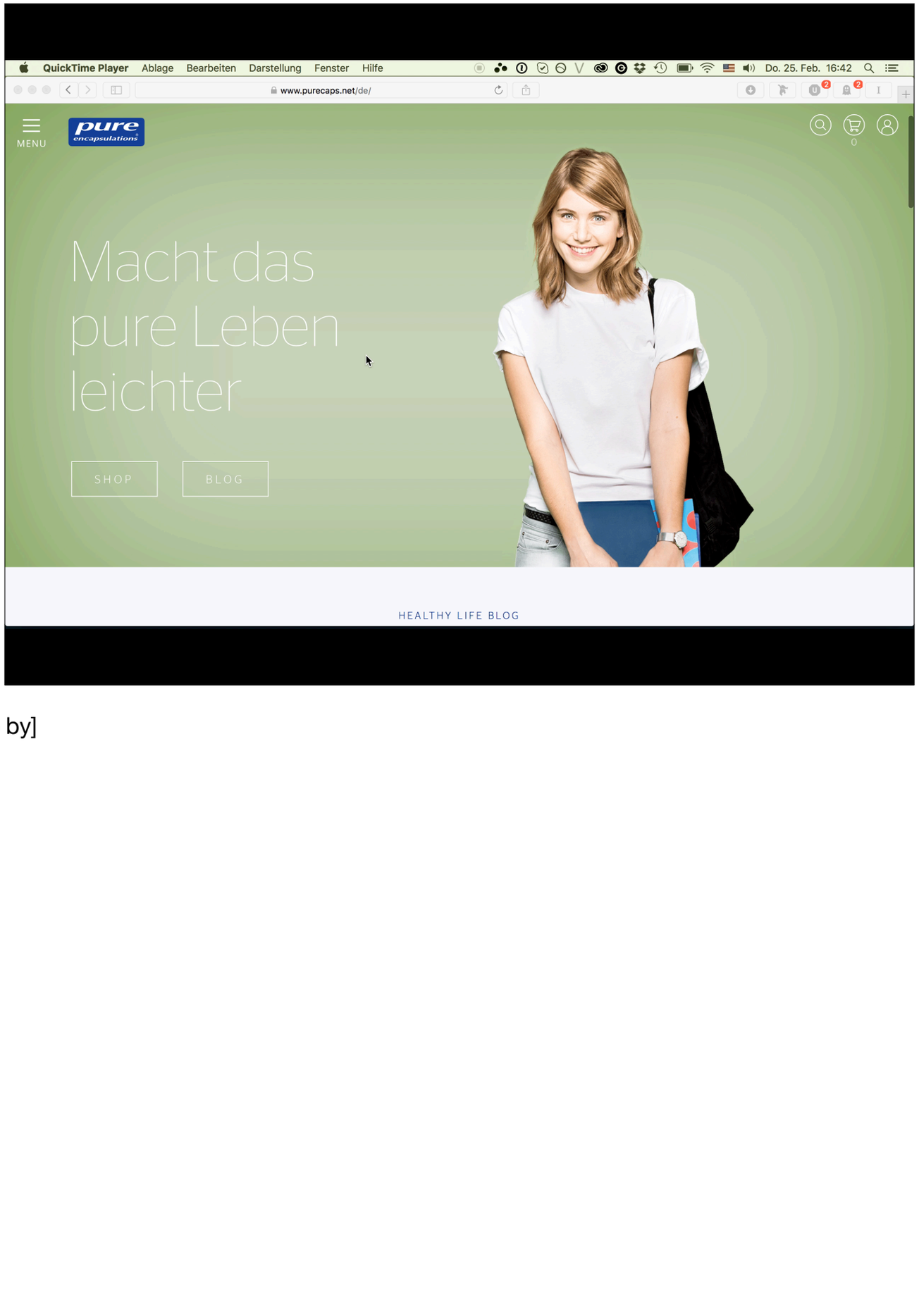
I didn’t do a project of this size alone. I have partnered here with Keynet GmbH, an agency specialized on E-Commerce projects, based in Amstetten, Lower Austria which provided project management and coordination.

The Project

Pure Encapsulations is a maker of high quality hypoallergenic, research-based dietary supplements, like Vitamins and based in the US. The branch responsible for distribution in Europe is based in Graz.

Pure Encapsulations

- High quality dietary supplements
- Over 100+ products
- Manufactured in the US



[Show off purecaps.net to make time go by]

The Problem

- Distribution in whole of Europe, currently 21 countries
- Multiple Languages
- Custom checkout experience
- Content integration
- Not reinventing the wheel too much

In 2014 Pure Encapsulations came to a Keynet with a project. They wanted a completely custom shop experience with their own checkout for the distribution of their products in Europe.

Also they had a backend in SAP which they wanted to integrate and some interesting requirements on handling their customers.

The initial plan was to serve Austrian, German, and Italian customers with the initial rollout. Later in 2015 we rolled out the shop for nearly all of Europe, though currently only in the German language while we try to find and iron out issues through early 2016.

We knew, this has to be coded custom. Adapting an existing solution would be as much or more work than writing it from scratch, with a hell of a lot more messy code, and a system which is very tough to maintain.

What we really didn’t want though, was reinventing the wheel completely. Some things in E-Commerce are a lot of boring work. Like how prices and taxes are defined, or how product data looks.

Also we wanted to easily integrate with any ERP system the customer would have. So cloud based with a sane API would also be very helpful.

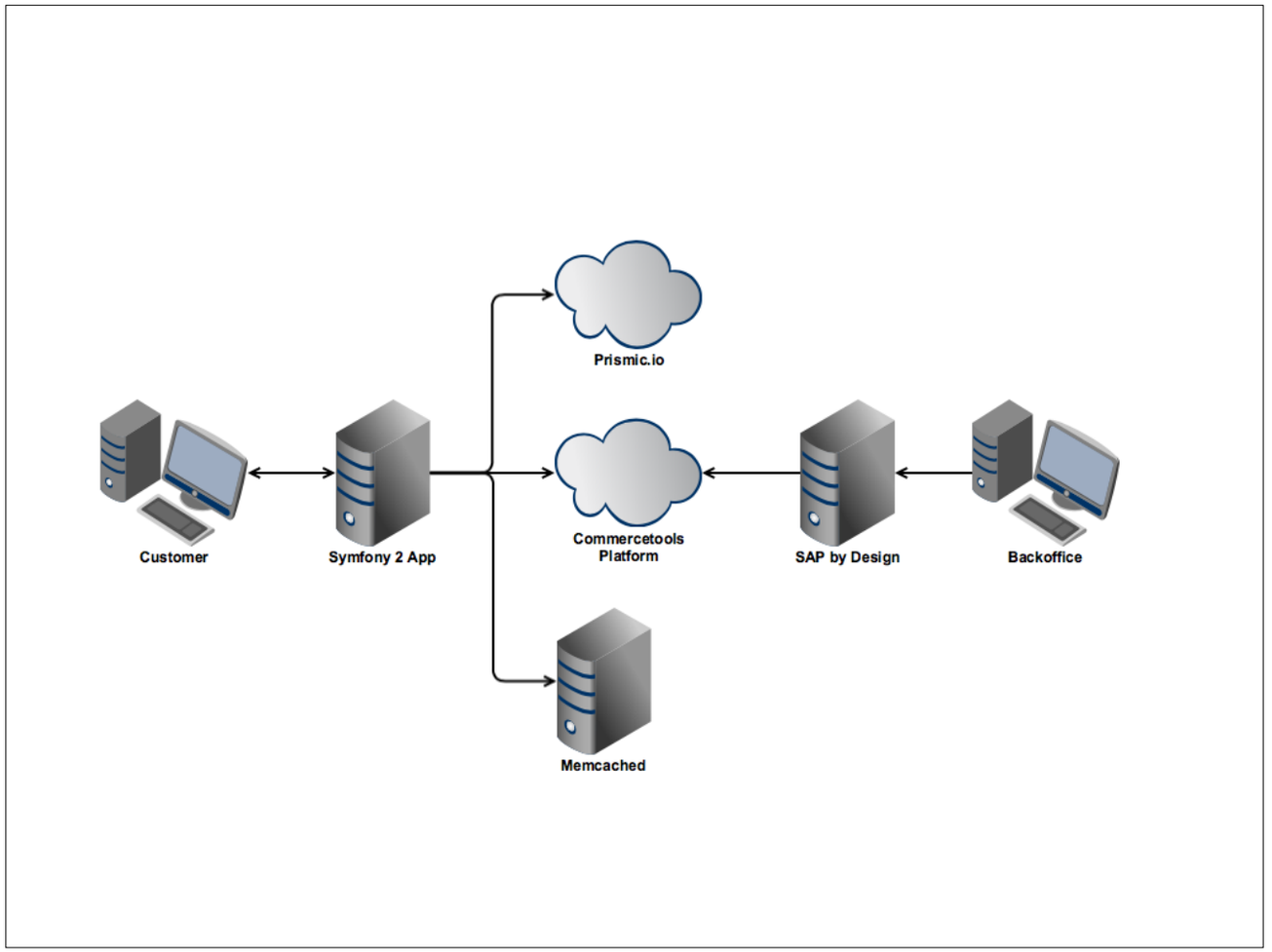
Then we discovered sphere.io (now called Commercetools Platform).

Just what we looked for.

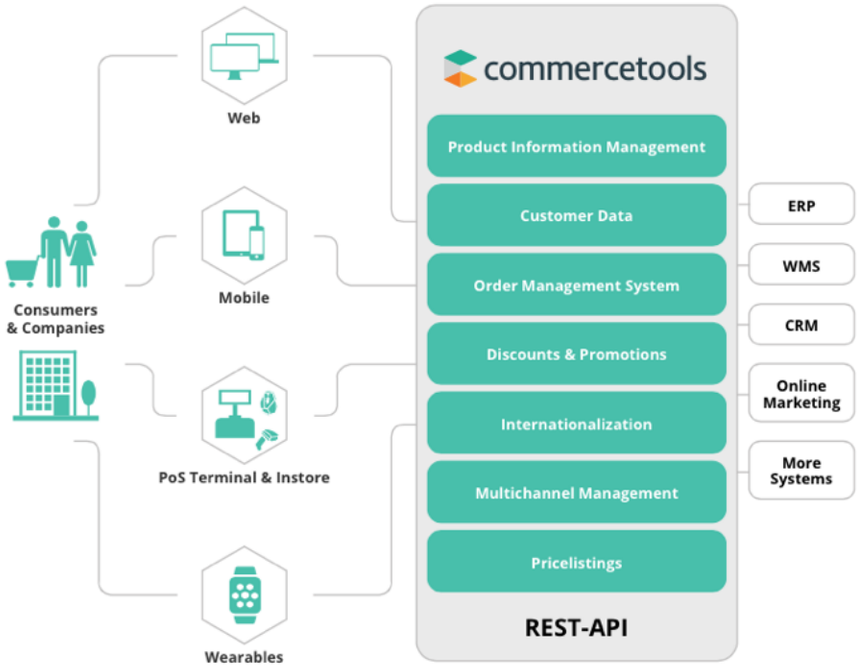
System Overview

This is the stack we ended up with:

- Symfony 2 Standard Edition (-Doctrine ORM Bundle), plus common tools in our own bundle (Security integration, Services, Controllers)
- Commercetools Platform for E-Commerce Data, like products, orders, customers, carts, payment data
- Prismic for Content Management
- Wirecard for handling Payment Transactions (kind of sucks, but strong preference by customer, also partnership)
- Memcached for making stuff go fast
- Guzzle for making HTTP requests


Prismic.io
Customer
Symfony 2 App
Commercetools
Platform
SAP by Design
Backoffice
Memcached

Commercetools Platform in a Nutshell

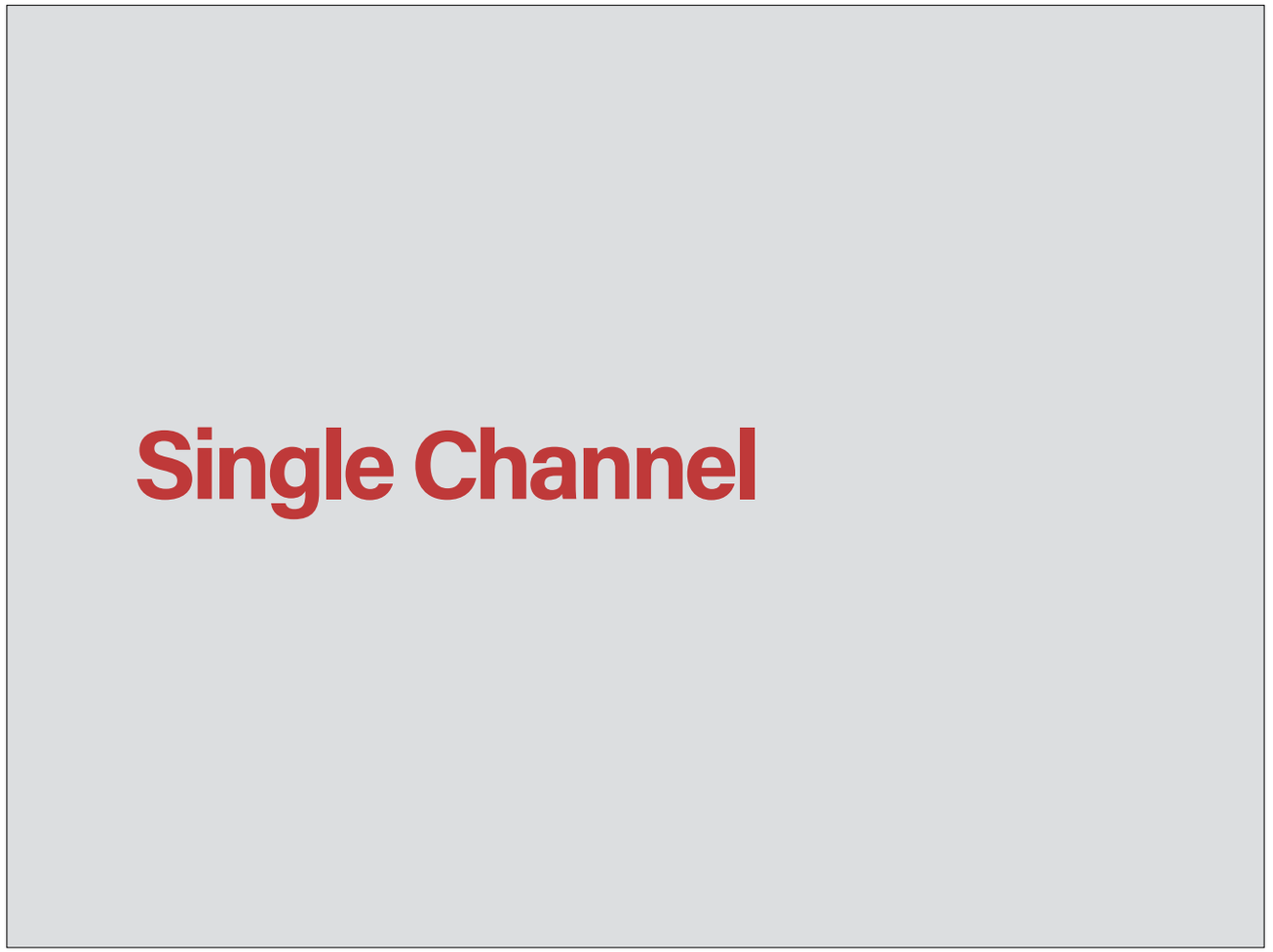

Web
Mobile
Consumers
& Companies
PoS Terminal & Instore
Wearables
commercetools
Product Information Management
Customer Data
Order Management System
Discounts & Promotions
Internationalization
Multichannel Management
Pricelistings
REST-API
ERP
WMS
CRM
Online
Marketing
More
Systems

Commercetools Platform

- Multi-Channel
- Key data structures
- REST API with flexible query language and search support
- Simple Backend UI
- Runs in the Cloud
- Monthly fee depending on number of API requests needed and a commission based on order amounts

- It defines representations of all key data structures, like products, customers, carts, orders, taxes, shipping methods, payment transactions
- You interface with these data structures by a REST API, which allows you to query data with a flexible query language
- A simple backend UI is provided, allowing you to create and manage products, view customers and orders, and manage shipping methods, taxes, and countries
- This runs in the „cloud”, there’s nothing to host on your own servers except your frontend
- You pay a monthly fee depending on the number of API requests needed and a commission based on order amounts

Multi-Channel?

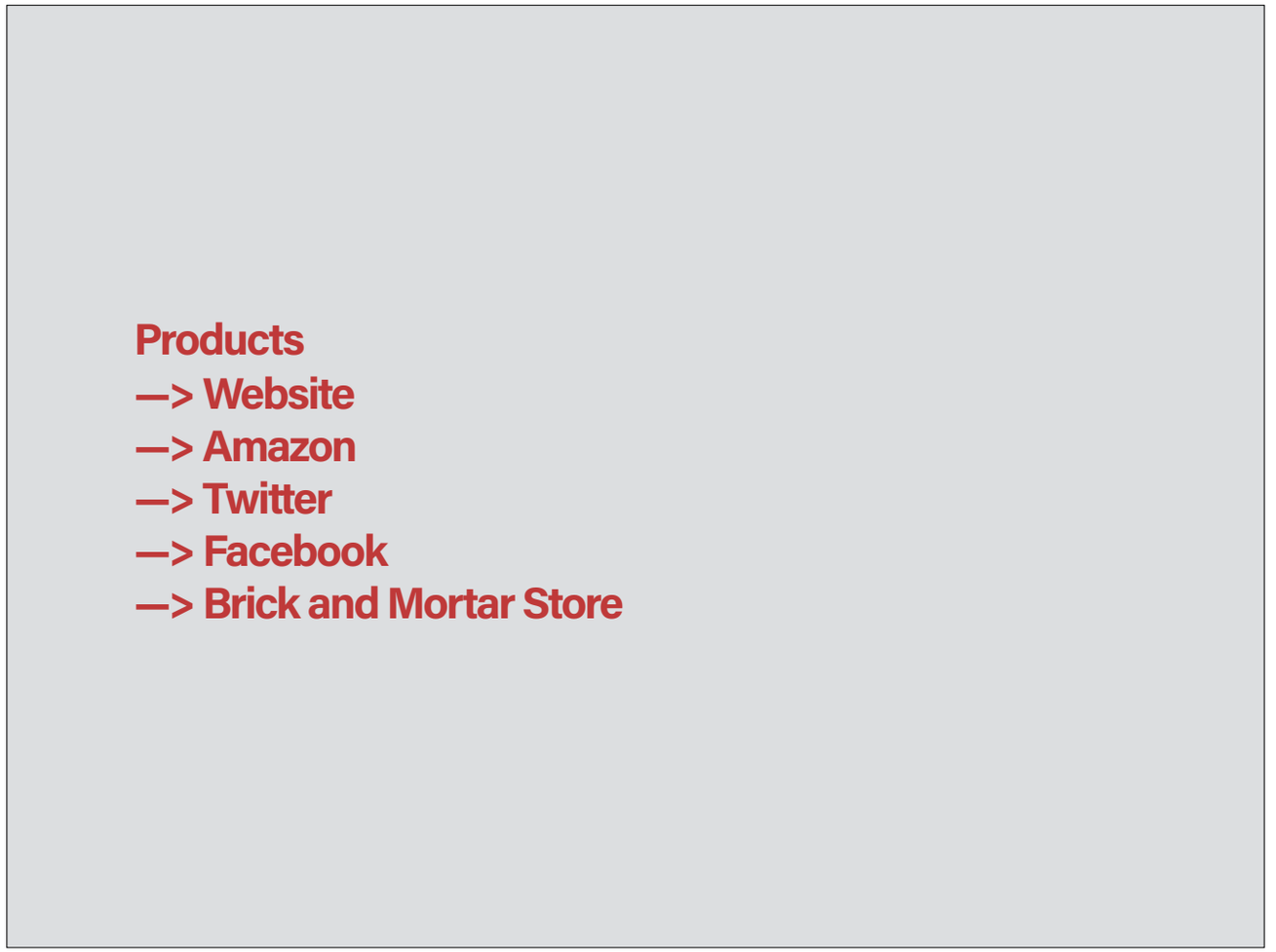


If you do a normal online shop or its a brick and mortar shop down the street, then it’s called Single Channel.

Most shop systems, like Magento or Shopware, are optimized to be deployed to sell on a single channel: The website made with them. They combine E-Commerce Business Logic with website building tools.

Products —> Website

Because in Single Channel you sell your products only on one place, for example your own website, built with something like Magento.



What „Multi-Channel” then means, is having a single product catalogue and selling the products not only on your website, but also in your own Smartphone App, on Amazon, with a Twitter Card, your Brick and Mortar Store, or even on Facebook Messenger.

Decoupled E-Commerce

A system which wants to handle this new way of selling stuff online in a great way has to do things differently.

For example, why have website building tools in the platform, if you only sell on Amazon? Why integrate with external systems with a plugin? Why not provide only a rich API instead and use whatever you want for the frontend and having a clean integrate external systems?

I call this approach Decoupled E-Commerce.

Decoupled E-Commerce

- Decoupling from the selling channel
- E-Commerce System is a Microservice providing an HTTP interface
- Defines how all structures look *and* business logic

In Decoupled E-Commerce the E-Commerce system is decoupled from the selling channel, i.e. it has only loose coupling with the website you sell your stuff on.

The way it's done is to treat the E-Commerce system as a Microservice. A Microservice which has an API accessed over e.g. HTTP.
First, this API defines how the data looks:

* How customer data looks like
* How product data looks like
* How carts look like
* How a line item looks like
* How an order looks like
* How a payment looks like
* How a product review looks like
* And much more

What's really neat about this, is that it makes interfacing with other systems so much easier.

For example, we had to integrate with SAP By Design (SAP ByD). With other E-Commerce Systems you have to integrate this directly with your chosen tool, like Magento.

This creates a strong coupling between the shop and the other component. For example, if you want to change the ERP tool, you have to change the shop to work with

What did we gain?

- Makes interfacing with other systems easy
- Loose coupling between channels and Webshop System
- Easily change parts, like building the website in the newest hipster framework, or switching the ERP
- Yay Orthogonality!
- Fronted can be anything as long as it can connect to the API

Decoupled Content Management

Decoupled Content Management

- Same principle
- Fits perfectly with Decoupled E-Commerce
- Microservice providing an HTTP interface to content
- Use the same content everywhere
- Output content wherever you need it

With Decoupled E-Commerce, Decoupled Content Management, or headless CMS, comes natural.

The problem with Content Management Systems in E-Commerce is, that Content Management and E-Commerce are both such large domains, that you either your CMS provides half assed E-Commerce, or your E-Commerce Platform provides half assed Content Management tools.

This is how it works:

1. Structure content independent of the representation
2. Define an API, in this case for retrieving content like blog posts
3. The API allows interaction from a variety of clients, like Smartphone Apps, or Shop systems which all can use the common content
4. Profit!

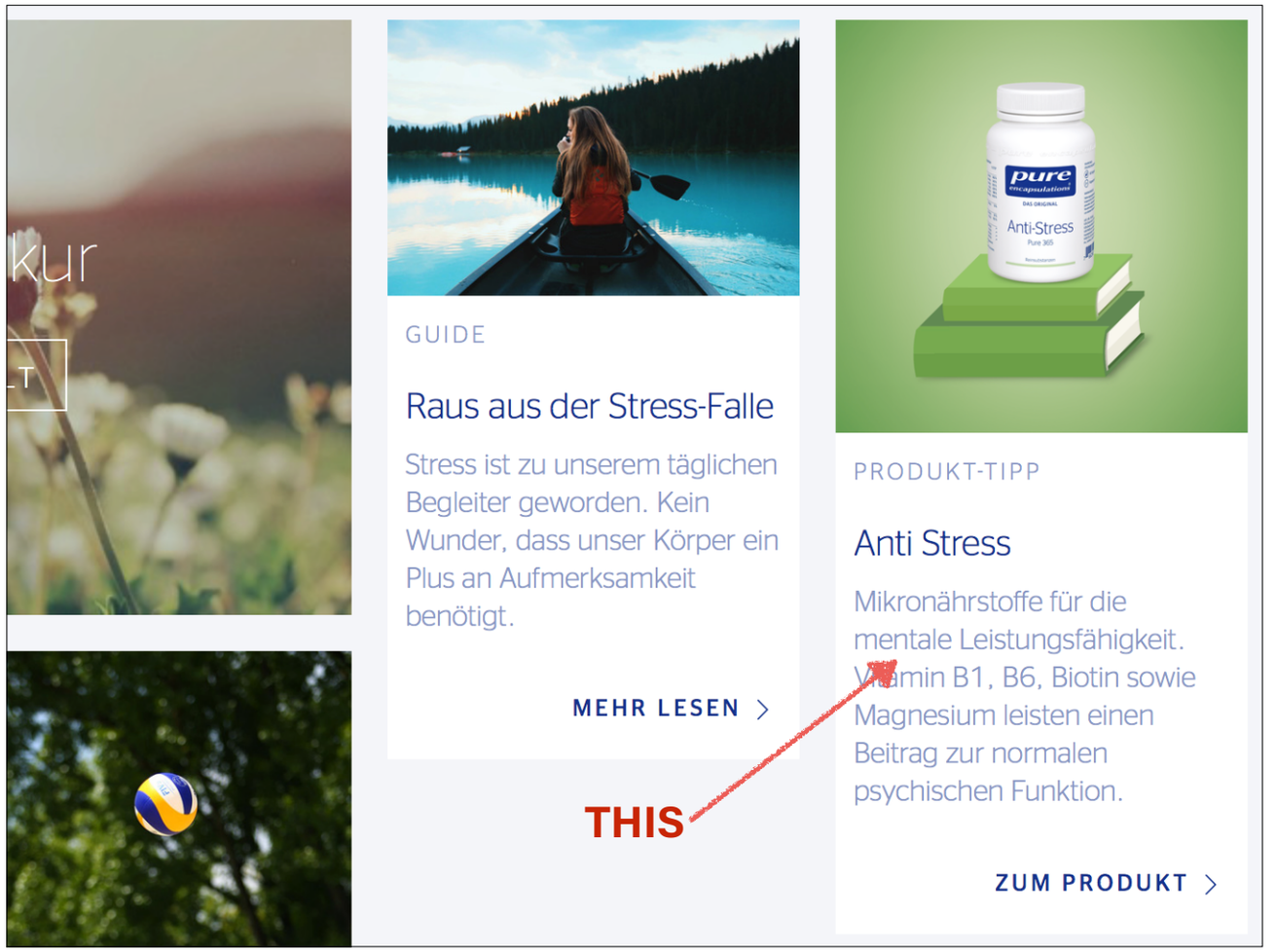
Prismic in a Nutshell

- Backend for defining document structure and editing documents
- Releases: Change documents, release all changes at once
- A/B Testing content
- API for querying and retrieving content
- Hosted in the cloud, monthly fee, starting free for Creative Commons licensed content

Prismic in a nutshell:

* It provides a backend to define your custom types with a variety of fields, like text, geo, rich text, and more
* Provides a great interface to create and edit your documents
* Allows you to combine content changes in more than one document into a release, which on publish deploys all changes at once
* It provides an API for querying and retrieving documents with a flexible query language
* All data is hosted in the cloud, you pay a monthly fee (7$/mo)

Content Commerce



Amazon is investing huge amounts of money in machine learning to surface the products to you which you are most likely to buy.

But if you do not have Amazon's money, you have to find something else. Here Content Commerce comes into play.

Content commerce means E-Commerce integrated into content, like blog posts. Think about a fashion blog which allows you to buy the outfit that's presented.

What we did here is allowed the content managers to integrate products into the blog streams. Because the frontend was custom built this was very easy to do. Just link Products by ID to a document in Prismic, then in the frontend resolve this ID and display a nice product preview.

Content Commerce

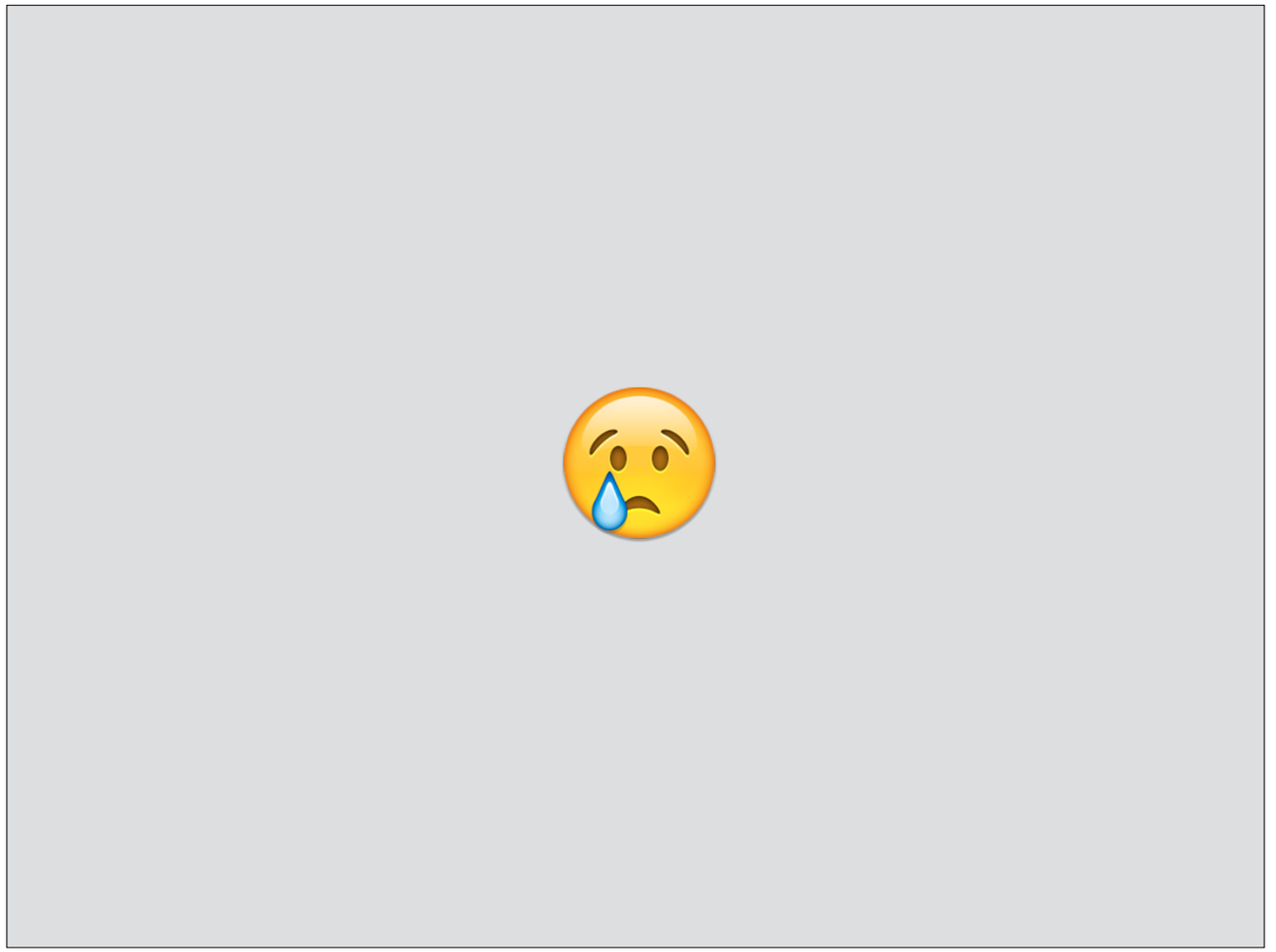
- Combine editorial content with e.g. product recommendations
- Big E-commerce portals, like Amazon, can’t do this
- Provide value to customer through information before buy
- Make it easy for customers to find a product they buy

Challenges

HTTP API Performance

100ms per Request

We noticed that requests to the APIs are generally around 100ms for each request, pretty regardless of request/response size, in a production environment.



This isn’t really fast. A typical MySQL query takes a couple of microseconds.

This is because of TCP and HTTP overhead and because the servers are not in the same datacenter and have to travel more hops over the network. Each hop adds latency. (HTTP/2 may fix some of this).

Avoid Cloud?

You can’t really do very much about this.

One way would be to avoid a cloud solution and look for solutions which you can host on your own infrastructure, like Woocommerce.

Cloud Benefits

- Scaling already handled, no caring about sharding, master-slave, replica sets, failover
- No hosting fees
- Security and Backups is handled by provider
- Easy integration with other systems over OAuth

This would mean that requests could happen within the same datacenter, cutting much of the latency, but you loose all the benefits of a cloud solution. Like not having to care about security updates, servers, or backups and easy integration with other cloud based services.

Batch requests and make them in parallel

So what else can you do?

You should absolutely care about how many requests you make within a page view.

Make as less as requests possible. Batch requests up and send them in parallel, as much as sanely possible. Most HTTP libraries for PHP let you do this (we used Guzzle).

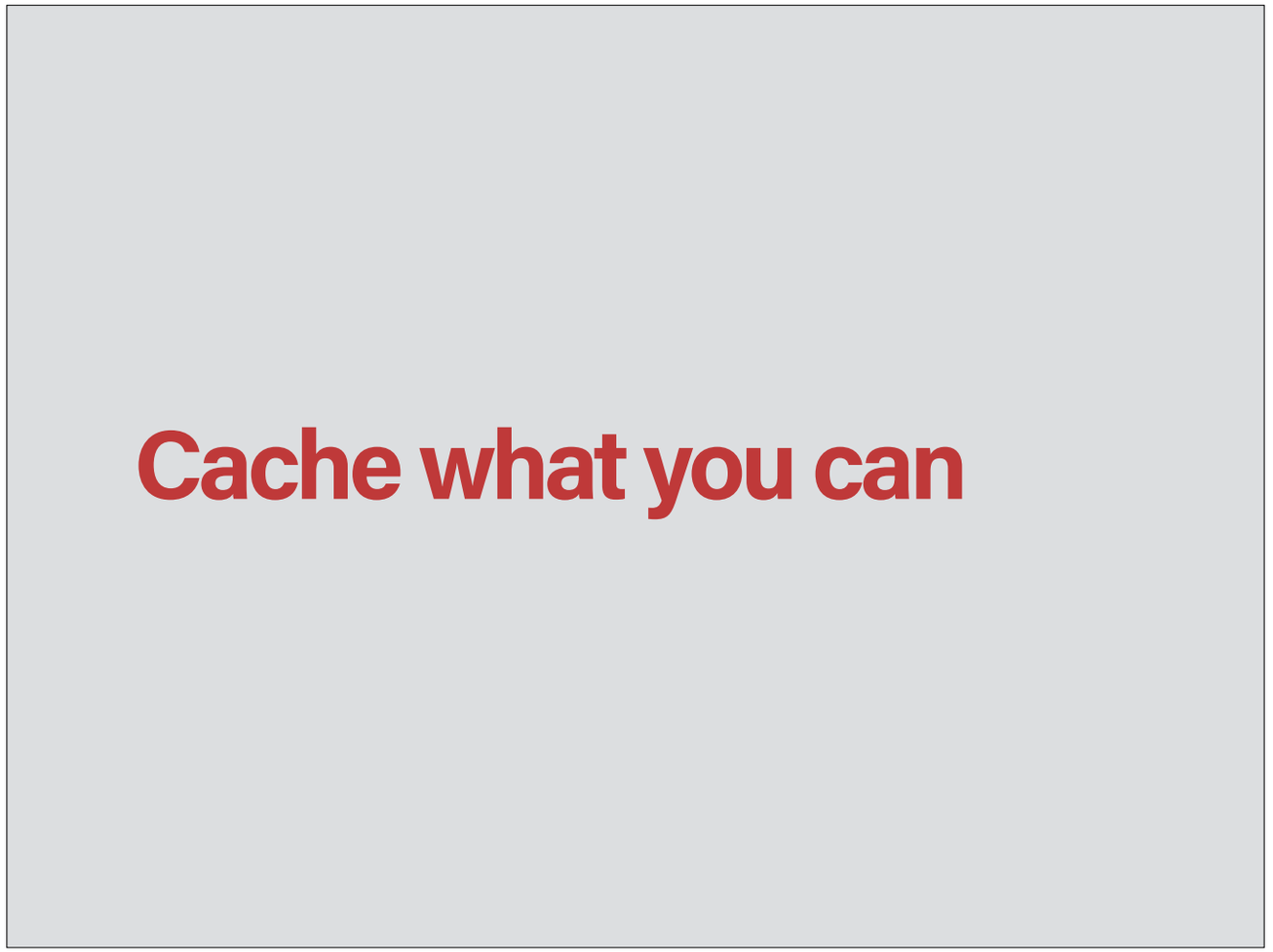
```php
use GuzzleHttp\Client;
use GuzzleHttp\Promise;

$client = new Client(['base_uri' => 'http://httpbin.org/']);

// Initiate each request but do not block
$promises = [
    'image' => $client->getAsync('/image'),
    'png'   => $client->getAsync('/image/png'),
    'jpeg'  => $client->getAsync('/image/jpeg'),
    'webp'  => $client->getAsync('/image/webp')
];

// Wait on all of the requests to complete.
$results = Promise\unwrap($promises);

// You can access each result using the key provided to the unwrap
// function.
echo $results['image']->getHeader('Content-Length');
echo $results['png']->getHeader('Content-Length');
```



The other thing is caching. Cache all you can, if possible to eternity.

Russian Doll Caching

Russian Doll Caching

- Cache in Templates ({% cache %} by @asm89)
- Cache template fragments containing dynamic data, ideally to eternity
- Cache Key contains digest of fragment content and last update time of data

Russian Doll Caching

- Automatic invalidation when content changes
- No Proxy, no ESI necessary
- Less Infrastructure
- Invalidation sometimes hard

Expand links to other resources on the server wherever possible

Also take advantage of any expansion of linked resources that is possible on the server. You should get as much data as possible in a single request.

Also consider the N+1 problem, which is much worse with this much latency involved. Never create GET requests for linked resources in a loop, combine them into a single request when possible.

Import/Export

Import/Export

- No “mysqldump” possible
- Look for availability of good import and export tools when choosing the platform
- Commercetools Platform has pretty good ones
- Prismic has literally no export, and not even a write API

This has to be supported by the system. You can’t just do mysqldump.

Commercetools Platform has robust tools for this and a very rich API for custom imports/exports. But you have to plan for creating custom import and export tools when needed. On the upside, through the API these things are very easily made and do exactly what you want, and don’t have to deal with raw database tables.

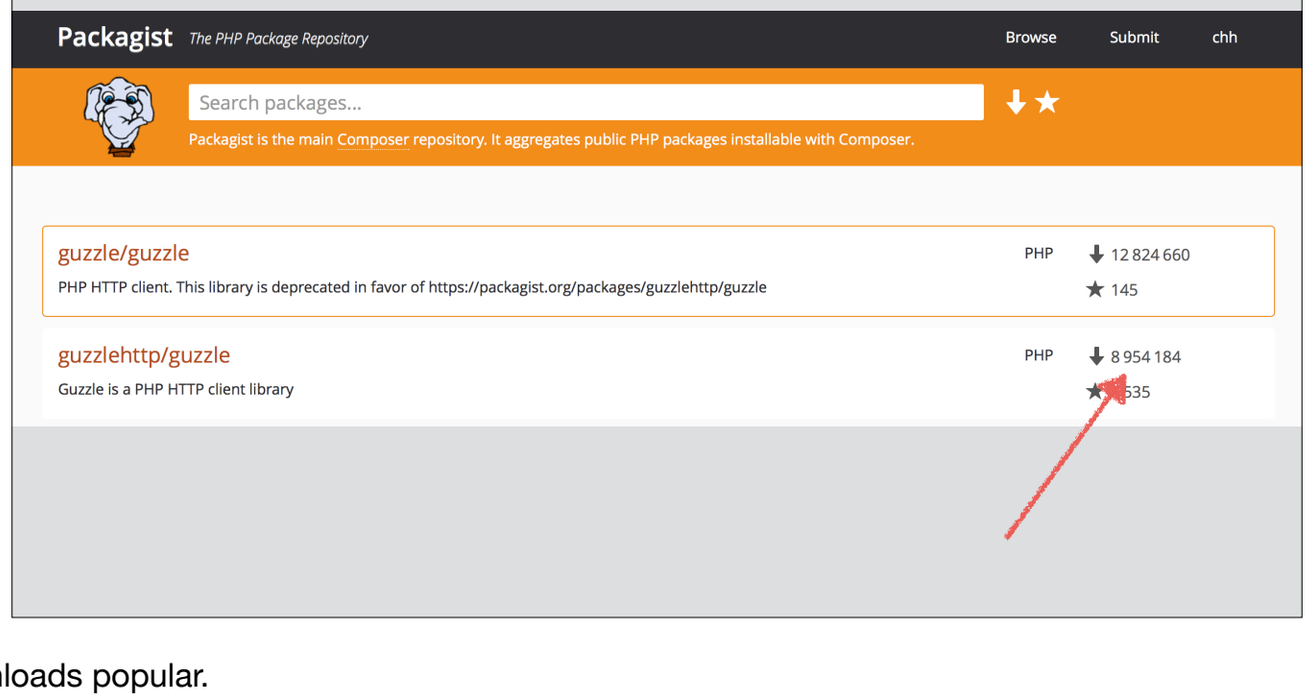
Prismic though, hasn’t any API for creating documents as of now. This means you can export your documents by using the API, but there isn’t any automated way to import data from other systems.

SDK Conflicts

Guzzle is awesome

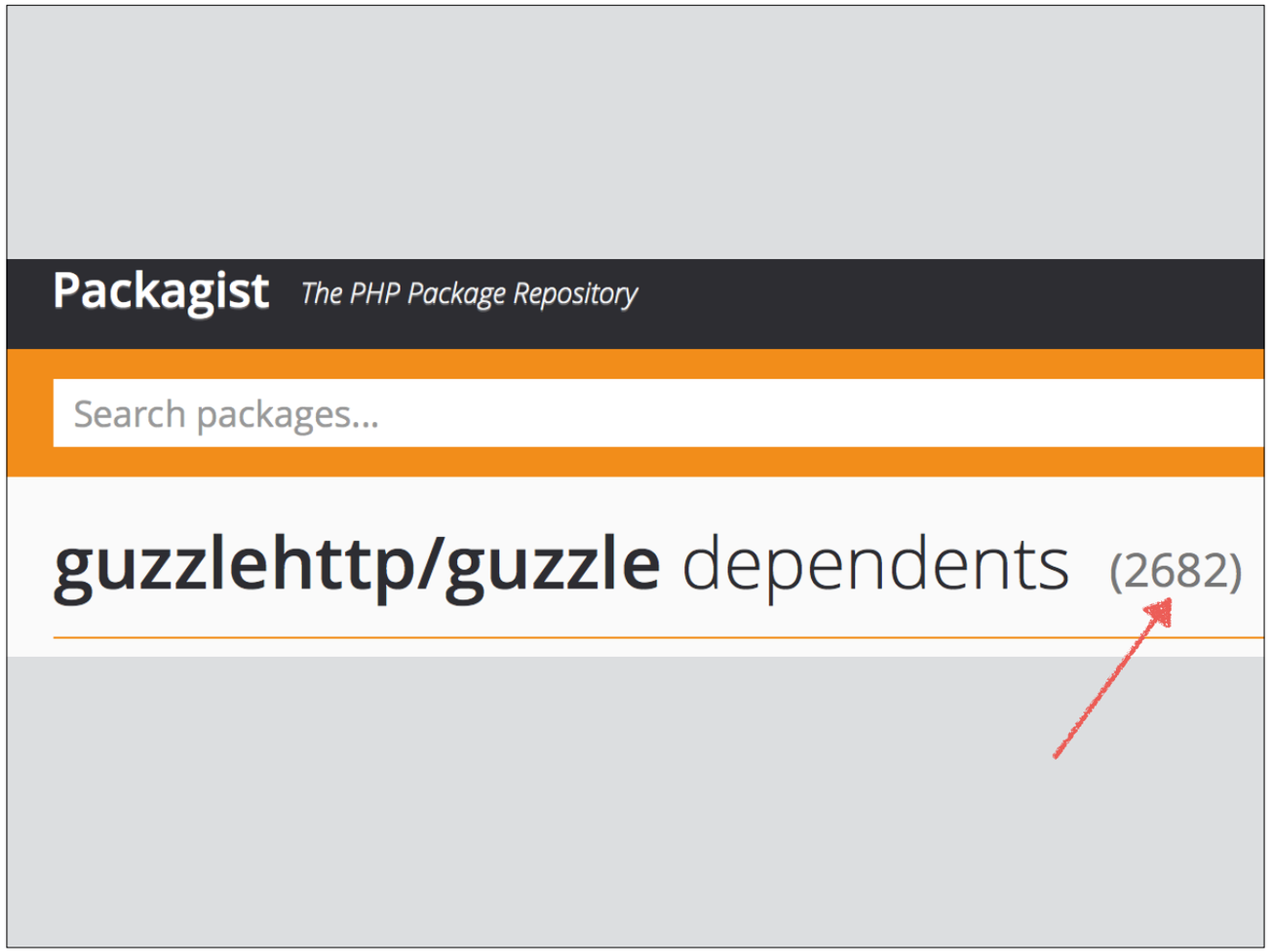
Guzzle is awesome. It has an awesome API, supports async requests and is fast.

Therefor it’s popular



Also it’s popular. Very popular. 8m downloads popular.

Many SDKs are using it as a base



2682 libraries are depending on Guzzle. At least. The Amazon Web Services SDK uses it.

What if two SDKs require Guzzle, but in conflicting versions?

But what if two SDK libraries use Guzzle? What if these two SDKs use different versions of Guzzle?

What happened

- There was no Commercetools SDK for PHP (there exists one now)
- So we built one on top of Guzzle 4.x and Guzzle Services
- Later we integrated Prismic
- The Prismic SDK for PHP requires Guzzle 5.x

Fail

Composer error. Understandably you can’t install two versions of a library with the same name.

You can't really do anything but make *your code* work with Guzzle 5.x

A brief call to action to SDK authors and application developers

So I want to make a brief call to action to SDK authors and application developers.

Take a look at PSR-7

Take a long hard look at PSR-7. Trust me it’s for the better.

PSR-7

- Defines interfaces for HTTP Requests, Responses, URLs, Content Streams and more
- PHP Standard Recommendation supported by Guzzle, Symfony, Zend Framework and more

There’s a standard developed in the PHP community called PSR-7, which defines how HTTP Requests and Responses should look like. It’s supported by Symfony, Zend Framework, Laravel, Guzzle and many more.

Write everything against PSR-7

- Use only PSR-7 RequestInterface for doing requests
- Every SDK request should be a PSR-7 request
- Allows complete decoupling from HTTP clients
- Easy bridging to whatever HTTP client you want to use
- Ideally avoid depending on an HTTP client in your SDK

Really, what's a HTTP Client anyway?

```php
<?php

use Psr\Http\Message\RequestInterface;
use Psr\Http\Message\ResponseInterface;

function httpClient(RequestInterface): ResponseInterface {
    // whatever
}
```

Because when we boil it down, what's a HTTP Client really?

It's a function which turns a request into a response. Nothing more. A function that's easily exchangeable.

What this means is that as long as the SDK depends only on PSR-7 interfaces you can use whatever HTTP library for PHP that you like and we wouldn't have had any trouble.

So…

- Try to use only PSR-7 interfaces when making HTTP requests
- Bridge PSR-7 objects to the HTTP library by writing an adapter
- Or, use a library which directly supports PSR-7, like Guzzle v6.x
- This keeps you independent of any HTTP client

The Future I want to see

- SDKs which only ship with means to create PSR-7 compatible request objects
- Without any dependency on a HTTP client
- You can easily adapt the SDK to use any library you want
- No two SDKs which have conflicting HTTP client versions

github.com/sphereio/commercetools-php-sdk

The authors of the Commercetools SDK have already gone in this direction. Take a look at it, it’s very interesting. This is definitely the way SDKs should be written in 2016.

Domain Models

No database = No ORM

Because you’re dealing with REST APIs and not with relational database systems, you can’t really rely on an ORM like Doctrine.

ORM often defines design of domain models

Not sure if mapping response objects to models is worth it

I’m not sure if mapping every model an SDK returns to an application model is worth it. Usually it’s much work without any benefit.

Repositories,
Repositories everywhere

Repositories

- Made for dealing with data sources other than databases
- If you dealt with Doctrine before this will be familiar to you

I prefer Repository classes. They are a quite good abstraction for this kind of thing, because they are made for dealing with data from a variety of sources.

Plus you end up with similar interfaces, like you would have had with Doctrine, because it also uses Repository classes.

Conclusion

Decoupled E-Commerce is still evolving, but it will be huge

Everything is still evolving. The Commercetools Platform is kind of in Beta, and new API endpoints are introduced every couple months.

On the one hand this meant they listened to us. We gave feedback which was directly incorporated into the product, which was awesome.

On the other hand this meant that some things took time to become available, like support for discounts. Some things are currently not existing at all, like Webhooks (which are coming in 2016).

There's also huge potential for integrations with third party tools.

Imagine an invoicing tool which directly hooks into this. Which uses orders from this system, without any need to import or export. It magically has all the information about the orders.

Imagine building a reporting tool once for a client, then easily reuse it for other clients on the platform (or selling it).

Imagine running your whole operation on the platform. Then having a cash register in a store which uses that data. And a website which allows the customer to order online. Feeding this data to Amazon and orders back into your system again. Creating an app which uses this same data. Cleanly interface with any inventory management solution without any hacks.

Imagine writing tools for the platform and comfortably selling these tools to all platform customers with a marketplace.

Everything is more loosely coupled.
This is good.

Everything is exchangeable. Display content in whatever frontend technology you like. Mix and match technologies. Easily switch to another ERP system.

Performance is hard, though

You need to care a lot about making as less requests as possible, expand links to other resources wherever possible on the API server and cache a lot.

Also API providers have to step up their caching game a lot.

You are not reinventing the wheel, but you still need code for a lot of stuff

With Commercetools handling most of the actual business logic, there's still about 15000 lines of code in the application. This isn't exactly small.

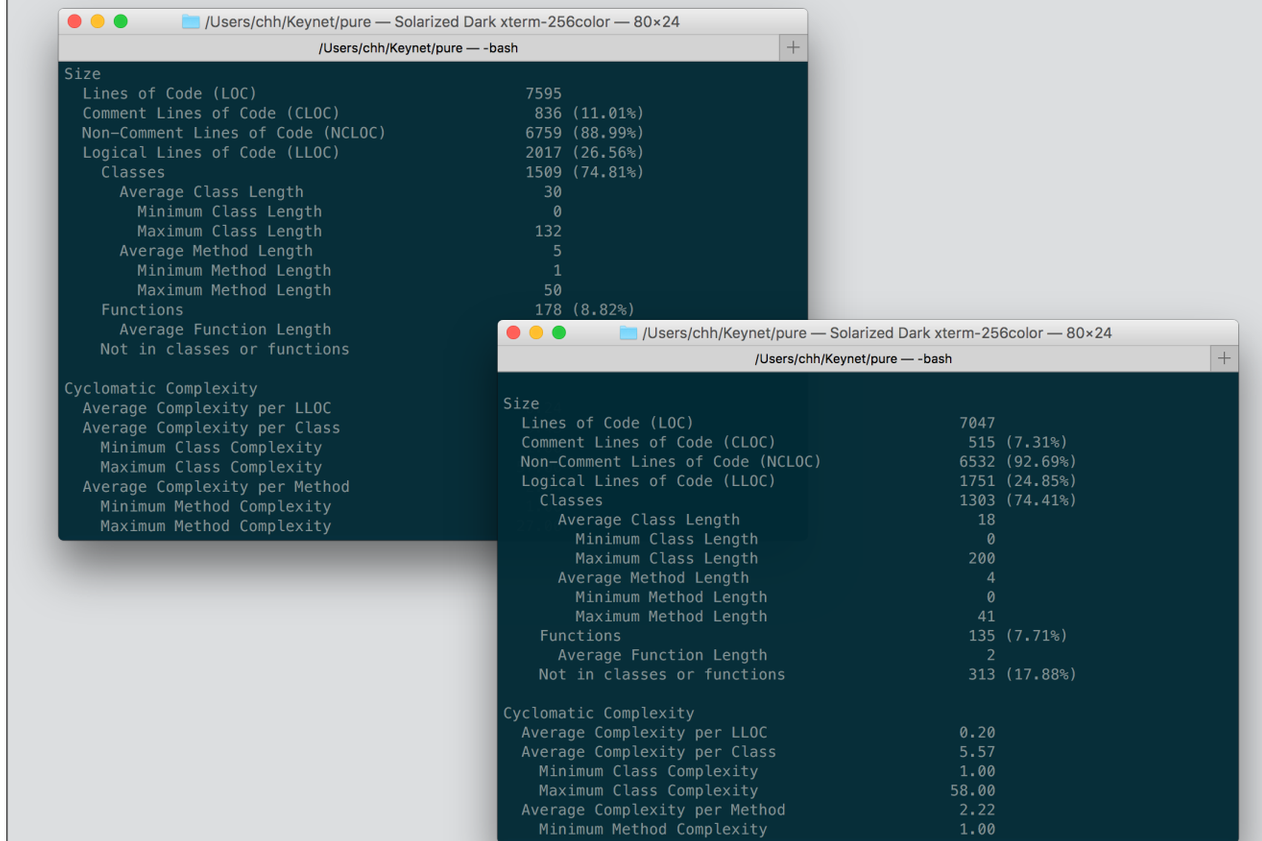
About half (7000 lines) is in a Symfony bundle which we can drop into any project we want.

But also don't go overboard with reusability. Caring too much about reusability leads to more code than absolutely necessary. Code which is most often specific to the project and has to be adapted to a clients needs.

The approach I've settled on is building small services which do common things, like adding a product to the cart, or initiating the payment process for a given cart. Then write the controllers using these services.

In hindsight it didn't make sense to try to reuse controllers. Because the controller defines UI flow, which varies a lot between clients when you do custom E-Commerce experiences.

Also, you do have a lot of code. But all of it is written by you and your teammates, which means you are familiar with it. This makes bugs easier to fix and you are less dependent on the framework, which are good things.


/Users/chh/Keynet/pure — Solarized Dark xterm-256color — 80×24
/Users/chh/Keynet/pure — -bash
Size
Lines of Code (LOC) 7595
Comment Lines of Code (CLOC) 836 (11.01%)
Non-Comment Lines of Code (NCLOC) 6759 (88.99%)
Logical Lines of Code (LLOC) 2017 (26.56%)
Classes 1509 (74.81%)
Average Class Length 30
Minimum Class Length 0
Maximum Class Length 132
Average Method Length 5
Minimum Method Length 1
Maximum Method Length 50
Functions 178 (8.82%)
Average Function Length
Not in classes or functions
Cyclomatic Complexity
Average Complexity per LLOC
Average Complexity per Class
Minimum Class Complexity
Maximum Class Complexity
Average Complexity per Method
Minimum Method Complexity
Maximum Method Complexity
/Users/chh/Keynet/pure — Solarized Dark xterm-256color — 80×24
/Users/chh/Keynet/pure — -bash
Size
Lines of Code (LOC) 7047
Comment Lines of Code (CLOC) 515 (7.31%)
Non-Comment Lines of Code (NCLOC) 6532 (92.69%)
Logical Lines of Code (LLOC) 1751 (24.85%)
Classes 1303 (74.41%)
Average Class Length 18
Minimum Class Length 0
Maximum Class Length 200
Average Method Length 4
Minimum Method Length 0
Maximum Method Length 41
Functions 135 (7.71%)
Average Function Length 2
Not in classes or functions 313 (17.88%)
Cyclomatic Complexity
Average Complexity per LLOC 0.20
Average Complexity per Class 5.57
Minimum Class Complexity 1.00
Maximum Class Complexity 58.00
Average Complexity per Method 2.22
Minimum Method Complexity 1.00

Search is builtin and always available

It’s great that both Prismic and Commercetools Platform have great search features built in. In most cases you don’t need to use Elastic Search (or something like Findologic).

No worries about Scalability

Scalability is one of the biggest upsides. Because your frontend app is only a client to the APIs you just don’t have to care about stuff like sharding, scaling databases, failovers, storage expansion and the like.

This means you just have to care about scaling the caching layer, which is not very hard, and scaling the web servers.

You may need a database after all

Because in the end, we couldn’t completely avoid using a database.

Because we had to implement a store finder, and like said, Prismic has location based search, but no import, we had to use MongoDB for this feature.

So should you use a decoupled E-Commerce approach on your next project?

It depends. But mostly no.

As awesome as this ap
proach is and what upsides it has in terms of flexibility, it’s still a lot of work to make an online shop.

For most cases it’s probably easier and makes more sense to create your shop with Shopify, WooCommerce, or Shopware.

But with these systems you get a predefined experiences which you may not be able to break out of.

If you have a custom experience to build decoupled E-Commerce is a viable option.

Thank you

Links

- My company homepage hochstrasser.io
- Keynet
- Commercetools Platform
- Commercetools PHP SDK
- PSR-7
- Prismic
- purecaps.net