**Зміст**

[Вступ 2](#_Toc70277364)

[1 Загальна характеристика підприємства практики 3](#_Toc70277365)

[1.1 Мета та рік створення 3](#_Toc70277366)

[1.2 Форма власності 3](#_Toc70277367)

[1.3 Основні види продукції, послуг чи видів діяльності 3](#_Toc70277368)

[1.4 Характеристика підрозділів підприємства 4](#_Toc70277369)

[1.5 Конкурентні переваги підприємства 5](#_Toc70277370)

[2 Посадові обов’язки IT-фахівців 6](#_Toc70277371)

[2.1 Технік програміст 6](#_Toc70277372)

[2.2 Веб-дизайнер 6](#_Toc70277373)

[2.3 Front-end розробник 6](#_Toc70277374)

[2.4 Back-end розробник 7](#_Toc70277375)

[2.5 Team lead 7](#_Toc70277376)

[2.6 Менеджер проекту (PM) 8](#_Toc70277377)

[2.7 Бізнес Аналітик (BA) 8](#_Toc70277378)

[2.8 Інженер із забезпечення якості (QA інженер) 9](#_Toc70277379)

[3 Звіт з індивідуального завдання 10](#_Toc70277380)

[3.1 Технічне завдання 10](#_Toc70277381)

[3.2 Проектування 10](#_Toc70277382)

[3.3 Програмна реалізація 18](#_Toc70277383)

[Висновки 53](#_Toc70277384)

[Список використаних джерел 54](#_Toc70277385)

Вступ

Одним із головних завдань навчального закладу є підготовка молодих спеціалістів до майбутньої роботи. Переддипломна практика дозволяє студенту навчитися пристосовуватися до праці на робочому місці, дає початкові уявлення про структуру, матеріальну й технічну базу, а також організацію процесів підприємства.

Звичайно, окрім впровадження в технологічний процес підприємства, одним із головних завдань даної практики є практичне використання набутих знань, що виходять із змісту обраної спеціальності. Це проявляється у самому процесі діяльності на підприємстві, так і при виконанні індивідуального завдання.

Переддипломна практика розпочинається з ознайомлення студентів з її метою, планом проходження, інструктажем з техніки безпеки, і згодом видається індивідуальне завдання. Практика проводиться на різноманітних підприємствах та в різних організаціях різних галузей господарювання, а саме таких як галузі освіти, охорони здоровя, праці, державного управління, які є оснащені відповідним чином. Також можливе проведення практики на базах навчальних закладів.

Основним завданням практики є оволодіння практикантами сучасних методів розробки ПЗ зокрема, та способами організації підприємницької діяльності в цілому.

Знання, набуті в результаті проходження практики є хорошим фундаментом для подальшої професійної діяльності. Набутий досвід дасть змогу в майбутньому швидше адаптовуватися до іншого виробничого середовища, комфортніше себе почувати на новому місці роботи, у новому робочому колективі, більш легко розбиратись із новими незнайомими технологіями та фреймворками.

1. Загальна характеристика підприємства практики
   1. Мета та рік створення

Підприємство займається наданням широкого спектру послуг з розробки програмного забезпечення, автоматизації та інформатизації різноманітних комп’ютерних систем. У ньому працює близько 100 співробітників різних кваліфікаційних напрямів, що дає змогу ефективно виконувати проекти різної складності в конкретно визначені строки. Офіс підприємства знаходиться в м. Хмельницькому.

MassMedia Group була заснована в 2007 році та розпочиналась як невелика веб-студія. На ранніх етапах було лише 2 працівника – Сергій та Олександр. Згодом до них приєднався ще Андрій Войтович, який до цього був їхнім замовником. Основною використовуваною мовою чого часу в компанії була мова PHP, яка й досі зараз переважає в відділі розробки компанії.

* 1. Форма власності

На законному рівні компанія є існує просто як ФОП Вержбицький С.В. Тобто, MassMedia Group – маркетингова назва компанії. Фізична особа підприємець (ФОП) – згідно з українським законодавством, фізична особа, яка реалізує свою здатність до праці шляхом самостійної, на власний ризик діяльності з метою отримання прибутку.

Окрім цього, MassMedia Group співпрацює зі своїми робітниками за контрактом так як із підприємцями. Тобто, кожен працівник організації повинен також бути оформленим як ФОП.

* 1. Основні види продукції, послуг чи видів діяльності

Зараз основною сферою роботи є розробка різних ступенів складності веб додатків переважно у сфері автоматизації бізнесу. Переважна більшість проектів – outstaff, тобто компанія є проміжною ланкою між розробниками та клієнтами. Для розробників складається чітке, зрозуміле їм технічне завдання, а для клієнтів – певні рекомендації по розвитку їх бізнесу. Тобто в даному ключі, MassMedia виступає також консалтинговою компанією.

В якості технологічних рішень, на поточний момент часу переважають:

* PHP (Laravel, Symfony);
* JavaScript (React, Angular, Vue);
* Node.js;
* Ruby;
* PostgreSQL;
* MySQL;
* Redis;
* Elasticsearch.
  1. Характеристика підрозділів підприємства

Організаційна структура підприємства складається з трьох співвласників, управлінського персоналу різних відділів та персоналу по кожному відділу. Схема структури підприємства зображена на рисунку 1.

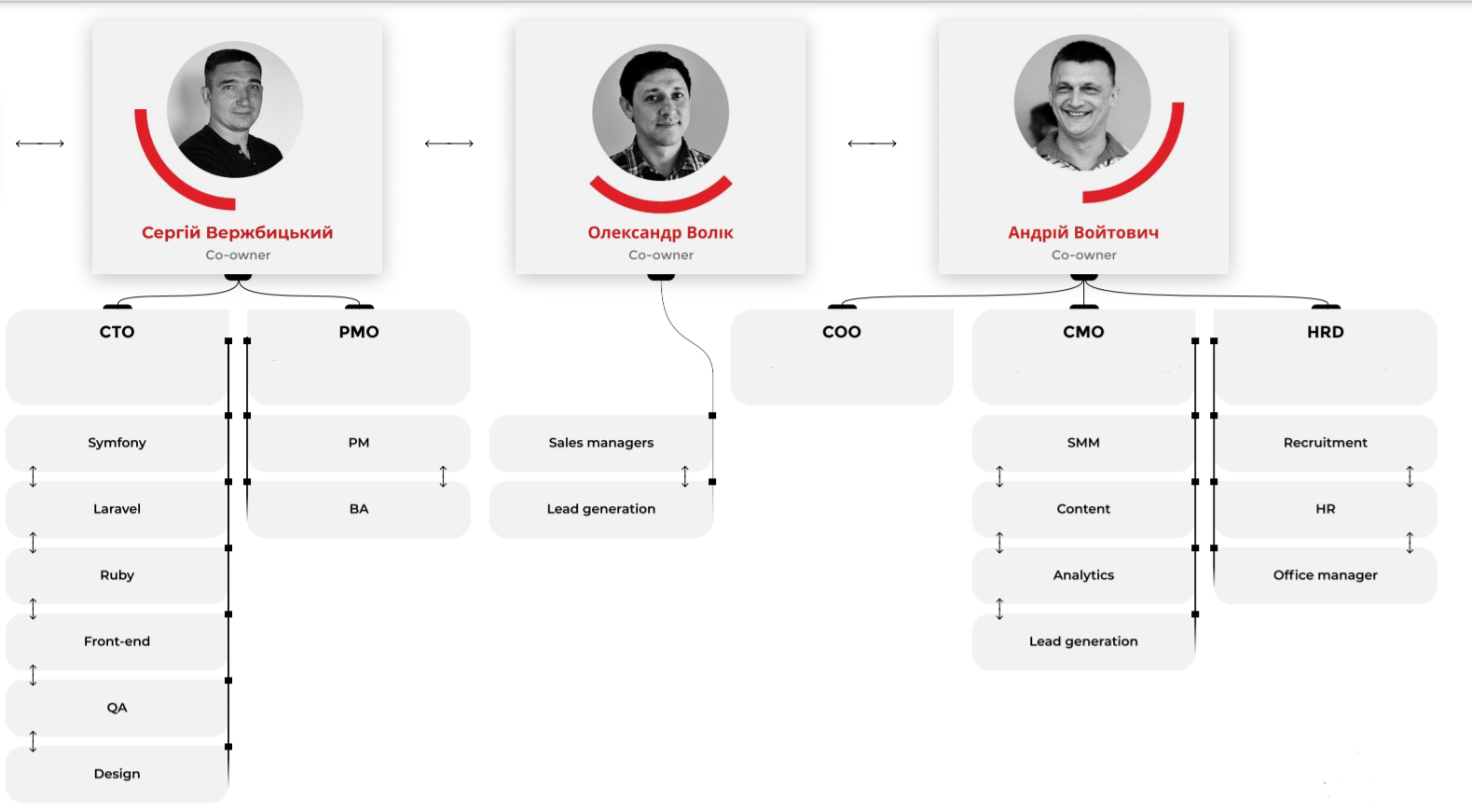


Рисунок 1 – Схема структури підприємства

Таким чином, підприємство складається з шести підрозділів, кожен з яких виконує свою роль в загальній справі розробки ПЗ. Більшість підрозділів розподіляється на відділи. Наприклад, CTO (chief technology officer), тобто технічний директор над виконавцями проектів, відповідає за набір ефективної команди згідно технічних навиків для виконання того чи іншого проекту, PMO (project management officer), тобто директор менеджерів проектів, відповідальний за організацію роботи відділу менеджменту, бізнес-аналітики та суміжних відділів. Такий розподіл зроблений для максимально ефективного виконання та менеджменту замовлених проектів.

Структура нижчих рівнів команди налічує такі спеціальності як технік-програміст (займається різноплановими задачами), front-end розробник (суто візуальна частина), back-end розробник (суто серверна частина), QA інженер (тестування правильності роботи пз), веб-дизайнер (проектування графічного відображення програмного забезпечення відповідно до фунціональних вимог).

* 1. Конкурентні переваги підприємства

Підприємство робить все можливе для розвитку дружньої атмосфери в своєму колективі, щоб співробітники ефективніше працювали в команді. На даний момент часу компанія налічує понад 70 працівників. З технічної сторони, кожному розробнику складається індивідуальний план розвитку, який кожен розробник виконує для покращення своїх знань. Також, компанія надає можливість своїм працівникам відвідувати курси англійської за кошти організації.

В даний момент підприємство активно розширюється, з’являються нові команди розробки та спеціалізації, спеціалісти які працюють в інших сферах та з іншим стеком технологій.

1. Посадові обов’язки IT-фахівців
   1. Технік програміст

Технік-програміст користується готовими програмами, робота його полягає в тому, щоб вирішити поставлені завдання. Він бере участь в розробках нескладних алгоритмів на різних мовах програмування. Для автоматичного введення даних в комп'ютер необхідні різні носії інформації. Цей фахівець стежить за їх наявністю і справністю, займається веденням технічної документації, обліком обсягів виконаних завдань, усуненням несправностей машини.

* 1. Веб-дизайнер

Web-дизайнер – це людина, що володіє художнім смаком і знаннями інтернет-технологій, який створює Web-сторінки і об'єднує їх в Web-сайти. Головне завдання web-дизайнера – оформити веб сайт так, щоб зацікавити якомога більшу кількість користувачів. Тобто, веб-дизайнер — спеціаліст галузі веб розробки, що охоплює цілий ряд напрямів і дисциплін із створення та супроводу сайтів або веб застосунків, таких як графічний вебдизайн, проєктування інтерфейсів, авторинг, використовність та оптимізація для пошукових систем.

* 1. Front-end розробник

Front-end розробник – це фахівець, який вміє верстати веб-сторінки, має хороші знання мови програмування JavaScript, знає один або кілька JavaScript-фреймворків (React, Angular, Vue.js), а також цілий ряд інших веб-технологій, які використовуються під час створення клієнтської сторони веб-сайту. Робота фахівця включає в себе як оригінальні дизайнерські, так технічні рішення в області проектування веб-інтерфейсів, що забезпечують зручність користування веб-ресурсом. Професійний Front-end developer повинен постійно ставити себе на місце віддаленого користувача і створювати сторінки з інтуїтивно-зрозумілим інтерфейсом. Таким чином, розробка функціональності користувальницького інтерфейсу сайту та налаштування взаємодії з серверною частиною – це головне завдання Frontend розробника.

* 1. Back-end розробник

Back-end – це фахівець, який займається програмно-адміністративною частиною веб-додатків  внутрішнім змістом системи, серверними технологіями – базою даних, архітектурою, програмної логікою. Back-end розробник повинен знати хоча б одну, але бажано декілька серверних мов програмування. Наприклад, як PHP, Python, Ruby, Java, Perl, Node JS. Однак, знань лише мови програмування буде не достатньо, так як в основному зараз проекти пишуться на якомусь з фреймворків. Тому додатково до обраної мови, спеціалісту потрібно вивчити хоча б основні елементи фреймворків (як Laravel, Symfony, Django, Express, Spring та ін). Знання back-end розробника на цьому не закінчуються, так як будь-який поважаючий себе програміст повинен вміти проектувати бази даних та створювати архітектуру високонавантажених систем.

* 1. Team lead

Тімлід – це IT-фахівець, який керує своєю командою розробників, володіє технічною стороною, бере участь в роботі над архітектурою проекту, займається рев'ю коду (англ. code review), а також розробкою деяких особливо складних завдань на проекті.

Під управлінську роль тімліда потрапляють такі обов'язки, як власне менеджмент, розподіл і делегування завдань, всілякі оцінки та складання робочого графіка, контроль стану проекту, а також мітинги, комунікації з замовником, керівництвом і всіма членами команди (розробниками, архітекторами, тестувальниками, менеджерами).

Під технічну роль: участь в написанні технічної документації, вибір технологій для проекту, розробка архітектури, code review, менторинг джуніорів, проведення технічних співбесід, грамотне залучення нових членів команди в робочий процес, відповідальність за технічну частину проекту.

* 1. Менеджер проекту (PM)

Це фахівець, в якого головним завданням є управління проектом в цілому: проектування роботи і розстановка пріоритетів, планування виконання завдань, контроль, комунікації, а також оперативне вирішення проблем.

В основному це нетехнічна посада, але більшість українських проектних менеджерів в IT – це колишні розробники або тестувальники. Якщо вірити статтистиці, то половина теперішніх менеджерів проектів – колишні технічні фахівці в основному розробники та QA інженери.

Основний обов'язок і відповідальність PM - довести ідею замовника до реалізації в установлений строк використовуючи існуючі ресурси. В рамках цього завдання менеджеру необхідно побудувати план розробки, організувати команду, налаштувати процес роботи над проектом, забезпечити зворотний зв'язок між командами і замовником, усувати перешкоди для команд, контролювати якість і поставку продукту вчасно.

* 1. Бізнес Аналітик (BA)

Бізнес-аналітик – фахівець з поліпшення бізнес-процесів. Технічні знання для цієї ролі не найголовніше, куди важливіше знання процесів або здатність швидко їх зрозуміти. Потрібно вміти слухати, ставити правильні питання і виокремлювати проблеми користувачів.

BA досліджує проблему замовника, шукає рішення і оформлює його концепцію в формі вимог, на які в подальшому будуть орієнтуватися розробники при створенні продукту.

Головне завдання бізнес-аналітика – виявити проблеми бізнесу замовника і знайти максимально ефективне рішення. Для цього він повинен володіти знаннями в предметній області. Бізнес-аналітик працює з технічними вимогами на всіх етапах життєвого циклу розробки ПЗ і постійно виступає посередником між замовником і командою програмістів.

* 1. Інженер із забезпечення якості (QA інженер)

Інженер із забезпечення якості - це фахівец, діяльність якого направлена на вдосконалення процесу розробки ПО, запобігання дефектам та виявлення помилок у роботі продукту.

Основна задача QA - забезпечення якості. Інженер фокусує увагу на процесах розробки ПО, покращує їх, запобігає появі дефектів і проблем. Іншими словами, фахівець перевіряє, що робота зроблена правильно та належним чином.

Фахівець з якості повинен знати: основні технології побудови ПО і структури програмних комплексів; знання операційних систем на рівні advanced user, мова запитів SQL, скриптові мови, принципи програмування, спеціальне програмне забезпечення для автоматизованого тестування і реєстрації помилок, англійська мова (як мінімум – на рівні читання технічної документації), принципи створення тест-кейсів.

1. Звіт з індивідуального завдання
   1. Технічне завдання

Під час проходження переддипломної практики індивідуальним завданням було створення бібліотеки для легковажної, швидкої та ефективної організації структури програмних додатків на основі шаблону Event Sourcing.

Бібліотека повинна надавати можливість зберігати події доменної області разом із самим записом доменного об’єкта в базі даних при його оновленні.

Технологіями виступають: мова програмування PHP, базова бібліотека Doctrine ORM, бібліотека тестування PHPUnit. Для локальної розробки та тестування виступають технології Docker та Docker Compose.

* 1. Проектування

Event Sourcing (ES) – це підхід зберігання даних, при якому замість кінцевого результату зберігається низка записів про події сталися з деякою сутністю.

Ключові механізми, які як раз і дозволяють будувати різний корисний функціонал наступні:

* кожній події дається ім'я, яке визначає його значення;
* немає обмежень на кількість подій для сутності. Відповідно нові події можуть відображати, як і нові види вчинених дій, так і розширювати вже існуючі;
* події, що вже відбулися незмінні.

Ми можемо дістати з бази даних стан програми, щоб з’ясувати поточний стан світу, і це відповідає на багато питань. Однак бувають випадки, коли ми не просто хочемо побачити, де ми перебуваємо, ми також хочемо знати, як ми туди потрапили.

Event Sourcing гарантує, що всі зміни стану програми зберігаються як послідовність подій. Ми можемо не просто діставати ці події з бази в сенсі як набір певних логів, але також можемо використовувати журнал подій для реконструкції минулих станів і як основу для автоматичного налаштування стану для боротьби із зворотними змінами.

Основна ідея ES полягає в тому, що кожна зміна стану програми фіксується в об’єкті події, і що ці об’єкти подій самі зберігаються в тій послідовності, в якій вони були застосовані протягом того самого періоду життя, що і сам стан програми.

Розглянемо простий приклад із повідомленнями про доставку. У цьому прикладі ми маємо багато кораблів у відкритому морі, і ми повинні знати, де вони знаходяться. Простий спосіб зробити це – мати програму відстеження з методами, які дозволять нам визначити, коли корабель прибуває у порт або відбуває від нього.

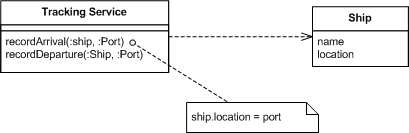
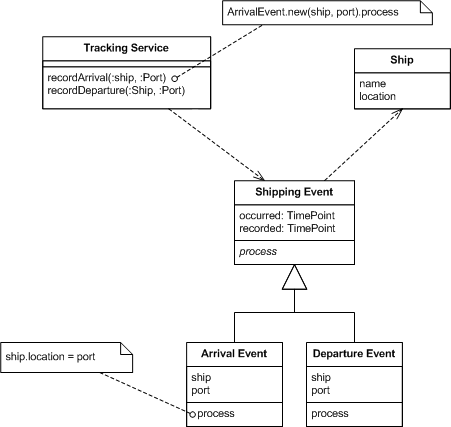


Рисунок 1 – Простий інтерфейс для відстеження руху суден

У цьому випадку, при виклику TrackingService, він знаходить відповідне судно та оновлює його місцезнаходження. Програмні об'єкти кораблів фіксують поточний відомий їх стан. ES додає ще один крок до цього процесу. Тепер сервіс створює об'єкт події для запису змін та обробляє його для оновлення інформації корабля. Тобто, первинним для оновлення інформації про судно є подія, що спричиняє це оновлення. Ми використовуємо об’єкт події для побудови подальшої логіки в сервісному класі. TrackingService отримує потрібну для оновлення інформацію з ShippingEvent, і вже на цій основі будує подальшу логіку.

   
Рисунок 2 – Використання події для фіксації змін

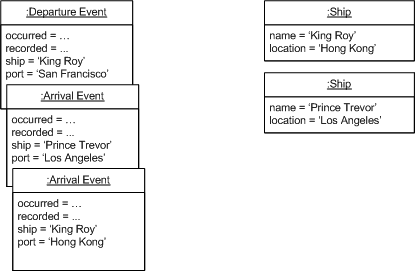
Якщо дивитись лише на обробку, то це лише зайвий рівень опосередкованості. Цікава відмінність полягає в тому, що ми подивимось на те, що зберігається в додатку після кількох подій. Давайте уявимо кілька простих змін:

* корабель "Кінг Рой" відправляється з Сан-Франциско;
* корабель "Принц Тревор" прибуває до Лос-Анджелеса;
* корабель "Кінг Рой" прибуває до Гонконгу.

З простим сервісом ми бачимо лише остаточний стан, отриманий судновими об’єктами, тобто це – стан програми.

   
Рисунок 3 – Стан після декількох рухів, відстежених простим трекером

За допомогою ES ми додатково фіксуємо кожну подію. Якщо ми використовуємо постійне сховище, події зберігатимуться так само, як і суднові об'єкти. Корисно також сказати, що ми зберігаємо дві різні речі - стан програми та журнал подій.

   
Рисунок 4 – Стан після декількох рухів, відстежених ES трекером

Найбільш очевидне, що ми отримали за допомогою ES це те, що тепер у нас є журнал усіх змін. Ми можемо не просто можемо побачити де знаходиться кожен корабель, але також знаємо де воно було. Однак це не такий великий виграш. Ми також могли б це зробити, зберігаючи історію минулих портів у судновому об'єкті, або записуючи у файл логів кожного разу, коли судно рухається. І те, і інше може дати нам адекватну історію.

Ключем в Event Sourcing є те, що ми гарантуємо, що всі зміни об’єктів доменної моделі ініціюються об’єктами подій. Це призводить до ряду корисних можливостей, які можна побудувати поверх журналу подій:

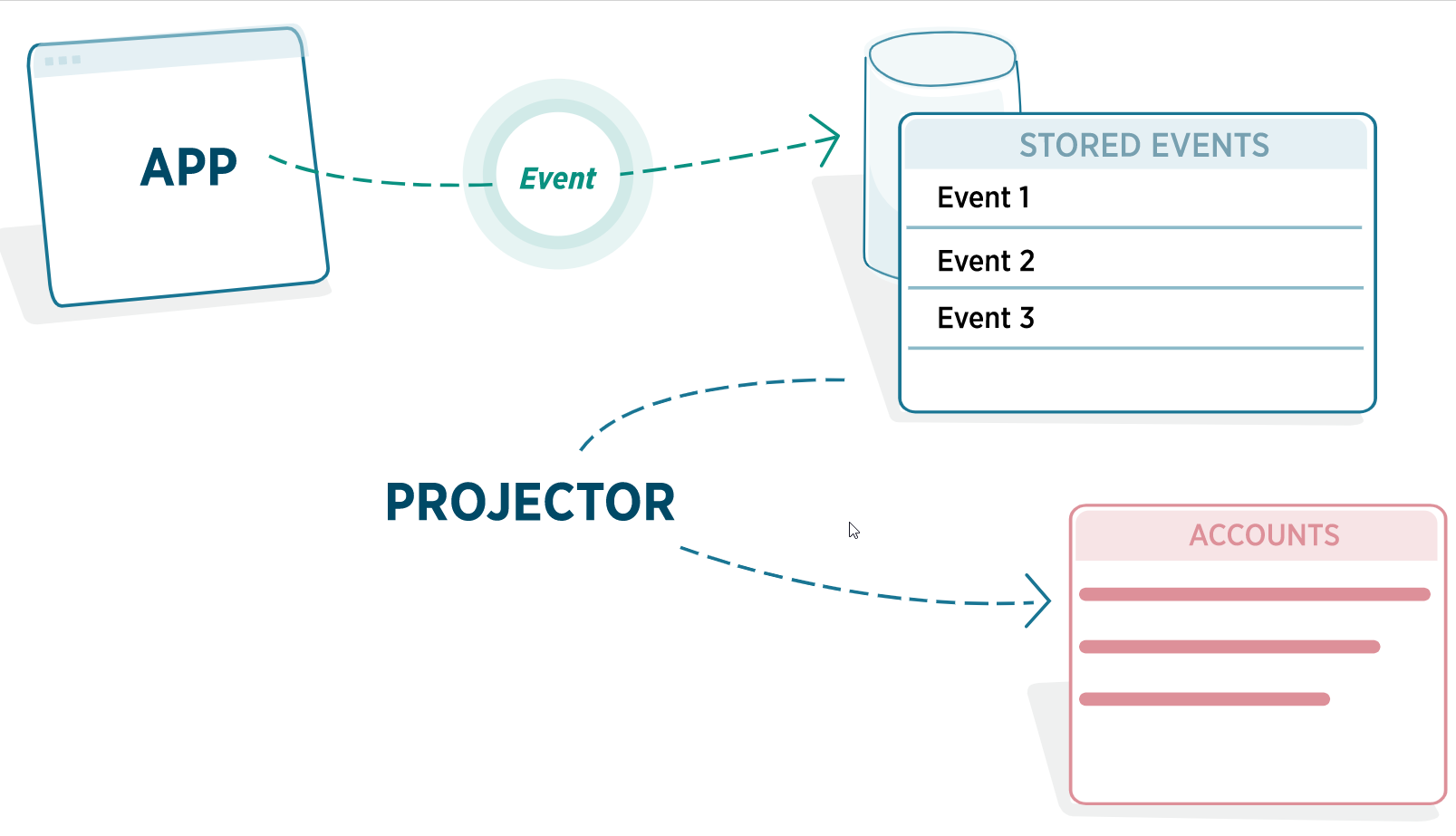
* повна перебудова – ми можемо повністю відкинути стан програми та відновити його, повторно запустивши події з журналу подій на порожній програмі;
* часовий запит – ми можемо визначити стан програми в будь-який момент часу. Умовно ми робимо це, починаючи з порожнього стану та повторюючи події до певного часу чи певної події. Ми можемо піти навіть далі, розглянувши кілька часових ліній (аналогічно розгалуженню в системі контролю версій);
* відтворення подій – якщо ми виявимо, що минула подія була неправильною, ми можемо обчислити наслідки правильного порядку подій, змінивши некоректну подію коректною, а потім відтворивши нову подію та наступні події. Або, взагалі відкидаючи стан програми та відтворюючи всі події з правильною подією в послідовності. Такий же прийом може обробляти події, отримані в неправильній послідовності – типова проблема систем, які комунікують за допомогою асинхронних повідомлень.

Поширеним прикладом програми, яка використовує ES, є система контролю версій. Така система використовує тимчасові запити досить часто. До прикладу, поточний стан файлів в Git – це закешований результат всіх попередніх комітів. Для переключення на певний коміт, умовно кажучи, система контролю версій відтворює всі попередні коміти аж до того, на який ми переключаємось. Насправді, там все зроблено ефективніше так що зберігаються зворотні події (коли ми переключаємось на попередній коміт, то буде відтворена одна подія, а не вся історія комітів).

Сховище стану програми. Найпростіший спосіб розуміння ES – це обчислення запитуваного стану програми, починаючи з порожнього стану програми, застосовуючи події для досягнення бажаного поточного стану. Аналогічно зрозуміло чому це повільний процес, особливо якщо є доволі багато подій.

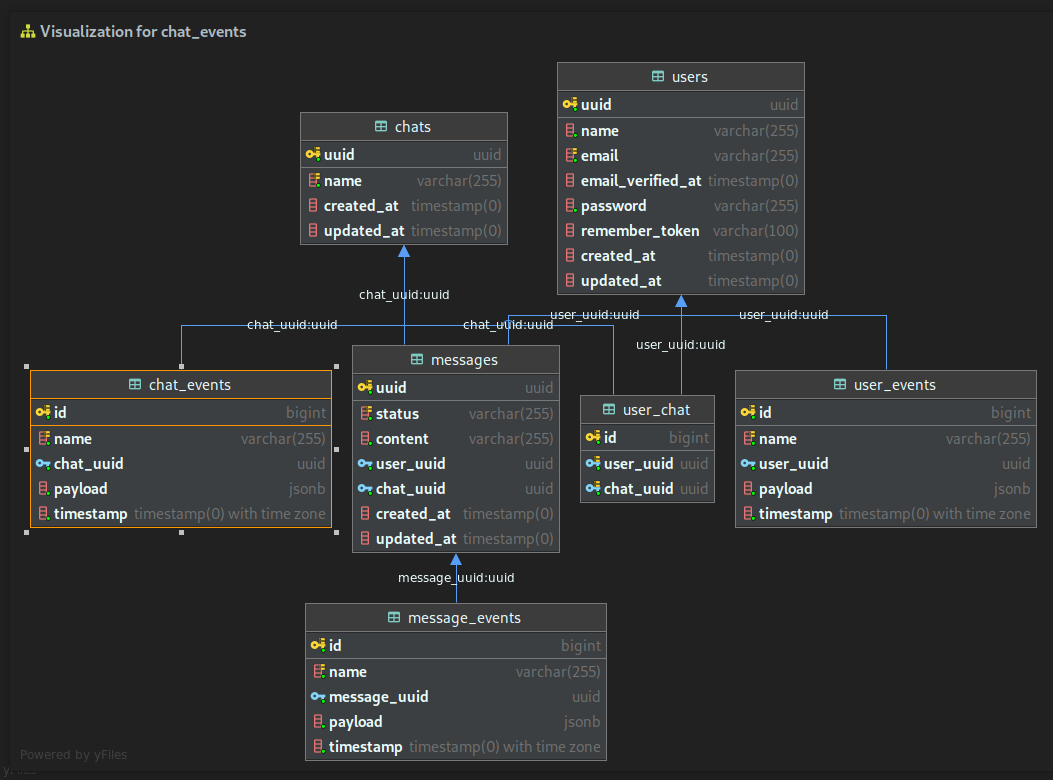
У багатьох додатках частіше вимагають останній стан програми, аніж роботу з історією подій. Для цього в ES є проекції.

Стани програми можна зберігати або в пам'яті, або на диску. Оскільки стан програми є суто похідним із журналу подій, ви можете кешувати його де завгодно. Система, що використовується протягом робочого дня, може бути запущена на початку дня з моментального знімка і зберегти поточний стан програми в пам'яті. Якщо вона аварійно завершує роботу, то повторює події з сховища, починаючи від нічного знімку. В кінці робочого дня можна зробити новий знімок стану. Нові знімки можна робити в будь-який час паралельно, не збиваючи запущений додаток.

Рисунок 5 – Архітектура ES системи

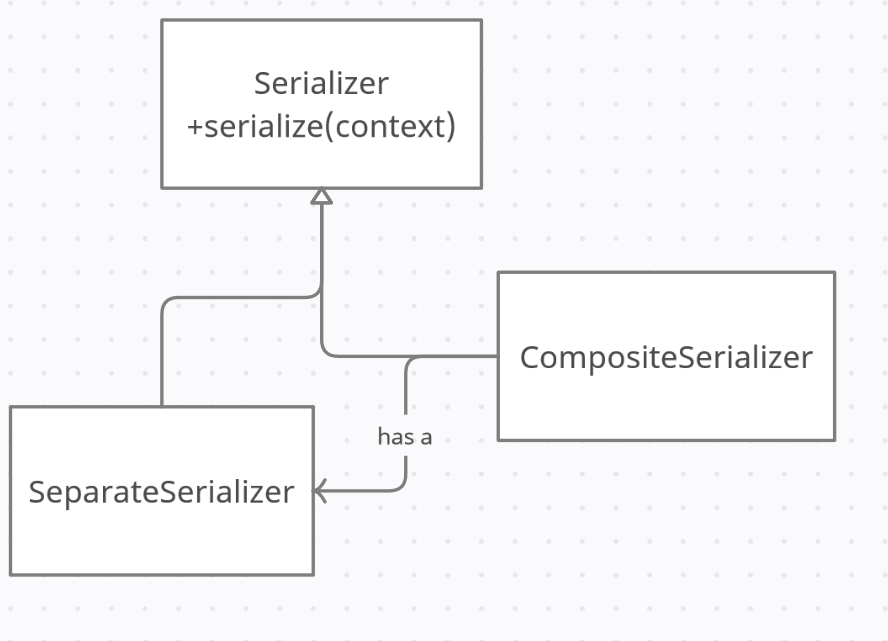
Розроблена бібліотека повинна дати можливість будувати додатки за принципом акценту на збереженні поточного стану програми, а події будуть зберігатись під час цього збереження так, що нам не потрібно буде акцентувати на цьому увагу. Тобто, основні таблиці бази даних в ES системах будуть вважатись проекціями. Однак, проекції – не зовсім правильний в даному контексті термін, адже тут зберігаються основні дані, які не будуть просто перезатиратись, але всі запити насправді будуть виконуватись над реальними даними. Таким чином, буде усунено недолік ES систем – час на оновлення проекції згідно з останніми подіями.

Основою архітектури буде те, що кожна таблиця, яка зберігає об’єкт доменної області, повинна мати таблицю для збереження подій, що відбулись над цим об’єктом протягом його життєвого циклу. Наприклад, таблиця messages має відповідну їй таблицю message\_events, chats – відповідно chat\_events, а users – user\_events. Зв’язки ж між основними таблицями зберігаються як і в звичайних традиційних системах.

   
Рисунок 6 – Приклад структури бази даних ES додатка

Для реалізації збереження подій, будуть використовуватись lifecycle hooks, які надає бібліотека doctrine orm. А саме, потрібно preFlush та postLoad. З їх використанням можна буде зберігати поля об’єкта події в базі даних та витягувати їх при вивантаженні з БД. Тобто, по-суті це операції серіалізації та десеріалізації. З цього можна зробити висновок, що будуть компоненти серіалізації та десеріалізації.

Однак, ми наперед не можемо точно знати яким чином повинно бути серіалізовано те чи інше поле об’єкта. Для деяких ми знаємо, наприклад, коли серіалізуємо об’єкт доменної області, то зберігаємо просто його ідентифікатор. У випадку серіалізації примітивного значення, можна просто його записати в базу даних після певного форматування. Також буде додано можливість серіалізувати за власною методикою, або просто вказавши в анотації бажаний стандартний тип серіалізації.

   
Рисунок 7 – Схема класів серіалізації

Структура передбачає, що ми будемо мати інтерфейс серіалізатора, який може бути реалізований або конкретно заточеним під якусь задачу серіалізатором (як наприклад, серіалізація стандартних зареєстрованих типів), або ж композитним серіалізатором, що вибере перший окремий серіалізатор, що знає як опрацювати конкретний об’єкт контексту. Ця ж сама схема стосується десеріалізації.

Всього є п’ять серіалізаторів та десеріалізаторів для конкретних випадків:

* Entity – для роботи з об’єктами домену, серіалізується як ідентифікатор;
* Typed – для роботи з стандартними об’єктами, функціонал серіалізації надається самою бібліотекою Doctrine ORM;
* Embedded – для серіалізації об’єктів-значень, конфігурація серіалізації вже задана користувачем, коли він об’являв метадані для збереження в базі даних;
* Catable – для серіалізації об’єктів згідно з логікою користувача, серіалізується згідно контракту якщо об’єкт його реалізує;
* Noop – для серіалізації примітивних значень або у випадку, якщо не знайдено більш відповідної стратегії.
  1. Програмна реалізація

Коли користувач захоче використати бібліотеку, йому потрібно буде реалізувати інтерфейс AggregateRoot, наданий поточною бібліотекою. Цей інтерфейс об’являє метод getPrimaryName, що повинен вернути назву поля, в якому міститься первинний ключ та getPrimaryName, що поверне власне первинний ключ. Тобто, початковим етапом для клієнта буде реалізація цих методів.

use Ramsey\Uuid\UuidInterface as Uuid;

use Rela589n\DoctrineEventSourcing\Entity\AggregateRoot;

class User implements AggregateRoot

{

private Uuid $uuid;

public static function getPrimaryName(): string

{

return 'uuid';

}

public function getPrimary(): Uuid

{

return $this->uuid;

}

}

Наступним кроком, користувач повинен створити базовий абстрактний клас події відповідно до об’єкта доменної області. Цей клас повинен реалізувати інтерфейс Contract\AggregateChanged. Насправді, користувачу самому не потрібно реалізувати його, так як він може або підключити трейт Concern\AggregateChanged, що реалізує всі потрібні методи. Іншим варіантом буде наслідування від базового класа AggregateChanged, що поставляється з бібліотекою.

use Rela589n\DoctrineEventSourcing\Event\AggregateChanged;

abstract class UserEvent extends AggregateChanged

{

public function \_\_construct(User $user)

{

parent::\_\_construct(entity: $user);

}

}

Всі події, що відбуваються з даною сутністю повинні наслідуватись від створеної на попередньому кроці. Також, кожен такий клас події має реалізувати метод NAME, результат виконання якого буде зберігатись у полі імені події в базі даних. До прикладу, подія реєстрації користувача.

class UserRegistered extends UserEvent

{

private Login $login;

private Password $password;

private UserName $userName;

private function \_\_construct(User $user, Login $login, Password $password, UserName $name)

{

parent::\_\_construct($user);

$this->login = $login;

$this->password = $password;

$this->userName = $name;

}

public static function withCredentials(User $user, Login $login, Password $password, UserName $name): self

{

return new self($user, $login, $password, $name);

}

// bunch of getters here

public function NAME(): string

{

return 'user\_registered';

}

}

Тобто, на поточному етапі йде побудова предметної області використовуючи події, які в ній відбуваються. До прикладу, реалізувавши логіку реєстрації користувачів ми можемо отримати наступний клас:

use Rela589n\DoctrineEventSourcing\Event\Exceptions\UnexpectedAggregateChangeEvent;

class User implements AggregateRoot

{

private Uuid $uuid;

private Login $login;

private Password $password;

private UserName $name;

private DateTimeInterface $createdAt;

private DateTimeInterface $updatedAt;

/\*\* @var Collection<UserEvent> \*/

private Collection $recordedEvents;

/\*\* @var UserEvent[] \*/

private array $newlyRecordedEvents = [];

private function \_\_construct(Uuid $uuid)

{

$this->uuid = $uuid;

$this->recordedEvents = new ArrayCollection();

}

public static function register(Uuid $uuid, Login $login, Password $password, UserName $name): self

{

$user = new self($uuid);

$user->recordThat(

UserRegistered::withCredentials($user, $login, $password, $name)

);

return $user;

}

private function recordThat(UserEvent $event): void

{

switch (true) {

case $event instanceof UserRegistered:

$this->createdAt = $event->getTimestamp();

$this->login = $event->getLogin();

$this->password = $event->getPassword();

$this->name = $event->getUserName();

break;

default:

throw new UnexpectedAggregateChangeEvent($event);

}

$this->updatedAt = $event->getTimestamp();

$this->newlyRecordedEvents[] = $event;

$this->recordedEvents->add($event);

}

public static function getPrimaryName(): string

{

return 'uuid';

}

public function getPrimary(): Uuid

{

return $this->uuid;

}

public function releaseEvents(): array

{

$events = $this->newlyRecordedEvents;

$this->newlyRecordedEvents = [];

return $events;

}

}

Наступним же чином, потрібно налаштувати логіку збереження подій разом із моделями в базу даних. Для цього, в конфігурації базового класа події прописуємо наступні налаштування:

<entity name="App\Models\User\Events\UserEvent" table="user\_events" inheritance-type="SINGLE\_TABLE">

<many-to-one field="entity" target-entity="App\Models\User\Doctrine\User" inversed-by="recordedEvents">

<join-column name="user\_uuid" referenced-column-name="uuid"/>

</many-to-one>

<discriminator-column name="name" type="string"/>

<discriminator-map>

<discriminator-mapping value="user\_changed\_login" class="App\Models\User\Events\UserChangedLogin"/>

<discriminator-mapping value="user\_registered" class="App\Models\User\Events\UserRegistered"/>

</discriminator-map>

</entity>

Значення в тегаї discriminator-mapping повинні бути назвами констант, що ми об’явили на кроці створення подій, а атрибут class повинен містити повний шлях до класу конкретної події.

Також, потрібно об’явити кожен такий клас події як Entity. В файлі xml конфігурації це має наступний вигляд.

<entity name="App\Models\User\Events\UserRegistered">

</entity>

Якщо ж ми не використовуємо xml конфігурацію, а скоріше анотації, тоді для того, щоб об’явити клас доменним об’єктом можна використати @Entity.

Налаштування збереження моделі не залежить від бібліотеки, а залишаєтсья стандартним для Doctrine:

<entity name="App\Models\User\Doctrine\User" table="users">

<!-- actually, doctrine doesn't have uuid type, this one is from third-party library -->

<id name="uuid" type="uuid" column="uuid">

<generator strategy="NONE"/>

</id>

<embedded name="login" class="App\Models\User\VO\Login" use-column-prefix="false"/>

<embedded name="password" class="App\Models\User\VO\Password" use-column-prefix="false"/>

<embedded name="name" class="App\Models\User\VO\UserName" use-column-prefix="false"/>

<field name="createdAt" type="datetimetz" column="created\_at"/>

<field name="updatedAt" type="datetimetz" column="updated\_at"/>

<!-- other fields and relations -->

<one-to-many field="recordedEvents" target-entity="App\Models\User\Events\UserEvent" mapped-by="entity">

<cascade>

<cascade-persist/>

</cascade>

</one-to-many>

</entity>

Тепер користувач може перевірити робочість коду наступним чином:

$email = 'johndoe'.Str::random().'@example.com';

$user = User::register(

Uuid::uuid4(),

Login::fromString($email),

Password::fromRaw('hello world'),

UserName::fromString('John Doe'),

);

$this->entityManager->persist($user);

$this->entityManager->flush();

Клас \Rela589n\DoctrineEventSourcing\Event\AggregateChanged:

**<?php  
  
declare**(strict\_types=1);  
  
**namespace** Rela589n\DoctrineEventSourcing\Event;  
  
**abstract class** AggregateChanged **implements** Contract\AggregateChanged  
{  
 **use** Concern\AggregateChanged;  
}

Клас \Rela589n\DoctrineEventSourcing\Event\Exceptions\UnexpectedAggregateChangeEvent:

**<?php  
  
declare**(strict\_types=1);  
  
**namespace** Rela589n\DoctrineEventSourcing\Event\Exceptions;  
  
**use** LogicException;  
  
**final class** UnexpectedAggregateChangeEvent **extends** LogicException  
{  
 **private $event**;  
  
 **public function** \_\_construct($event)  
 {  
 **parent**::*\_\_construct*(**'Unexpected event: '**.$event::**class**);  
 $this->**event** = $event;  
 }  
  
 **public function** getEvent()  
 {  
 **return** $this->**event**;  
 }  
}

Клас \Rela589n\DoctrineEventSourcing\Event\Contract\AggregateChanged:

**<?php  
  
declare**(strict\_types=1);  
  
**namespace** Rela589n\DoctrineEventSourcing\Event\Contract;  
  
**use** DateTimeInterface **as** DateTime;  
**use** Doctrine\ORM\Event\LifecycleEventArgs;  
**use** Doctrine\ORM\Event\PreFlushEventArgs;  
  
**interface** AggregateChanged  
{  
 **public function** NAME(): **string**;  
  
 **public function** getTimestamp(): *DateTime*;  
  
 **public function** onPreFlushAggregateChanged(PreFlushEventArgs $args): **void**;  
  
 **public function** onPostLoadAggregateChanged(LifecycleEventArgs $args): **void**;  
}

Клас \Rela589n\DoctrineEventSourcing\Event\Concern\AggregateChanged:

**<?php  
  
declare**(strict\_types=1);  
  
**namespace** Rela589n\DoctrineEventSourcing\Event\Concern;  
  
**use** DateTimeImmutable;  
**use** DateTimeInterface **as** DateTime;  
**use** Doctrine\ORM\EntityManagerInterface;  
**use** Doctrine\ORM\Event\LifecycleEventArgs;  
**use** Doctrine\ORM\Event\PreFlushEventArgs;  
**use** ReflectionClass;  
**use** ReflectionProperty;  
**use** Rela589n\DoctrineEventSourcing\Entity\AggregateRoot;  
**use** Rela589n\DoctrineEventSourcing\Event\Annotations\HideFromPayload;  
**use** Rela589n\DoctrineEventSourcing\Event\Concern\TypesMeta\CollectEventSerializeMeta;  
**use** Rela589n\DoctrineEventSourcing\Event\Concern\TypesMeta\CollectEventSerializeMetaImpl;  
**use** Rela589n\DoctrineEventSourcing\Event\Concern\TypesMeta\CollectEventSerializeMetaInMemoryCacheDecorator;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Composed\ComposedDeserializer;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Composed\ComposedSerializer;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Context\DeserializationContext;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Context\SerializationContext;  
  
*/\*\** ***@mixin*** *\Rela589n\DoctrineEventSourcing\Event\AggregateChanged \*/***trait** AggregateChanged  
{  
 *#[HideFromPayload]* **protected int** $id;  
  
 *#[HideFromPayload]* **protected string $name**;  
  
 *#[HideFromPayload]* **protected** *DateTime* **$timestamp**;  
  
 *#[HideFromPayload]* **protected array $payload**;  
  
 *#[HideFromPayload]* **protected** AggregateRoot **$entity**;  
  
 *#[HideFromPayload]* **private** ?CollectEventSerializeMeta **$collectPropertiesMeta** = **null**;  
  
 *#[HideFromPayload]* **private** ?ComposedSerializer\Factory\Context **$serializerFactoryContext** = **null**;  
  
 *#[HideFromPayload]* **private** ?ComposedSerializer\Factory **$serializerFactory** = **null**;  
  
 *#[HideFromPayload]* **private** ?ComposedDeserializer\Factory\Context **$deserializerFactoryContext** = **null**;  
  
 *#[HideFromPayload]* **private** ?ComposedDeserializer\Factory **$deserializerFactory** = **null**;  
  
 **public function** \_\_construct(AggregateRoot $entity)  
 {  
 $this->**entity** = $entity;  
 $this->**timestamp** = **new** DateTimeImmutable();  
 $this->**name** = $this->NAME();  
 $this->**payload** = [];  
 }  
  
 **private function** collectPropertiesMeta(): CollectEventSerializeMeta  
 {  
 **return** $this->**collectPropertiesMeta** ?? $this->**collectPropertiesMeta** = $this->makeMetaCollector();  
 }  
  
 **private function** makeMetaCollector(): CollectEventSerializeMeta  
 {  
 **return new** CollectEventSerializeMetaInMemoryCacheDecorator(  
 **new** CollectEventSerializeMetaImpl(),  
 );  
 }  
  
 **public function** getTimestamp(): *DateTime* {  
 **return** $this->**timestamp**;  
 }  
  
 */\*\** ***@internal*** *\*/* **final public function** onPreFlushAggregateChanged(PreFlushEventArgs $args): **void** {  
 $reflectionClass = $this->reflectionClass($args->getEntityManager());  
 $properties = $this->properties($reflectionClass);  
  
 $serialize = $this->serializerFactory(  
 $this->**serializerFactoryContext** ?? $this->**serializerFactoryContext** =  
 ComposedSerializer\Factory\Context::*make*()  
 ->withEntityManager($args->getEntityManager())  
 ->withEntity($this->**entity**)  
 ->withPropertiesMeta($this->collectPropertiesMeta()(... $properties))  
 ->withCastArgumentsMap($this->castArguments())  
 )  
 ->make();  
  
 **foreach** ($properties **as** $property) {  
 $name = $property->getName();  
 $value = $property->getValue($this);  
  
 [**'name'** => $saveUnderName, **'value'** => $serialized] = $serialize(  
 SerializationContext::*make*()  
 ->withFieldName($name)  
 ->withValue($value)  
 ->withAttributes($this->**payload**)  
 );  
 $this->**payload**[$saveUnderName] = $serialized;  
 }  
 }  
  
 */\*\** ***@internal*** *\*/* **final public function** onPostLoadAggregateChanged(LifecycleEventArgs $args): **void** {  
 $reflectionClass = $this->reflectionClass($args->getEntityManager());  
 $properties = $this->properties($reflectionClass);  
  
 $deserialize = $this->deserializerFactory(  
 $this->**deserializerFactoryContext** ?? $this->**deserializerFactoryContext** = ComposedDeserializer\Factory\Context::*make*()  
 ->withEntityManager($args->getEntityManager())  
 ->withEntity($this->**entity**)  
 ->withPropertiesMeta($this->collectPropertiesMeta()(...$properties))  
 ->withCastArgumentsMap($this->castArguments())  
 )  
 ->make();  
  
 **foreach** ($properties **as** $property) {  
 $name = $property->getName();  
 $typename = (**string**)$property->getType()?->getName();  
  
 $property->setValue(  
 $this,  
 $deserialize(  
 DeserializationContext::*make*()  
 ->withFieldName($name)  
 ->withType($typename)  
 ->withSerialized($this->**payload**),  
 )  
 );  
 }  
 }  
  
 */\*\** ***@return*** *ReflectionProperty[] \*/* **private function** properties(ReflectionClass $reflectionClass): **array** {  
 $properties = *array\_filter*(  
 $reflectionClass->getProperties(),  
 **fn**(ReflectionProperty $property) => !$this->shouldNotSerialize($property)  
 );  
  
 *array\_walk*($properties, **static fn**(ReflectionProperty $property) => $property->setAccessible(**true**));  
  
 **return** $properties;  
 }  
  
 **private function** shouldNotSerialize(ReflectionProperty $property): **bool** {  
 **return** $property->isStatic()  
 || !**empty**($property->getAttributes(HideFromPayload::**class**));  
 }  
  
 **private function** reflectionClass(EntityManagerInterface $manager): ReflectionClass  
 {  
 **return** $manager->getClassMetadata(**static**::**class**)->**reflClass**;  
 }  
  
 **private function** serializerFactory(ComposedSerializer\Factory\Context $context): ComposedSerializer\Factory  
 {  
 **return** $this->**serializerFactory** ?? $this->**serializerFactory** = $this->makeSerializerFactory($context);  
 }  
  
 **protected function** makeSerializerFactory(ComposedSerializer\Factory\Context $context): ComposedSerializer\Factory  
 {  
 **return** ComposedSerializer\Factory\Impl::*fromContext*($context);  
 }  
  
 **private function** deserializerFactory(ComposedDeserializer\Factory\Context $context): ComposedDeserializer\Factory  
 {  
 **return** $this->**deserializerFactory** ?? $this->**deserializerFactory** = $this->makeDeserializerFactory($context);  
 }  
  
 **protected function** makeDeserializerFactory(ComposedDeserializer\Factory\Context $context): ComposedDeserializer\Factory  
 {  
 **return** ComposedDeserializer\Factory\Impl::*fromContext*($context);  
 }  
  
 **protected function** castArguments(): **array** {  
 **return** [];  
 }  
}

Клас \Rela589n\DoctrineEventSourcing\Event\Concern\TypesMeta\CollectEventSerializeMeta:

**<?php  
  
declare**(strict\_types=1);  
  
**namespace** Rela589n\DoctrineEventSourcing\Event\Concern\TypesMeta;  
  
**use** ReflectionProperty;  
  
**interface** CollectEventSerializeMeta  
{  
 **public function** \_\_invoke(ReflectionProperty...$properties): **array**;  
}

Клас \Rela589n\DoctrineEventSourcing\Event\Concern\TypesMeta\CollectEventSerializeMetaImpl:

**<?php  
  
declare**(strict\_types=1);  
  
**namespace** Rela589n\DoctrineEventSourcing\Event\Concern\TypesMeta;  
  
**use** ReflectionProperty;  
**use** Rela589n\DoctrineEventSourcing\Event\Annotations\SerializeAs;  
  
**final class** CollectEventSerializeMetaImpl **implements** CollectEventSerializeMeta  
{  
 **public function** \_\_invoke(ReflectionProperty...$properties): **array** {  
 $metaInfo = *array\_map*(  
 **static fn**(ReflectionProperty $property) => [  
 **'property'** => $property->getName(),  
 **'annotation'** => ($property->getAttributes(SerializeAs::**class**)[0] ?? **null**)?->newInstance(),  
 ],  
 $properties,  
 );  
  
 $metaInfo = *array\_filter*(  
 $metaInfo,  
 **static fn**(**array** $annotated) => $annotated[**'annotation'**],  
 );  
  
 **return** *array\_column*($metaInfo, **'annotation'**, **'property'**);  
 }  
}

Клас \Rela589n\DoctrineEventSourcing\Event\Concern\TypesMeta\CollectEventSerializeMetaInMemoryCacheDecorator:

**<?php  
  
declare**(strict\_types=1);  
  
**namespace** Rela589n\DoctrineEventSourcing\Event\Concern\TypesMeta;  
  
**use** ReflectionProperty;  
  
**final class** CollectEventSerializeMetaInMemoryCacheDecorator **implements** CollectEventSerializeMeta  
{  
 **private array $storage** = [];  
  
 **public function** \_\_construct(**private** CollectEventSerializeMeta **$decorated**) { }  
  
 **public function** \_\_invoke(ReflectionProperty ...$properties): **array** {  
 $hash = $this->calculateHash(...$properties);  
  
 **return** $this->**storage**[$hash]  
 ?? $this->**storage**[$hash] = $this->**decorated**->\_\_invoke(...$properties);  
 }  
  
 **private function** calculateHash(ReflectionProperty ...$properties): **int** {  
 $hash = 0;  
 $i = 1;  
  
 **foreach** ($properties **as** $property) {  
 $hash += $i \* spl\_object\_id($property);  
 $i = ($i \* 31) % (1 << 32);  
 }  
  
 **return** $hash;  
 }  
}

Клас \Rela589n\DoctrineEventSourcing\Event\Annotations\HideFromPayload:

**<?php  
  
declare**(strict\_types=1);  
  
**namespace** Rela589n\DoctrineEventSourcing\Event\Annotations;  
  
**use** Attribute;  
  
*#[Attribute(Attribute::TARGET\_PROPERTY)]***final class** HideFromPayload  
{  
}

Клас \Rela589n\DoctrineEventSourcing\Event\Annotations\SerializeAs:

**<?php  
  
declare**(strict\_types=1);  
  
**namespace** Rela589n\DoctrineEventSourcing\Event\Annotations;  
  
**use** Attribute;  
**use** Doctrine\DBAL\Types\Type;  
  
*#[Attribute(Attribute::TARGET\_PROPERTY)]***final class** SerializeAs  
{  
 **private** ?Type **$type**;  
 **private** ?**string $name**;  
  
 **public function** \_\_construct(?**string** $type = **null**, ?**string** $name = **null**)  
 {  
 $this->**name** = $name;  
 $this->**type** = (**null** !== $type)  
 ? Type::*getType*($type)  
 : **null**;  
 }  
  
 **public function** getType(): ?Type  
 {  
 **return** $this->**type**;  
 }  
  
 **public function** getName(): ?**string** {  
 **return** $this->**name**;  
 }  
}

Клас \Rela589n\DoctrineEventSourcing\Entity\AggregateRoot:

**<?php  
  
declare**(strict\_types=1);  
  
**namespace** Rela589n\DoctrineEventSourcing\Entity;  
  
**interface** AggregateRoot  
{  
 **public static function** getPrimaryName(): **string**;  
  
 **public function** getPrimary(): **mixed**;  
}

Клас \Rela589n\DoctrineEventSourcing\Serializer\Separate\SeparateDeserializer:

**<?php  
  
declare**(strict\_types=1);  
  
**namespace** Rela589n\DoctrineEventSourcing\Serializer\Separate;  
  
**use** Rela589n\DoctrineEventSourcing\Serializer\Context\DeserializationContext;  
  
**interface** SeparateDeserializer  
{  
 **public function** isPossible(DeserializationContext $context): **bool**;  
  
 **public function** \_\_invoke(DeserializationContext $context): **mixed**;  
}

Клас \Rela589n\DoctrineEventSourcing\Serializer\Separate\SeparateSerializer:

**<?php  
  
declare**(strict\_types=1);  
  
**namespace** Rela589n\DoctrineEventSourcing\Serializer\Separate;  
  
**use** Rela589n\DoctrineEventSourcing\Serializer\Context\SerializationContext;  
  
**interface** SeparateSerializer  
{  
 **public function** isPossible(SerializationContext $context): **bool**;  
  
 **public function** \_\_invoke(SerializationContext $context): **mixed**;  
}

Клас \Rela589n\DoctrineEventSourcing\Serializer\Separate\Typed\DeserializeTyped:

**<?php  
  
declare**(strict\_types=1);  
  
**namespace** Rela589n\DoctrineEventSourcing\Serializer\Separate\Typed;  
  
**use** Doctrine\DBAL\Types\Type;  
**use** Doctrine\ORM\EntityManagerInterface;  
**use** JetBrains\PhpStorm\Immutable;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Context\DeserializationContext;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Separate\SeparateDeserializer;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Util\Converter\ConvertToPHPValue;  
  
*#[Immutable]***final class** DeserializeTyped **implements** SeparateDeserializer  
{  
 **public function** \_\_construct(  
 **private** ConvertToPHPValue **$convertToPHPValue**,  
 **private array $propertiesTypes**,  
 ) {  
 $this->propertiesTypes = array\_map(**static fn**(Type $type) => $type, $this->propertiesTypes);  
 }  
  
 **public static function** from(EntityManagerInterface $manager, **array** $propertiesTypes): **self** {  
 **return new self**(ConvertToPHPValue\Impl::*fromEntityManager*($manager), $propertiesTypes);  
 }  
  
 **public function** isPossible(DeserializationContext $context): **bool** {  
 **return isset**($this->**propertiesTypes**[$context->getFieldName()]);  
 }  
  
 **public function** \_\_invoke(DeserializationContext $context): **mixed** {  
 $fieldName = $context->getFieldName();  
 $serialized = $context->getSerialized();  
 $name = $context->getName();  
  
 **return** ($this->**convertToPHPValue**)($this->**propertiesTypes**[$fieldName], $serialized[$name]);  
 }  
}

Клас \Rela589n\DoctrineEventSourcing\Serializer\Separate\Typed\SerializeTyped:

**<?php  
  
declare**(strict\_types=1);  
  
**namespace** Rela589n\DoctrineEventSourcing\Serializer\Separate\Typed;  
  
**use** Doctrine\DBAL\Types\Type;  
**use** Doctrine\ORM\EntityManagerInterface;  
**use** JetBrains\PhpStorm\Immutable;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Context\SerializationContext;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Separate\SeparateSerializer;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Util\Converter\ConvertToDatabaseValue;  
  
*#[Immutable]***final class** SerializeTyped **implements** SeparateSerializer  
{  
 **public function** \_\_construct(  
 **private** ConvertToDatabaseValue **$convertToDatabaseValue**,  
 **private array $propertiesTypes**,  
 ) {  
 $this->propertiesTypes = array\_map(**static fn**(Type $type) => $type, $this->propertiesTypes);  
 }  
  
 **public static function** from(EntityManagerInterface $manager, **array** $propertiesTypes): **self** {  
 **return new self**(ConvertToDatabaseValue\Impl::*fromEntityManager*($manager), $propertiesTypes);  
 }  
  
 **public function** isPossible(SerializationContext $context): **bool** {  
 **return isset**($this->**propertiesTypes**[$context->getFieldName()]);  
 }  
  
 **public function** \_\_invoke(SerializationContext $context): **mixed** {  
 **return** ($this->**convertToDatabaseValue**)($this->**propertiesTypes**[$context->getFieldName()], $context->getValue());  
 }  
}

Клас \Rela589n\DoctrineEventSourcing\Serializer\Separate\Noop\DeserializeNoop:

**<?php  
  
declare**(strict\_types=1);  
  
**namespace** Rela589n\DoctrineEventSourcing\Serializer\Separate\Noop;  
  
**use** Rela589n\DoctrineEventSourcing\Serializer\Context\DeserializationContext;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Separate\SeparateDeserializer;  
  
**final class** DeserializeNoop **implements** SeparateDeserializer  
{  
 **private static** *$instance* = **null**;  
  
 **private function** \_\_construct() { }  
  
 **public static function** instance(): **self** {  
 **return self**::*$instance* ?? **self**::*$instance* = **new self**();  
 }  
  
 **public function** isPossible(DeserializationContext $context): **bool** {  
 **return isset**($context->getSerialized()[$context->getFieldName()]);  
 }  
  
 **public function** \_\_invoke(DeserializationContext $context): **mixed** {  
 **return** $context->getSerialized()[$context->getFieldName()];  
 }  
}

Клас \Rela589n\DoctrineEventSourcing\Serializer\Separate\Noop\SerializeNoop:

**<?php  
  
declare**(strict\_types=1);  
  
**namespace** Rela589n\DoctrineEventSourcing\Serializer\Separate\Noop;  
  
**use** Rela589n\DoctrineEventSourcing\Serializer\Context\SerializationContext;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Separate\SeparateSerializer;  
  
**final class** SerializeNoop **implements** SeparateSerializer  
{  
 **private static** *$instance* = **null**;  
  
 **private function** \_\_construct() { }  
  
 **public static function** instance(): **self** {  
 **return self**::*$instance* ?? **self**::*$instance* = **new self**();  
 }  
  
 **public function** isPossible(SerializationContext $context): **bool** {  
 **return true**;  
 }  
  
 **public function** \_\_invoke(SerializationContext $context): **mixed** {  
 **return** $context->getValue();  
 }  
}

Клас \Rela589n\DoctrineEventSourcing\Serializer\Separate\Entity\DeserializeEntity:

**<?php  
  
declare**(strict\_types=1);  
  
**namespace** Rela589n\DoctrineEventSourcing\Serializer\Separate\Entity;  
  
**use** Doctrine\Common\Proxy\AbstractProxyFactory **as** ProxyFactory;  
**use** Doctrine\ORM\EntityManagerInterface;  
**use** Doctrine\ORM\UnitOfWork;  
**use** JetBrains\PhpStorm\Immutable;  
**use** Rela589n\DoctrineEventSourcing\Entity\AggregateRoot;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Context\DeserializationContext;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Separate\SeparateDeserializer;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Util\Converter\ConvertToPHPValue;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Util\Types\ResolvePrimaryType;  
  
*#[Immutable]***final class** DeserializeEntity **implements** SeparateDeserializer  
{  
 **public function** \_\_construct(  
 **private** UnitOfWork **$unitOfWork**,  
 **private** *ProxyFactory* **$proxyFactory**,  
 **private** ConvertToPHPValue **$convertToPHPValue**,  
 **private** ResolvePrimaryType **$resolvePrimaryType**,  
 ) {  
 }  
  
 **public static function** from(EntityManagerInterface $manager): **self** {  
 **return new self**(  
 $manager->getUnitOfWork(),  
 $manager->getProxyFactory(),  
 ConvertToPHPValue\Impl::*fromEntityManager*($manager),  
 **new** ResolvePrimaryType\Impl($manager),  
 );  
 }  
  
 **public function** isPossible(DeserializationContext $context): **bool** {  
 **return** *is\_subclass\_of*($context->getType(), AggregateRoot::**class**);  
 }  
  
 **public function** \_\_invoke(DeserializationContext $context): AggregateRoot  
 {  
 */\*\** ***@var*** *string|AggregateRoot $type \*/* $name = $context->getName();  
 $type = $context->getType();  
 $serialized = $context->getSerialized();  
  
 $primary = $this->getPrimary($type, $serialized[$name]);  
 $identifier = [$type::*getPrimaryName*() => $primary];  
  
 **return** $this->**unitOfWork**->tryGetById($identifier, $type)  
 ?: $this->**proxyFactory**->getProxy($type, $identifier);  
 }  
  
 **protected function** getPrimary(**string** $type, **mixed** $serialized)  
 {  
 $primaryType = ($this->**resolvePrimaryType**)($type);  
  
 **return** ($this->**convertToPHPValue**)($primaryType, $serialized);  
 }  
}

Клас \Rela589n\DoctrineEventSourcing\Serializer\Separate\Entity\SerializeEntity:

**<?php  
  
declare**(strict\_types=1);  
  
**namespace** Rela589n\DoctrineEventSourcing\Serializer\Separate\Entity;  
  
**use** Doctrine\ORM\EntityManagerInterface;  
**use** JetBrains\PhpStorm\Immutable;  
**use** Rela589n\DoctrineEventSourcing\Entity\AggregateRoot;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Context\SerializationContext;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Separate\SeparateSerializer;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Util\Converter\ConvertToDatabaseValue;  
  
*#[Immutable]***final class** SerializeEntity **implements** SeparateSerializer  
{  
 **public function** \_\_construct(  
 **private** EntityManagerInterface **$em**,  
 **private** ConvertToDatabaseValue **$convertToDatabaseValue**,  
 ) {  
 }  
  
 **public static function** from(EntityManagerInterface $manager): **self** {  
 **return new self**($manager, ConvertToDatabaseValue\Impl::*fromEntityManager*($manager));  
 }  
  
 **public function** isPossible(SerializationContext $context): **bool** {  
 **return** $context->getValue() **instanceof** AggregateRoot;  
 }  
  
 **public function** \_\_invoke(SerializationContext $context): **mixed** {  
 */\*\** ***@var*** *AggregateRoot $value \*/* $value = $context->getValue();  
  
 $typeName = $this->**em**->getClassMetadata($value::**class**)  
 ->getFieldMapping($value::*getPrimaryName*())[**'type'**];  
  
 **return** ($this->**convertToDatabaseValue**)($typeName, $value->getPrimary());  
 }  
}

Клас \Rela589n\DoctrineEventSourcing\Serializer\Separate\Embedded\DeserializeEmbedded:

**<?php  
  
declare**(strict\_types=1);  
  
**namespace** Rela589n\DoctrineEventSourcing\Serializer\Separate\Embedded;  
  
**use** Doctrine\ORM\EntityManagerInterface;  
**use** Doctrine\ORM\Mapping\ClassMetadataInfo **as** ClassMetadata;  
**use** JetBrains\PhpStorm\Immutable;  
**use** Rela589n\DoctrineEventSourcing\Entity\AggregateRoot;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Context\DeserializationContext;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Separate\SeparateDeserializer;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Util\Converter\ConvertToPHPValue;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Util\Types\TypeIsEmbedded;  
  
*#[Immutable]***final class** DeserializeEmbedded **implements** SeparateDeserializer  
{  
 **public function** \_\_construct(  
 **private** EntityManagerInterface **$manager**,  
 **private** *ClassMetadata* **$entityMetadata**,  
 **private** TypeIsEmbedded **$typeIsEmbedded**,  
 **private** ConvertToPHPValue **$convertToPHPValue**,  
 ) {  
 }  
  
 **public static function** from(EntityManagerInterface $manager, AggregateRoot $entity): **self** {  
 **return new self**(  
 $manager,  
 $manager->getClassMetadata($entity::**class**),  
 **new** TypeIsEmbedded\Impl($manager),  
 ConvertToPHPValue\Impl::*fromEntityManager*($manager),  
 );  
 }  
  
 **public function** isPossible(DeserializationContext $context): **bool** {  
 **return** ($this->**typeIsEmbedded**)($context->getType());  
 }  
  
 **public function** \_\_invoke(DeserializationContext $context): **mixed** {  
 $name = $context->getName();  
 $type = $context->getType();  
 $serialized = $context->getSerialized();  
  
 $valueMetadata = $this->**manager**->getClassMetadata($type);  
 $instance = $valueMetadata->**reflClass**->newInstanceWithoutConstructor();  
  
 **foreach** ($this->**entityMetadata**->**fieldMappings as** $data) {  
 **if** (($data[**'originalClass'**] ?? **null**) !== $type) {  
 **continue**;  
 }  
  
 [  
 **'type'** => $dbalType,  
 **'columnName'** => $columnName,  
 **'originalField'** => $originalField,  
 ] = $data;  
  
 $reflectedValue = ($this->**convertToPHPValue**)($dbalType, $serialized[$name][$columnName]);  
  
 $valueMetadata->**reflFields**[$originalField]  
 ->setValue($instance, $reflectedValue);  
 }  
  
 **return** $instance;  
 }  
}

Клас \Rela589n\DoctrineEventSourcing\Serializer\Separate\Embedded\SerializeEmbedded:

**<?php  
  
declare**(strict\_types=1);  
  
**namespace** Rela589n\DoctrineEventSourcing\Serializer\Separate\Embedded;  
  
**use** Doctrine\ORM\EntityManagerInterface;  
**use** Doctrine\ORM\Mapping\ClassMetadataInfo **as** ClassMetadata;  
**use** JetBrains\PhpStorm\Immutable;  
**use** Rela589n\DoctrineEventSourcing\Entity\AggregateRoot;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Context\SerializationContext;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Separate\SeparateSerializer;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Util\Converter\ConvertToDatabaseValue;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Util\Types\TypeIsEmbedded;  
  
*#[Immutable]***final class** SerializeEmbedded **implements** SeparateSerializer  
{  
 **public function** \_\_construct(  
 **private** EntityManagerInterface **$manager**,  
 **private** *ClassMetadata* **$entityMetadata**,  
 **private** TypeIsEmbedded **$typeIsEmbedded**,  
 **private** ConvertToDatabaseValue **$convertToDatabaseValue**,  
 ) {  
 }  
  
 **public static function** from(EntityManagerInterface $manager, AggregateRoot $entity): **self** {  
 **return new self**(  
 $manager,  
 $manager->getClassMetadata($entity::**class**),  
 **new** TypeIsEmbedded\Impl($manager),  
 ConvertToDatabaseValue\Impl::*fromEntityManager*($manager),  
 );  
 }  
  
 **public function** isPossible(SerializationContext $context): **bool** {  
 $value = $context->getValue();  
  
 **return** *is\_object*($value)  
 && ($this->**typeIsEmbedded**)($value::**class**);  
 }  
  
 **public function** \_\_invoke(SerializationContext $context): **array** {  
 $value = $context->getValue();  
  
 $valueMeta = $this->**manager**->getClassMetadata($value::**class**);  
  
 $serialized = [];  
  
 **foreach** ($this->**entityMetadata**->**fieldMappings as** $data) {  
 **if** (($data[**'originalClass'**] ?? **null**) !== $value::**class**) {  
 **continue**;  
 }  
  
 [  
 **'type'** => $dbalType,  
 **'columnName'** => $columnName,  
 **'originalField'** => $originalField,  
 ] = $data;  
  
 $reflectedValue = $valueMeta  
 ->**reflFields**[$originalField]  
 ->getValue($value);  
  
 $serialized[$columnName] = ($this->**convertToDatabaseValue**)($dbalType, $reflectedValue);  
 }  
  
 **return** $serialized;  
 }  
}

Клас \Rela589n\DoctrineEventSourcing\Serializer\Separate\Castable\DeserializeCastable:

**<?php  
  
declare**(strict\_types=1);  
  
**namespace** Rela589n\DoctrineEventSourcing\Serializer\Separate\Castable;  
  
**use** JetBrains\PhpStorm\Immutable;  
**use** Rela589n\DoctrineEventSourcing\Entity\AggregateRoot;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Context\DeserializationContext;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Separate\Castable\Contract\Castable;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Separate\Castable\Contract\CastsAttributes;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Separate\SeparateDeserializer;  
  
*#[Immutable]***final class** DeserializeCastable **implements** SeparateDeserializer  
{  
 **public function** \_\_construct(  
 **private** AggregateRoot **$entity**,  
 **private array $castArguments**,  
 ) {  
 }  
  
 **public static function** from(AggregateRoot $entity, **array** $castArguments = []): **self** {  
 **return new self**($entity, $castArguments);  
 }  
  
 **public function** isPossible(DeserializationContext $context): **bool** {  
 **return** *is\_subclass\_of*($context->getType(), Castable::**class**);  
 }  
  
 **public function** \_\_invoke(DeserializationContext $context): **mixed** {  
 */\*\** ***@var*** *string|Castable $type \*/* $fieldName = $context->getFieldName();  
 $name = $context->getName();  
 $type = $context->getType();  
 $serialized = $context->getSerialized();  
  
 */\*\** ***@var*** *CastsAttributes $caster \*/* $caster = $type::*castUsing*($this->**castArguments**[$fieldName] ?? $this->**castArguments**[$type] ?? []);  
  
 $attributes = $serialized[$name];  
  
 **return** $caster->get($this->**entity**, $fieldName, $attributes[$name] ?? **null**, $attributes);  
 }  
}

Клас \Rela589n\DoctrineEventSourcing\Serializer\Separate\Castable\SerializeCastable:

**<?php  
  
declare**(strict\_types=1);  
  
**namespace** Rela589n\DoctrineEventSourcing\Serializer\Separate\Castable;  
  
**use** JetBrains\PhpStorm\Immutable;  
**use** Rela589n\DoctrineEventSourcing\Entity\AggregateRoot;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Context\SerializationContext;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Separate\Castable\Contract\Castable;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Separate\Castable\Contract\CastsAttributes;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Separate\SeparateSerializer;  
  
*#[Immutable]***final class** SerializeCastable **implements** SeparateSerializer  
{  
 **public function** \_\_construct(  
 **private** AggregateRoot **$entity**,  
 **private array $castArguments** = [],  
 ) {  
 }  
  
 **public static function** from(AggregateRoot $entity, **array** $castArguments = []): **self** {  
 **return new self**($entity, $castArguments);  
 }  
  
 **public function** isPossible(SerializationContext $context): **bool** {  
 **return** $context->getValue() **instanceof** Castable;  
 }  
  
 **public function** \_\_invoke(SerializationContext $context): **mixed** {  
 $fieldName = $context->getFieldName();  
 $name = $context->getName();  
 $value = $context->getValue();  
 $attributes = $context->getAttributes();  
  
 */\*\** ***@var*** *CastsAttributes $caster \*/* $caster = $value::*castUsing*($this->**castArguments**[$fieldName] ?? $this->**castArguments**[$value::**class**] ?? []);  
  
 **return** $caster->set($this->**entity**, $fieldName, $value, $attributes[$name] ?? []);  
 }  
}

Клас \Rela589n\DoctrineEventSourcing\Serializer\Separate\Castable\Contract\Castable:

**<?php  
  
declare**(strict\_types=1);  
  
**namespace** Rela589n\DoctrineEventSourcing\Serializer\Separate\Castable\Contract;  
  
**interface** Castable  
{  
 */\*\** ***@return*** *CastsAttributes \*/* **public static function** castUsing(**array** $arguments);  
}

Клас \Rela589n\DoctrineEventSourcing\Serializer\Separate\Castable\Contract\CastsAttributes:

**<?php  
  
namespace** Rela589n\DoctrineEventSourcing\Serializer\Separate\Castable\Contract;  
  
**use** Rela589n\DoctrineEventSourcing\Entity\AggregateRoot;  
  
**interface** CastsAttributes  
{  
 */\*\*  
 \* Transform the attribute from the underlying entity values.  
 \*  
 \** ***@param*** *AggregateRoot $model  
 \** ***@param*** *string $key  
 \** ***@param*** *mixed $value  
 \** ***@param*** *array $attributes  
 \*  
 \** ***@return*** *mixed  
 \*/* **public function** get($model, **string** $key, $value, **array** $attributes);  
  
 */\*\*  
 \* Transform the attribute to its underlying entity values.  
 \*  
 \** ***@param*** *AggregateRoot $model  
 \** ***@param*** *string $key  
 \** ***@param*** *mixed $value  
 \** ***@param*** *array $attributes  
 \*  
 \** ***@return*** *array  
 \*/* **public function** set($model, **string** $key, $value, **array** $attributes);  
}

Клас \Rela589n\DoctrineEventSourcing\Serializer\Pipeline\DeserializationContextPipe:

**<?php  
  
declare**(strict\_types=1);  
  
**namespace** Rela589n\DoctrineEventSourcing\Serializer\Pipeline;  
  
**use** JetBrains\PhpStorm\Immutable;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Context\DeserializationContext;  
  
*#[Immutable]***interface** DeserializationContextPipe  
{  
 **public function** \_\_invoke(DeserializationContext $context): DeserializationContext;  
}

Клас \Rela589n\DoctrineEventSourcing\Serializer\Pipeline\SerializationContextPipe:

**<?php  
  
declare**(strict\_types=1);  
  
**namespace** Rela589n\DoctrineEventSourcing\Serializer\Pipeline;  
  
**use** JetBrains\PhpStorm\Immutable;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Context\SerializationContext;  
  
*#[Immutable]***interface** SerializationContextPipe  
{  
 **public function** \_\_invoke(SerializationContext $context): SerializationContext;  
}

Клас \Rela589n\DoctrineEventSourcing\Serializer\Pipeline\Pipes\SubstituteAnnotatedDeserializeName:

**<?php  
  
declare**(strict\_types=1);  
  
**namespace** Rela589n\DoctrineEventSourcing\Serializer\Pipeline\Pipes;  
  
**use** JetBrains\PhpStorm\Immutable;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Context\DeserializationContext;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Pipeline\DeserializationContextPipe;  
  
*#[Immutable]***final class** SubstituteAnnotatedDeserializeName **implements** DeserializationContextPipe  
{  
 **public function** \_\_construct(**private array $namesMeta**) { }  
  
 **public function** \_\_invoke(DeserializationContext $context): DeserializationContext  
 {  
 **if** (**empty**($this->**namesMeta**[$context->getFieldName()])) {  
 **return** $context;  
 }  
  
 **return** $context->withName($this->**namesMeta**[$context->getFieldName()]);  
 }  
}

Клас \Rela589n\DoctrineEventSourcing\Serializer\Pipeline\Pipes\SubstituteAnnotatedSerializeName:

**<?php  
  
declare**(strict\_types=1);  
  
**namespace** Rela589n\DoctrineEventSourcing\Serializer\Pipeline\Pipes;  
  
**use** JetBrains\PhpStorm\Immutable;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Context\SerializationContext;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Pipeline\SerializationContextPipe;  
  
*#[Immutable]***final class** SubstituteAnnotatedSerializeName **implements** SerializationContextPipe  
{  
 **public function** \_\_construct(**private array $namesMeta**) { }  
  
 **public function** \_\_invoke(SerializationContext $context): SerializationContext  
 {  
 **if** (**empty**($this->**namesMeta**[$context->getFieldName()])) {  
 **return** $context;  
 }  
  
 **return** $context->withName($this->**namesMeta**[$context->getFieldName()]);  
 }  
}

Клас \Rela589n\DoctrineEventSourcing\Serializer\Context\DeserializationContext:

**<?php  
  
declare**(strict\_types=1);  
  
**namespace** Rela589n\DoctrineEventSourcing\Serializer\Context;  
  
**use** JetBrains\PhpStorm\Immutable;  
  
*#[Immutable(Immutable::PRIVATE\_WRITE\_SCOPE)]***class** DeserializationContext  
{  
 **protected string $fieldName**;  
 **protected string $type**;  
 **protected array $serialized**;  
 **protected** ?**string $name** = **null**;  
  
 **private function** \_\_construct()  
 {  
 }  
  
 **public static function** make(): **static** {  
 **return new static**();  
 }  
  
 **public function** withFieldName(**string** $fieldName): **static** {  
 $static = **clone** $this;  
 $static->**fieldName** = $fieldName;  
 **return** $static;  
 }  
  
 **public function** withType(**string** $type): **static** {  
 $static = **clone** $this;  
 $static->**type** = $type;  
 **return** $static;  
 }  
  
 **public function** withSerialized(**array** $serialized): **static** {  
 $static = **clone** $this;  
 $static->**serialized** = $serialized;  
 **return** $static;  
 }  
  
 **public function** withName(**string** $name): **static** {  
 $static = **clone** $this;  
 $static->**name** = $name;  
 **return** $static;  
 }  
  
 **public function** getName(): **string** {  
 **return** $this->**name** ?? $this->**fieldName**;  
 }  
  
 **public function** getFieldName(): **string** {  
 **return** $this->**fieldName**;  
 }  
  
 **public function** getType(): **string** {  
 **return** $this->**type**;  
 }  
  
 **public function** getSerialized(): **array** {  
 **return** $this->**serialized**;  
 }  
}

Клас \Rela589n\DoctrineEventSourcing\Serializer\Context\SerializationContext:

**<?php  
  
declare**(strict\_types=1);  
  
**namespace** Rela589n\DoctrineEventSourcing\Serializer\Context;  
  
**use** JetBrains\PhpStorm\Immutable;  
  
*#[Immutable(Immutable::PRIVATE\_WRITE\_SCOPE)]***class** SerializationContext  
{  
 **protected string $fieldName**;  
 **protected mixed $value**;  
 **protected array $attributes**;  
 **protected** ?**string $name** = **null**;  
  
 **public static function** make(): **static** {  
 **return new static**();  
 }  
  
 **public function** withFieldName(**string** $fieldName): **static** {  
 $static = **clone** $this;  
 $static->**fieldName** = $fieldName;  
 **return** $static;  
 }  
  
 **public function** withValue(**mixed** $value): **static** {  
 $static = **clone** $this;  
 $static->**value** = $value;  
 **return** $static;  
 }  
  
 **public function** withAttributes(**array** $attributes): **static** {  
 $static = **clone** $this;  
 $static->**attributes** = $attributes;  
 **return** $static;  
 }  
  
 **public function** withName(**string** $name): **static** {  
 $static = **clone** $this;  
 $static->**name** = $name;  
 **return** $static;  
 }  
  
 **public function** getName(): **string** {  
 **return** $this->**name** ?? $this->**fieldName**;  
 }  
  
 **public function** getFieldName(): **string** {  
 **return** $this->**fieldName**;  
 }  
  
 **public function** getValue(): **mixed** {  
 **return** $this->**value**;  
 }  
  
 **public function** getAttributes(): **array** {  
 **return** $this->**attributes**;  
 }  
}

Клас \Rela589n\DoctrineEventSourcing\Serializer\Composed\ComposedDeserializer:

**<?php  
  
declare**(strict\_types=1);  
  
**namespace** Rela589n\DoctrineEventSourcing\Serializer\Composed;  
  
**use** Closure;  
**use** JetBrains\PhpStorm\Immutable;  
**use** LogicException;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Context\DeserializationContext;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Pipeline\DeserializationContextPipe;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Separate\SeparateDeserializer;  
  
*#[Immutable]***final class** ComposedDeserializer  
{  
 */\*\** ***@var*** *Closure(DeserializationContext):iterable<DeserializationContextPipe> \*/* **private** Closure **$pipes**;  
  
 */\*\** ***@var*** *Closure(DeserializationContext):iterable<SeparateDeserializer> \*/* **private** Closure **$deserializers**;  
  
 **public function** \_\_construct(Closure $pipes, Closure $deserializers)  
 {  
 $this->**pipes** = $pipes;  
 $this->**deserializers** = $deserializers;  
 }  
  
 **public function** \_\_invoke(DeserializationContext $context): **mixed** {  
 */\*\** ***@var*** *DeserializationContextPipe $pipe \*/* **foreach** (($this->**pipes**)($context) **as** $pipe) {  
 $context = $pipe($context);  
 }  
  
 */\*\** ***@var*** *SeparateDeserializer $deserialize \*/* **foreach** (($this->**deserializers**)($context) **as** $deserialize) {  
 **if** ($deserialize->isPossible($context)) {  
 **return** $deserialize($context);  
 }  
 }  
  
 **throw new** LogicException(**"No deserializer matches context for '**{$context->getName()}**' property"**);  
 }  
}

Клас \Rela589n\DoctrineEventSourcing\Serializer\Composed\ComposedSerializer:

**<?php  
  
declare**(strict\_types=1);  
  
**namespace** Rela589n\DoctrineEventSourcing\Serializer\Composed;  
  
**use** Closure;  
**use** JetBrains\PhpStorm\ArrayShape;  
**use** JetBrains\PhpStorm\Immutable;  
**use** LogicException;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Context\SerializationContext;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Pipeline\SerializationContextPipe;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Separate\SeparateSerializer;  
  
*#[Immutable]***final class** ComposedSerializer  
{  
 */\*\** ***@var*** *Closure(SerializationContext):iterable<SerializationContextPipe> \*/* **private** Closure **$pipes**;  
  
 */\*\** ***@var*** *Closure(SerializationContext):iterable<SeparateSerializer> \*/* **private** Closure **$serializers**;  
  
 **public function** \_\_construct(Closure $pipes, Closure $serializers)  
 {  
 $this->**pipes** = $pipes;  
 $this->**serializers** = $serializers;  
 }  
  
 *#[ArrayShape(['name' => "string", 'value' => "mixed"])]* **public function** \_\_invoke(SerializationContext $context): **array** {  
 */\*\** ***@var*** *SerializationContextPipe $pipe \*/* **foreach** (($this->**pipes**)($context) **as** $pipe) {  
 $context = $pipe($context);  
 }  
  
 */\*\** ***@var*** *SeparateSerializer $serialize \*/* **foreach** (($this->**serializers**)($context) **as** $serialize) {  
 **if** ($serialize->isPossible($context)) {  
 **return** [  
 **'name'** => $context->getName(),  
 **'value'** => $serialize($context)  
 ];  
 }  
 }  
  
 **throw new** LogicException(**"No serializer matches context for '**{$context->getFieldName()}**' property"**);  
 }  
}

Клас \Rela589n\DoctrineEventSourcing\Serializer\Composed\ComposedSerializer\Factory:

**<?php  
  
declare**(strict\_types=1);  
  
**namespace** Rela589n\DoctrineEventSourcing\Serializer\Composed\ComposedSerializer;  
  
**use** Rela589n\DoctrineEventSourcing\Serializer\Composed\ComposedSerializer;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Context\SerializationContext;  
  
**abstract class** Factory  
{  
 **final public function** make(): ComposedSerializer  
 {  
 **return new** ComposedSerializer(  
 **fn**($c) => $this->pipes($c),  
 **fn**($c) => $this->serializers($c),  
 );  
 }  
  
 **abstract protected function** pipes(SerializationContext $context): **iterable**;  
  
 **abstract protected function** serializers(SerializationContext $context): **iterable**;  
}

Клас \Rela589n\DoctrineEventSourcing\Serializer\Composed\ComposedSerializer\Factory\Context:

**<?php  
  
declare**(strict\_types=1);  
  
**namespace** Rela589n\DoctrineEventSourcing\Serializer\Composed\ComposedSerializer\Factory;  
  
**use** Doctrine\ORM\EntityManagerInterface;  
**use** Rela589n\DoctrineEventSourcing\Entity\AggregateRoot;  
  
**final class** Context  
{  
 **private** EntityManagerInterface **$manager**;  
 **private** AggregateRoot **$entity**;  
 **private array $propertiesMeta**;  
 **private array $castArgumentsMap**;  
  
 **public static function** make(): **self** {  
 **return new self**();  
 }  
  
 **public function** withEntityManager(EntityManagerInterface $manager): **self** {  
 $this->**manager** = $manager;  
  
 **return** $this;  
 }  
  
 **public function** withEntity(AggregateRoot $entity): **self** {  
 $this->**entity** = $entity;  
  
 **return** $this;  
 }  
  
 **public function** withPropertiesMeta(**array** $propertiesMeta): **self** {  
 $this->**propertiesMeta** = $propertiesMeta;  
  
 **return** $this;  
 }  
  
 **public function** withCastArgumentsMap(**array** $castArgumentsMap): **self** {  
 $this->**castArgumentsMap** = $castArgumentsMap;  
  
 **return** $this;  
 }  
  
 **public function** getManager(): EntityManagerInterface  
 {  
 **return** $this->**manager**;  
 }  
  
 **public function** getEntity(): AggregateRoot  
 {  
 **return** $this->**entity**;  
 }  
  
 **public function** getPropertiesMeta(): **array** {  
 **return** $this->**propertiesMeta**;  
 }  
  
 **public function** getCastArgumentsMap(): **array** {  
 **return** $this->**castArgumentsMap**;  
 }  
}

Клас \Rela589n\DoctrineEventSourcing\Serializer\Composed\ComposedSerializer\Factory\Impl:

**<?php  
  
declare**(strict\_types=1);  
  
**namespace** Rela589n\DoctrineEventSourcing\Serializer\Composed\ComposedSerializer\Factory;  
  
**use** Doctrine\DBAL\Types\Type;  
**use** Doctrine\ORM\EntityManagerInterface;  
**use** Generator;  
**use** JetBrains\PhpStorm\Immutable;  
**use** Rela589n\DoctrineEventSourcing\Entity\AggregateRoot;  
**use** Rela589n\DoctrineEventSourcing\Event\Annotations\SerializeAs;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Composed\ComposedSerializer\Factory;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Context\SerializationContext;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Pipeline\Pipes\SubstituteAnnotatedSerializeName;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Separate\Castable\SerializeCastable;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Separate\Embedded\SerializeEmbedded;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Separate\Entity\SerializeEntity;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Separate\Noop\SerializeNoop;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Separate\Typed\SerializeTyped;  
  
*#[Immutable]***final class** Impl **extends** Factory  
{  
 */\*\** ***@var*** *Type[] \*/* **private array $typesMeta**;  
 **private array $namesMeta**;  
  
 **private function** \_\_construct(  
 **private** EntityManagerInterface **$manager**,  
 **private** AggregateRoot **$entity**,  
 **private array $propertiesMeta**,  
 **private array $castArgumentsMap**,  
 ) {  
 $this->propertiesMeta = array\_map(**static fn**(SerializeAs $as) => $as, $this->propertiesMeta);  
 $this->typesMeta = array\_filter(array\_map(**static fn**(SerializeAs $as) => $as->getType(), $this->propertiesMeta));  
 $this->namesMeta = array\_filter(array\_map(**static fn**(SerializeAs $as) => $as->getName(), $this->propertiesMeta));  
 }  
  
 **public static function** fromContext(Context $context): **self** {  
 **return new self**(  
 $context->getManager(),  
 $context->getEntity(),  
 $context->getPropertiesMeta(),  
 $context->getCastArgumentsMap(),  
 );  
 }  
  
 **protected function** pipes(SerializationContext $context): Generator  
 {  
 **yield new** SubstituteAnnotatedSerializeName($this->**namesMeta**);  
 }  
  
 **protected function** serializers(SerializationContext $context): Generator  
 {  
 **yield** SerializeTyped::*from*($this->**manager**, $this->**typesMeta**);  
 **yield** SerializeEntity::*from*($this->**manager**);  
 **yield** SerializeCastable::*from*($this->**entity**, $this->**castArgumentsMap**);  
 **yield** SerializeEmbedded::*from*($this->**manager**, $this->**entity**);  
 **yield** SerializeNoop::*instance*();  
 }  
}

Клас \Rela589n\DoctrineEventSourcing\Serializer\Composed\ComposedDeserializer\Factory:

**<?php  
  
declare**(strict\_types=1);  
  
**namespace** Rela589n\DoctrineEventSourcing\Serializer\Composed\ComposedDeserializer;  
  
**use** Rela589n\DoctrineEventSourcing\Serializer\Composed\ComposedDeserializer;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Context\DeserializationContext;  
  
**abstract class** Factory  
{  
 **final public function** make(): ComposedDeserializer  
 {  
 **return new** ComposedDeserializer(  
 **fn**($c) => $this->pipes($c),  
 **fn**($c) => $this->deserializers($c),  
 );  
 }  
  
 **abstract protected function** pipes(DeserializationContext $context): **iterable**;  
  
 **abstract protected function** deserializers(DeserializationContext $context): **iterable**;  
}

Клас \Rela589n\DoctrineEventSourcing\Serializer\Composed\ComposedDeserializer\Factory\Context:

**<?php  
  
declare**(strict\_types=1);  
  
**namespace** Rela589n\DoctrineEventSourcing\Serializer\Composed\ComposedDeserializer\Factory;  
  
**use** Doctrine\ORM\EntityManagerInterface;  
**use** Rela589n\DoctrineEventSourcing\Entity\AggregateRoot;  
  
**final class** Context  
{  
 **private** EntityManagerInterface **$manager**;  
 **private** AggregateRoot **$entity**;  
 **private array $propertiesMeta**;  
 **private array $castArgumentsMap**;  
  
 **public static function** make(): **self** {  
 **return new self**();  
 }  
  
 **public function** withEntityManager(EntityManagerInterface $manager): **self** {  
 $this->**manager** = $manager;  
  
 **return** $this;  
 }  
  
 **public function** withEntity(AggregateRoot $entity): **self** {  
 $this->**entity** = $entity;  
  
 **return** $this;  
 }  
  
 **public function** withPropertiesMeta(**array** $propertiesMeta): **self** {  
 $this->**propertiesMeta** = $propertiesMeta;  
  
 **return** $this;  
 }  
  
 **public function** withCastArgumentsMap(**array** $castArgumentsMap): **self** {  
 $this->**castArgumentsMap** = $castArgumentsMap;  
  
 **return** $this;  
 }  
  
 **public function** getManager(): EntityManagerInterface  
 {  
 **return** $this->**manager**;  
 }  
  
 **public function** getEntity(): AggregateRoot  
 {  
 **return** $this->**entity**;  
 }  
  
 **public function** getPropertiesMeta(): **array** {  
 **return** $this->**propertiesMeta**;  
 }  
  
 **public function** getCastArgumentsMap(): **array** {  
 **return** $this->**castArgumentsMap**;  
 }  
}

Клас \Rela589n\DoctrineEventSourcing\Serializer\Composed\ComposedDeserializer\Factory\Impl:

**<?php  
  
declare**(strict\_types=1);  
  
**namespace** Rela589n\DoctrineEventSourcing\Serializer\Composed\ComposedDeserializer\Factory;  
  
**use** Doctrine\DBAL\Types\Type;  
**use** Doctrine\ORM\EntityManagerInterface;  
**use** Generator;  
**use** JetBrains\PhpStorm\Immutable;  
**use** Rela589n\DoctrineEventSourcing\Entity\AggregateRoot;  
**use** Rela589n\DoctrineEventSourcing\Event\Annotations\SerializeAs;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Composed\ComposedDeserializer\Factory;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Context\DeserializationContext;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Pipeline\Pipes\SubstituteAnnotatedDeserializeName;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Separate\Castable\DeserializeCastable;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Separate\Embedded\DeserializeEmbedded;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Separate\Entity\DeserializeEntity;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Separate\Noop\DeserializeNoop;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Separate\SeparateDeserializer;  
**use** Rela589n\DoctrineEventSourcing\Serializer\Separate\Typed\DeserializeTyped;  
  
*#[Immutable]***final class** Impl **extends** Factory  
{  
 */\*\** ***@var*** *Type[] \*/* **private array $typesMeta**;  
 **private array $namesMeta**;  
  
 **private function** \_\_construct(  
 **private** EntityManagerInterface **$manager**,  
 **private** AggregateRoot **$entity**,  
 **private array $propertiesMeta**,  
 **private array $castArgumentsMap**,  
 ) {  
 $this->propertiesMeta = array\_map(**static fn**(SerializeAs $as) => $as, $this->propertiesMeta);  
 $this->typesMeta = array\_filter(array\_map(**static fn**(SerializeAs $as) => $as->getType(), $this->propertiesMeta));  
 $this->namesMeta = array\_filter(array\_map(**static fn**(SerializeAs $as) => $as->getName(), $this->propertiesMeta));  
 }  
  
 **public static function** fromContext(Context $context): **self** {  
 **return new self**(  
 $context->getManager(),  
 $context->getEntity(),  
 $context->getPropertiesMeta(),  
 $context->getCastArgumentsMap(),  
 );  
 }  
  
 */\*\** ***@return*** *iterable<DeserializationContext> \*/* **protected function** pipes(DeserializationContext $context): Generator  
 {  
 **yield new** SubstituteAnnotatedDeserializeName($this->**namesMeta**);  
 }  
  
 */\*\** ***@return*** *iterable<SeparateDeserializer> \*/* **protected function** deserializers(DeserializationContext $context): Generator  
 {  
 **yield** DeserializeTyped::*from*($this->**manager**, $this->**typesMeta**);  
 **yield** DeserializeEntity::*from*($this->**manager**);  
 **yield** DeserializeCastable::*from*($this->**entity**, $this->**castArgumentsMap**);  
 **yield** DeserializeEmbedded::*from*($this->**manager**, $this->**entity**);  
 **yield** DeserializeNoop::*instance*();  
 }  
}

Висновки

Переддипломна практика корисна для студентів тим, що кожен студент може закріпити і проявити на практиці в зовсім новій та незнайомій ситуації свої знання. Практика виконання реальних завдань є найважливішою частиною навчального процесу для розробників програмного забезпечення. Завдяки їй, студенти навчаються використовувати набір вивченої інформації для вирішення реальних завдань.

Практика дає основу для подальшої професійної діяльності, так як вона передбачає своєю основною метою закріплення отриманих в процесі навчання знань, набуття практичних навичок вирішення проблем. Завдяки їй студенти отримують початковий досвід роботи в сфері інформаційних технологій, що неодмінно веде до підвищення рівня професійної підготовки.

Під час проходження практики було проведено ознайомлення з режимом роботи підприємства, використовуваними технологіями, способом взаємодії між працівниками, основними виконуваними завданнями та загальним внеском результатів виконання завдань на проект зокрема та на компанію в цілому. На практиці була змога спробувати себе в якості розробника програмного забезпечення, архітектора інфраструктури коду та бази даних для нового проекту. Закріпив свої знання в розробці серверної частини веб-додатків, освоїв основні принципи архітектури декомпонованих систем, дізнався деякі тонкощі роботи з різними системами баз даних, їх сильними сторонами та певними обмеженнями.

Отриманий у результаті практики досвід, дає змогу кожному студенту в майбутньому краще адаптуватися до певного виробничого середовища, легко влитись в новий колектив, зменшити стрес від зміни обстановки й налагодити свій робочий розпорядок для отримання найефективніших результатів. В результаті чого добре себе зарекомендувати як хорошого спеціаліста і отримати більше шансів для отримання вакантного місця.

Список використаних джерел

* 1. Роберт Мартин. Чистый код. создание, анализ и рефакторинг
  2. Роберт Мартин. Чистая архитектура. Искусство разработки программного обеспечения
  3. https://te.dsp.gov.ua/vplyv-pratsi-ta-vidpochynku-na-organizm-lyudyny/ – стаття про вплив режиму праці та відпочинку на організм людини
  4. <https://www.php.net/docs.php/>- документація php
  5. https://laravel.com/docs/ - документація фреймворку Laravel
  6. <https://refactoring.guru/> - сайт зі схемами для рефакторингу
  7. <https://proglib.io/> - сайт з корисною інформацією по розробці
  8. <https://circleci.com/docs/2.0/> - документація circleci