Thomas Bley

From a monolith to a microservice shop architecture

Backend

Bringmeister.de Partner von EDEKA

code.talks

About me

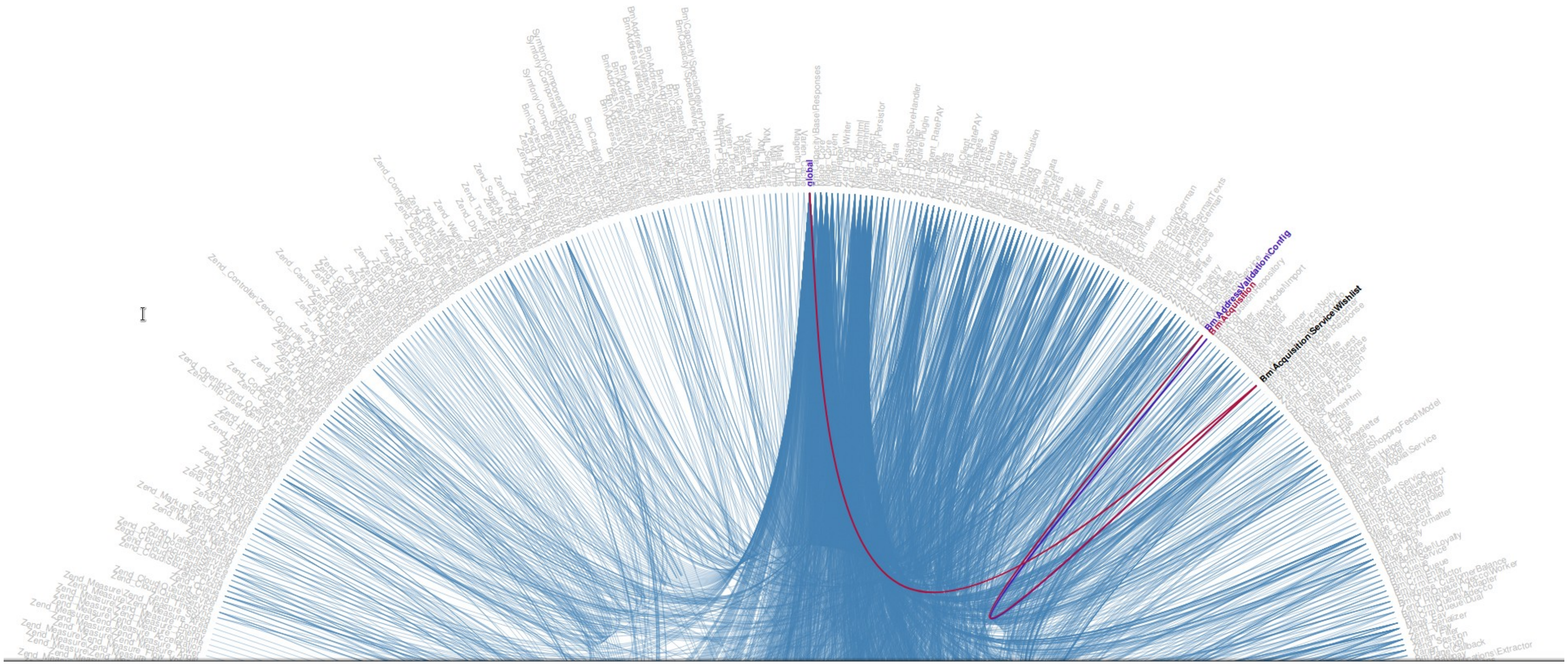
- Senior PHP Developer
- Linux, PHP, MySQL since 2001
- studied at TU München
- working for Bringmeister in Berlin

The Monolith

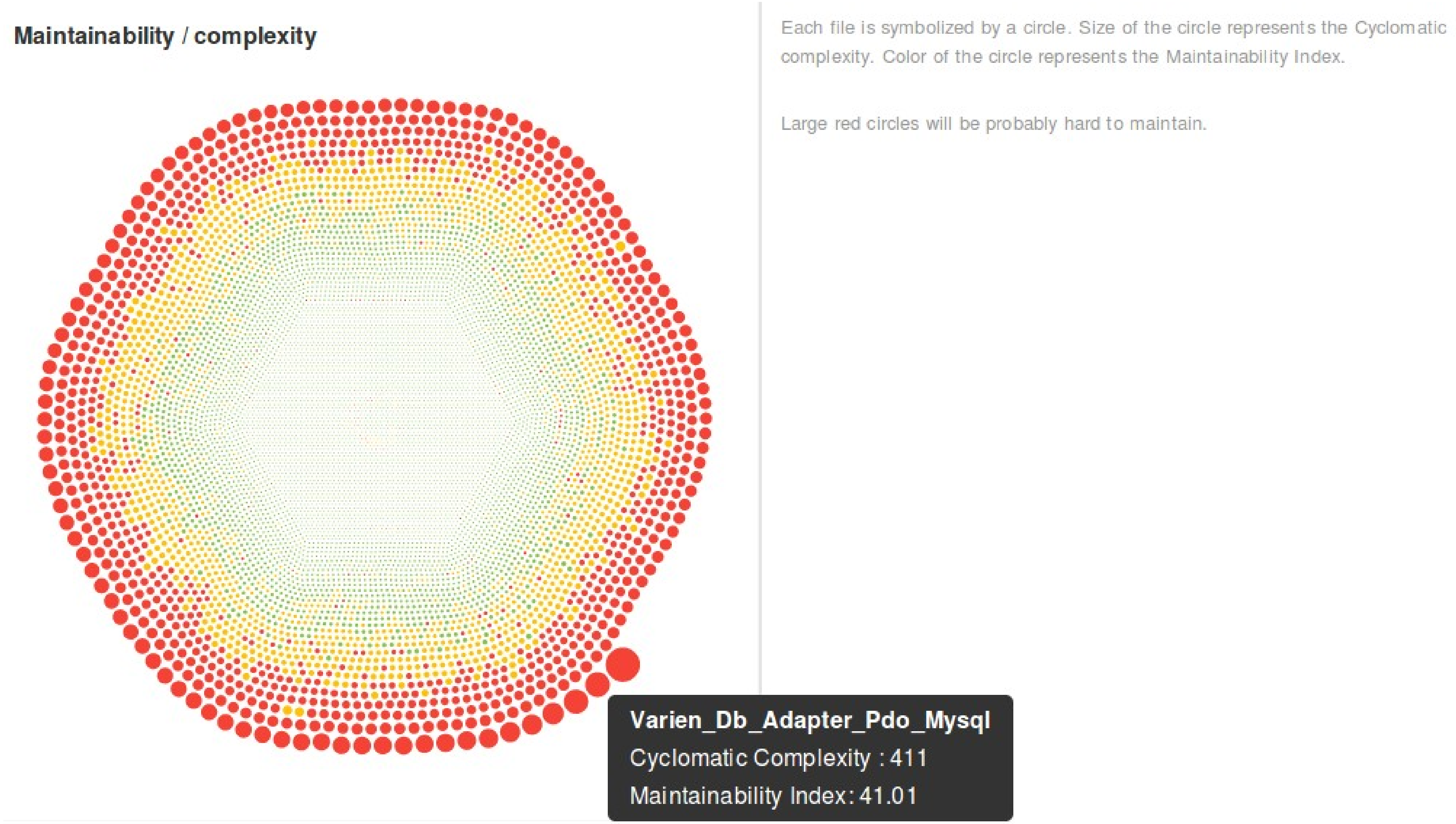
- Magento 1 Enterprise: support was discontinued by Adobe, license costs
- Large codebase with 1.8 mloc: high complexity, strong coupling between modules, unused code, unknown code, bad code, security issues, not designed for GDPR, bugs, developed over 10 years by 130 developers, hard to maintain
- Many frameworks: Magento Varien, Zend Framework 1, Symfony 2, ReactPHP, node.js, no composer, multiple API layers, etc.
- Performance issues: add product to cart 400 queries, order placement 3200 queries using 5 transactions, deadlocks, slow queries, memory limit issues, Admin login >60s, single order search in Admin 15s, database designed by Entity–attribute–value model, aggregation tables filled synchronously, difficult to scale the business
- Tests: coverage <10%, require 7 GB database dump, many tests broken, only run locally, CI only used for deployment, mostly manual testing
- Productivity: slow development of new features, analyzing production issues very complex, updates only applied manually
- Multiple sources for product data: Database, Solr, Algolia => mostly not in sync
- Small team: 5 developers

Developers not happy, Management not happy, Customers not happy with the shop

How does the monolith look like?



How complex is it?



Maintainability / complexity

Each file is symbolized by a circle. Size of the circle represents the Cyclomatic complexity. Color of the circle represents the Maintainability Index.

Large red circles will be probably hard to maintain.

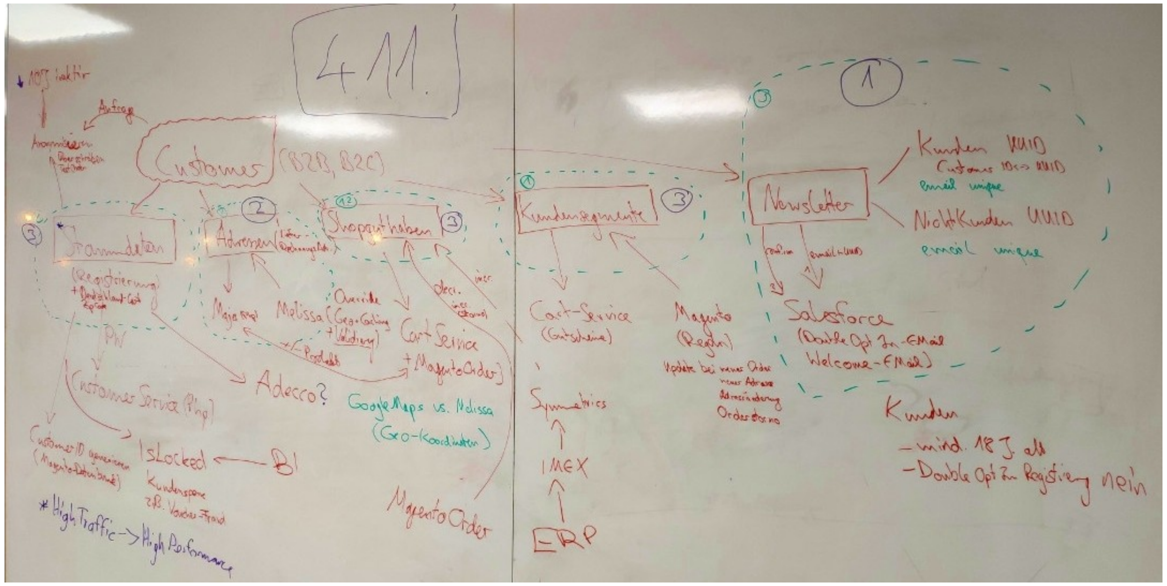
Varien_Db_Adapter_Pdo_Mysql
Cyclomatic Complexity : 411
Maintainability Index: 41.01

CI/CD, Operations

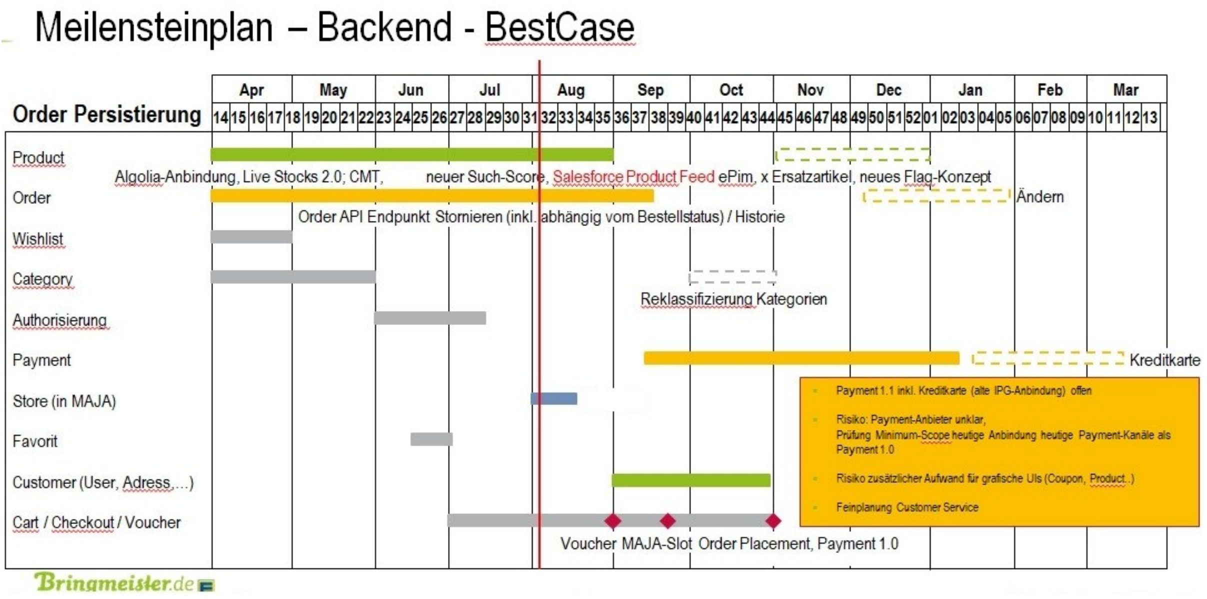
Why microservices?

- Smaller code base to work on
  - easier to develop, easier to change, easier to maintain
  - less complexity, focus on problem solving
- More options for databases and programming languages
- Easier to split work on multiple teams
- Hard system boundary with standard communication between services
- Limit impact of bugs and failures
- Isolation of data
  - less complexity in data storage
  - more complexity to join data

Explaining the project to developers



Explaining the project to management



- Payment 1.1 inkl. Kreditkarte (alte IPG-Anbindung) offen
- Risiko: Payment-Anbieter unklar,
  Prüfung Minimum-Scope heutige Anbindung heutige Payment-Kanäle als Payment 1.0
- Risiko zusätzlicher Aufwand für grafische UIs (Coupon, Product..)
- Feinplanung Customer Service

Define the future architecture

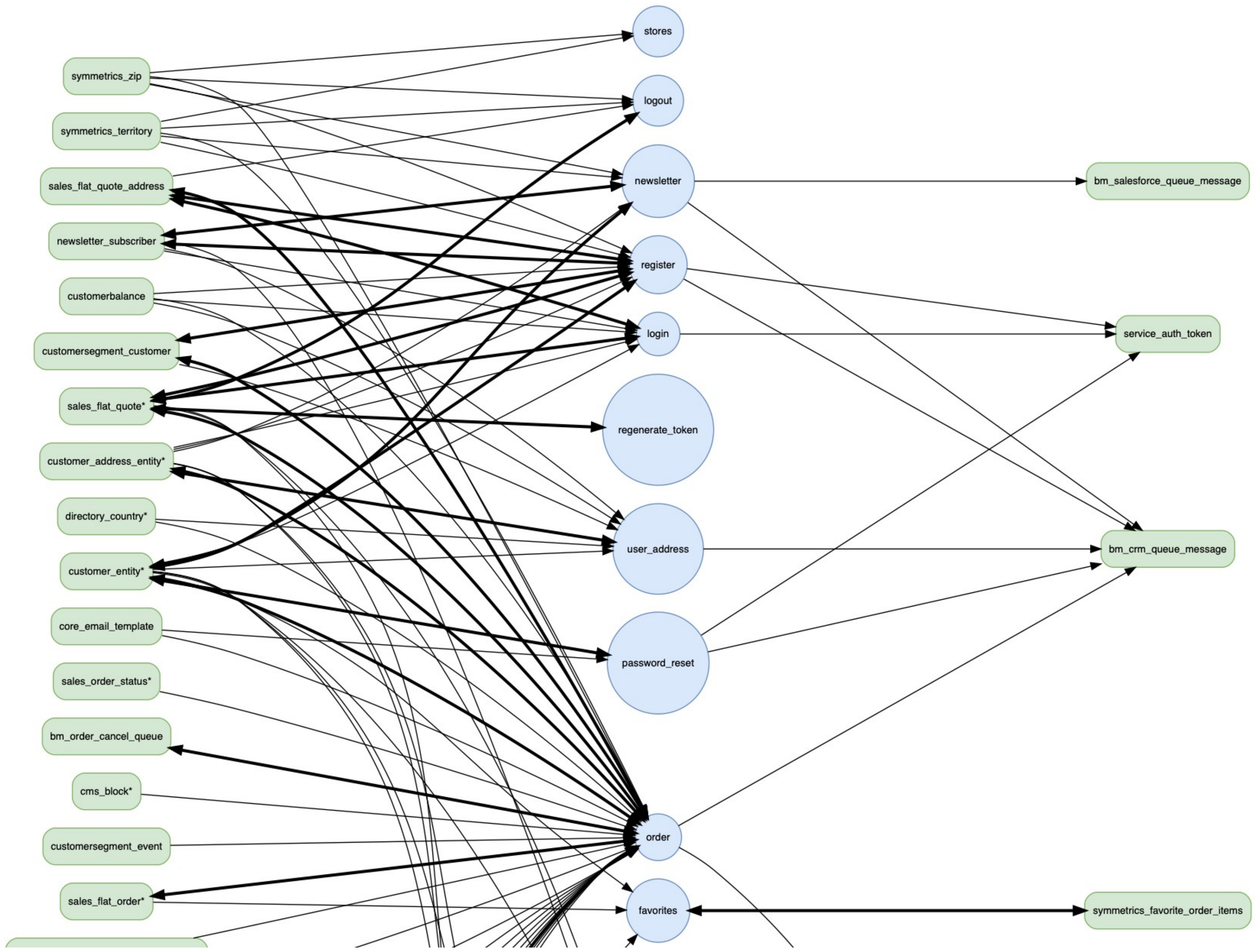
- We decided to stay with PHP and MySQL

  most experience, all required libraries and SDKs available, easier to port from PHP to PHP
- Keep existing server infrastructure
- No fullstack framework
  ➔ use our own mini-framework (200 loc)
- Write queries and schema definitions directly in SQL, no ORM, reduce joins by using JSON columns, no foreign keys
- Use JWT instead of sessions
- Single Monorepo for all services, each service with own code base, own database, own composer.json, etc.
- All product data in Algolia
- SOLID, Kiss

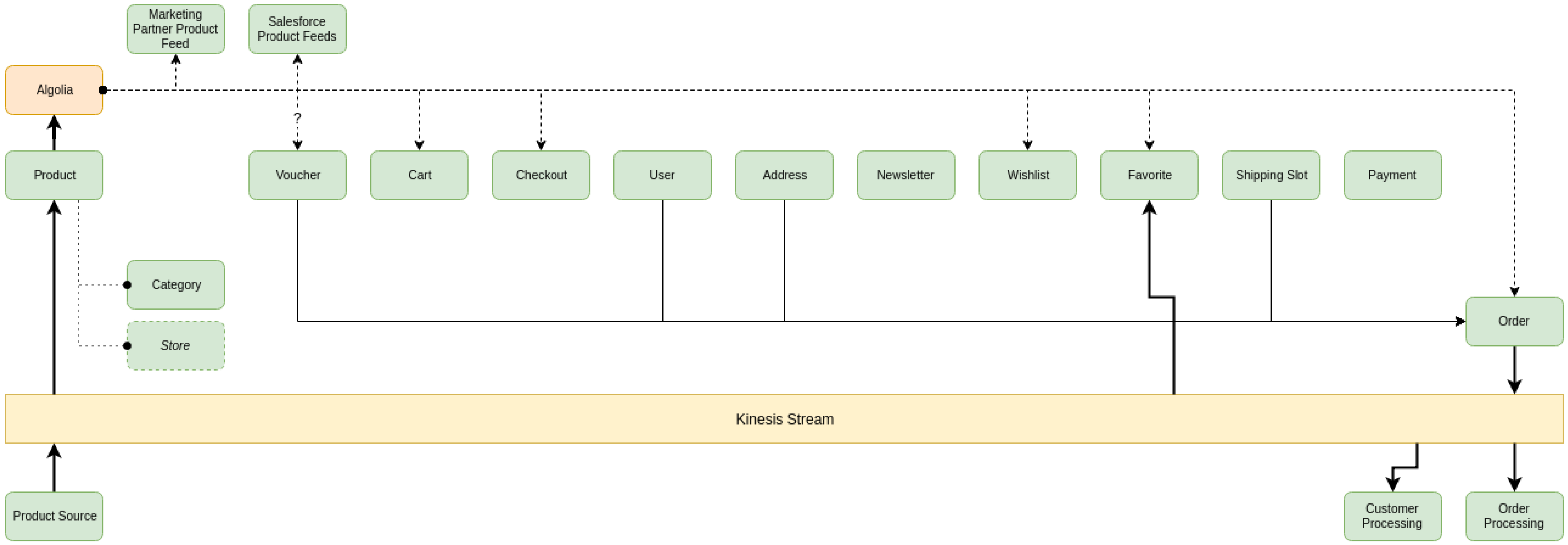
source: https://www.youtube.com/watch?v=fCt2_AsCWKI

- 100% test coverage, keep Behat tests
- Static code analysis with Psalm
- Enforce coding styles with PHP-CS-Fixer
- new CI/CD with Bitbucket Pipelines

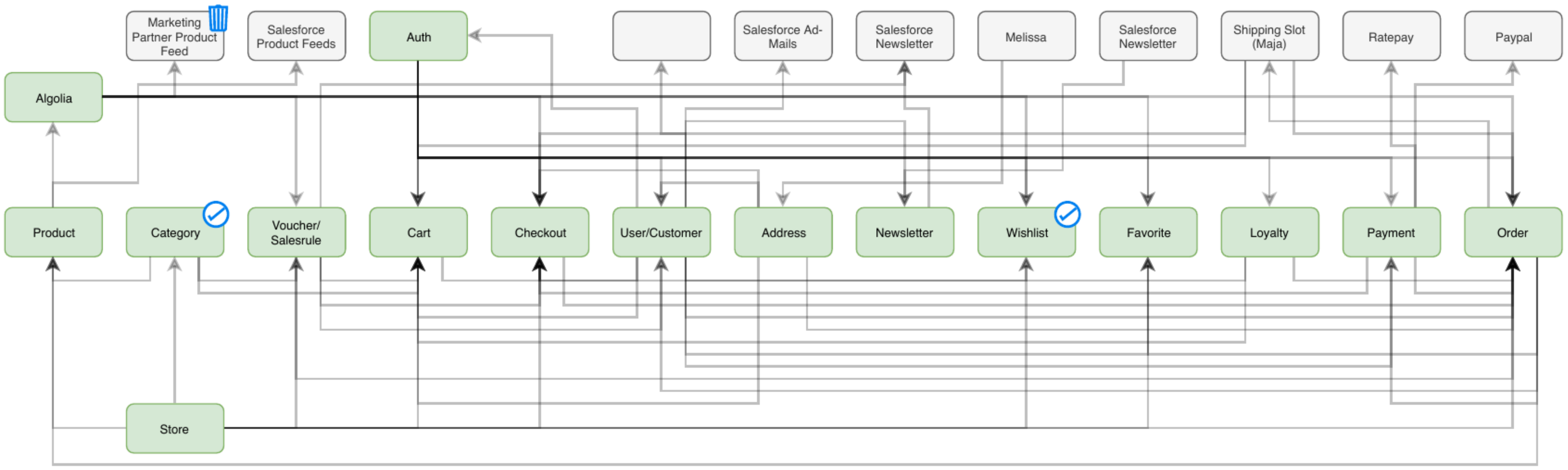
Analyze dependencies on database level



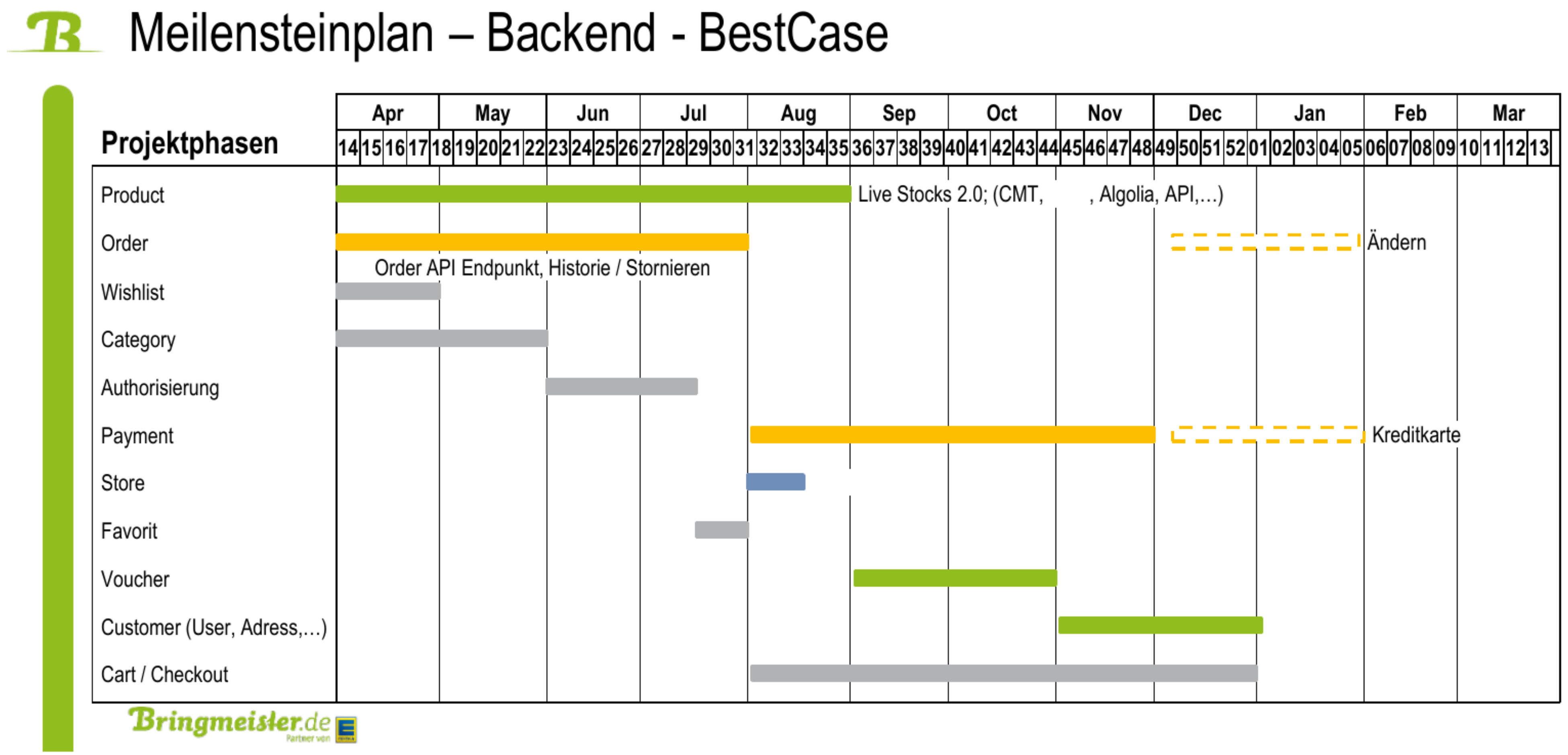
Identify services



Start with easy services

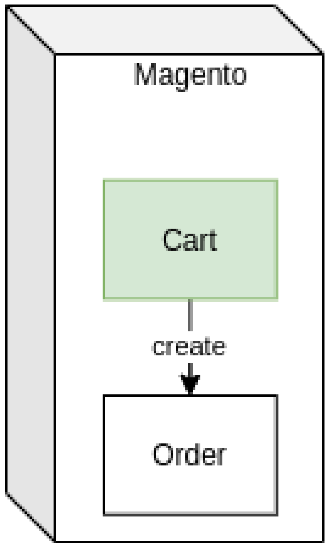


Implement and launch one by one

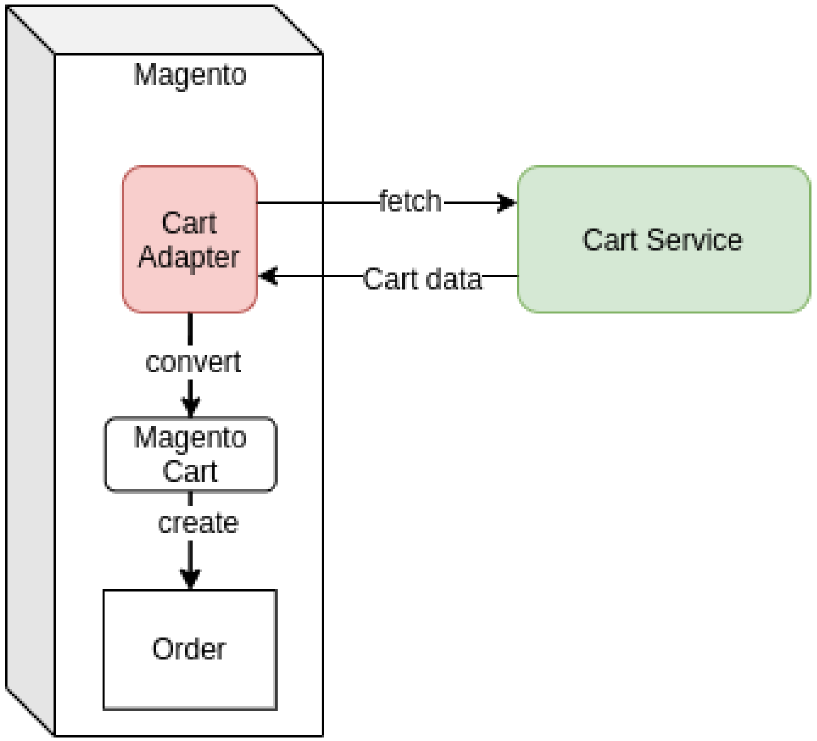


Build adapters to cut out components with strong dependencies

Before:



After:



Communication between Microservices and external providers

- Synchronous using REST, load balancer (strong consistency)
  - Customer master data, addresses
  - Cart, Vouchers
  - Product data
  - Payment providers
- Synchronous using SOAP (strong consistency)
  - Legacy systems (tour planning)
- Asynchronous using Events (Kinesis)
  - Orders, Logging
- Asynchronous using REST and queues
  - external providers (CRM, Customer Support systems, etc.)
- Forward customer’s JWT token between services

Results

- Project finished in time and in quality (Mar - Nov 2019)
- 10 microservices, 3 admin interfaces
- 99.99% test coverage with unit and integration tests
- Code size reduced to 100 kloc (coming from 1.8 mloc)
- Data size in database reduced by 80%
- System performance and revenue significantly increased
- Hardware costs reduced by 50%
- External security audit passed
- Tests, Build and Deployment in < 10 minutes
- Development of new features and maintenance much quicker and easier

Developers happy, Management happy, Customers happy with the shop

Thanks for listening!

Questions?

download slides:
TODO