|  |
| --- |
| **Health & Social Care Information Centre** |
|  |
| **CIAO the Book**  **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**  **Software to Accelerate, Commoditise and Evolve  connecting care ICT systems together** |

Mike Kelly Adam Hatherly

Richard Dobson Iestyn Evans

Table of Contents

[Copyright 7](#_Toc437954604)

[Foreword 8](#_Toc437954605)

[Preface 9](#_Toc437954606)

[About This Book 9](#_Toc437954607)

[Who Should Read This Book 9](#_Toc437954608)

[Navigating This Book 9](#_Toc437954609)

[Conventions Used in This Book 10](#_Toc437954610)

[Downloads 10](#_Toc437954611)

[Further Reading 11](#_Toc437954612)

[Acknowledgments 11](#_Toc437954613)

[About the Cover Illustration 11](#_Toc437954614)

[Part 1: Understanding CIAO 12](#_Toc437954615)

[1. Introduction 12](#_Toc437954616)

[2. Architecture 14](#_Toc437954617)

[2.1. Microservices 14](#_Toc437954618)

[2.2. Containers 14](#_Toc437954619)

[2.3. Care Integration Patterns 14](#_Toc437954620)

[2.4. Applications 14](#_Toc437954621)

[2.5. Clouds 14](#_Toc437954622)

[2.6. Product and Technology Stack 15](#_Toc437954623)

[2.6.1. Linux 15](#_Toc437954624)

[2.6.2. Docker 16](#_Toc437954625)

[2.6.3. etcd 17](#_Toc437954626)

[2.6.4. ZooKeeper 21](#_Toc437954627)

[2.6.5. ActiveMQ 23](#_Toc437954628)

[2.6.6. Nagios 26](#_Toc437954629)

[2.6.7. Logspout 29](#_Toc437954630)

[2.6.8. Logstash 30](#_Toc437954631)

[2.6.9. Elastic 30](#_Toc437954632)

[2.6.10. Kibana 30](#_Toc437954633)

[2.6.11. File Sharing 31](#_Toc437954634)

[2.6.12. Port Map 31](#_Toc437954635)

[2.7. CIAO Conventions 31](#_Toc437954636)

[2.7.1. CIP Name 31](#_Toc437954637)

[2.7.2. CIP Directory Structure 32](#_Toc437954638)

[2.7.3. CIP Log Record Structure 34](#_Toc437954639)

[2.7.4. Configuration Management 35](#_Toc437954640)

[Part 2: Care Integration Pattern Catalogue 37](#_Toc437954641)

[3. ciao-docs-parser 37](#_Toc437954642)

[3.1. Description 37](#_Toc437954643)

[3.2. Download 38](#_Toc437954644)

[3.3. Configuration 39](#_Toc437954645)

[3.3.1. Spring XML 39](#_Toc437954646)

[3.3.2. CIAO Properties 40](#_Toc437954647)

[3.4. Log Events 43](#_Toc437954648)

[3.5. Technical Narrative 43](#_Toc437954649)

[3.5.1. Parsed Document 43](#_Toc437954650)

[4. ciao-docs-enricher 45](#_Toc437954651)

[4.1. Description 45](#_Toc437954652)

[4.2. Download 45](#_Toc437954653)

[4.3. Configuration 46](#_Toc437954654)

[4.3.1. Spring XML 46](#_Toc437954655)

[4.3.2. CIAO Properties 47](#_Toc437954656)

[4.4. Log Events 51](#_Toc437954657)

[4.5. Technical Narrative 51](#_Toc437954658)

[5. ciao-docs-transformer 52](#_Toc437954659)

[5.1. Description 52](#_Toc437954660)

[5.2. Download 53](#_Toc437954661)

[5.3. Configuration 54](#_Toc437954662)

[5.3.1. Spring XML 54](#_Toc437954663)

[5.3.2. CIAO Properties 55](#_Toc437954664)

[5.4. Log Events 56](#_Toc437954665)

[5.5. Technical Narrative 56](#_Toc437954666)

[6. ciao-cda-builder 57](#_Toc437954667)

[6.1. Description 57](#_Toc437954668)

[6.2. Download 57](#_Toc437954669)

[6.3. Configuration 58](#_Toc437954670)

[6.3.1 Spring XML 58](#_Toc437954671)

[6.3.2. CIAO Properties 59](#_Toc437954672)

[6.4. Log Events 60](#_Toc437954673)

[6.5. Technical Narrative 60](#_Toc437954674)

[7. ciao-transport-spine 61](#_Toc437954675)

[7.1. Description 61](#_Toc437954676)

[7.2. Download 61](#_Toc437954677)

[7.3. Configuration 62](#_Toc437954678)

[7.3.1. Spring XML 62](#_Toc437954679)

[7.3.2. CIAO Properties 63](#_Toc437954680)

[7.4. Log Events 67](#_Toc437954681)

[7.5. Technical Narrative 67](#_Toc437954682)

[8. ciao-transport-dts 74](#_Toc437954683)

[8.1. Description 74](#_Toc437954684)

[8.2. Download 74](#_Toc437954685)

[8.3. Configuration 75](#_Toc437954686)

[8.3.1. Spring XML 75](#_Toc437954687)

[8.3.2. CIAO Properties 76](#_Toc437954688)

[8.4. Log Events 80](#_Toc437954689)

[8.5. Technical Narrative 80](#_Toc437954690)

[9. ciao-docs-finalizer 83](#_Toc437954691)

[9.1. Description 83](#_Toc437954692)

[9.2. Download 83](#_Toc437954693)

[9.3. Configuration 84](#_Toc437954694)

[9.3.1. Spring XML 84](#_Toc437954695)

[9.3.2. CIAO Properties 85](#_Toc437954696)

[9.4. Log Events 87](#_Toc437954697)

[9.5. Technical Narrative 87](#_Toc437954698)

[10. tkw-spine 91](#_Toc437954699)

[10.1. Description 91](#_Toc437954700)

[10.2. Download 91](#_Toc437954701)

[10.3. Configuration 91](#_Toc437954702)

[10.4. Log Events 91](#_Toc437954703)

[10.5. Technical Narrative 91](#_Toc437954704)

[11. Example Applications 92](#_Toc437954705)

[11.1. Send Discharge Summary via Spine 92](#_Toc437954706)

[Part 3: Deploy and Run 94](#_Toc437954707)

[12. Deploying the Cloud 94](#_Toc437954708)

[13. Operating the Cloud 95](#_Toc437954709)

[14. Deploying CIPs 96](#_Toc437954710)

[14.1 Install Ansible 97](#_Toc437954711)

[14.2 Playbooks Setup 98](#_Toc437954712)

[14.3 CIAO Base Services 99](#_Toc437954713)

[14.4. Application to Send Discharge Summary via Spine 101](#_Toc437954714)

[15. Operating CIPs 104](#_Toc437954715)

[Part 4: Building a new CIP 105](#_Toc437954716)

[16. Design 105](#_Toc437954717)

[16.1. Principles 105](#_Toc437954718)

[16.2. Methods and Tools 105](#_Toc437954719)

[16.3. Outputs 105](#_Toc437954720)

[17. Environments and Tools 106](#_Toc437954721)

[18. Testing 107](#_Toc437954722)

[19. Development 108](#_Toc437954723)

[ciao-docs-parser 108](#_Toc437954724)

[Configuring a new parser route 108](#_Toc437954725)

[Adding a new type of parser 108](#_Toc437954726)

[Creating mappings 109](#_Toc437954727)

[ciao-docs-transformer 109](#_Toc437954728)

[Configuring a new transformer route 109](#_Toc437954729)

[Adding a new type of transformer 109](#_Toc437954730)

[20. Contribution 111](#_Toc437954731)

[21. Assurance 112](#_Toc437954732)

# Copyright

The contents of this publication are protected by copyright. The information featured in this publication is the copyright of the Health and Social Care Information Centre (HSCIC) unless otherwise indicated.

© 2015 by the Health and Social Care Information Centre. All rights reserved.

Under the Open Government Licence you are encouraged to use and re-use the information in this publication free of charge. Re-use includes copying, issuing copies to the public, publishing, broadcasting and translating into other languages and its subsequent use in commercial or non-commercial enterprise.

The HSCIC grants you a worldwide, royalty-free, perpetual, non-exclusive licence to use the information subject to the following conditions. This licence does not affect your freedom under fair dealing or fair use or any other copyright or database right exceptions and limitations.

Permission to re-use is subject to the following conditions:

* you must acknowledge the source and our copyright in cases where you supply the information to others; "Copyright © 2015, Re-used with the permission of the Health and Social Care Information Centre. All rights reserved."
* you must reproduce the information accurately, without amendment or alteration;
* you must not use the information in a misleading way;
* you must not use the information for the principal purpose of advertising or promoting a particular product or service;
* ensure that you do not use the information in a way that suggests any official status or that the HSCIC endorses you or your use of the information;
* ensure that your use of the information does not breach the Data Protection Act 1998 or the Privacy and Electronic Communications (EC Directive) Regulations 2003.

Where you wish to re-use the information but the proposed re-use would contravene any of the conditions set out above, email enquiries@hscic.gov.uk to determine whether the proposed re-use would be permitted and what, if any, additional conditions may apply. The application should be in writing, identifying the information to be re-used and the purpose for re-use.

The information is licensed 'as is' and the HSCIC excludes all representations, warranties, obligations and liabilities in relation to the information to the maximum extent permitted by law.

The HSCIC is not liable for any errors or omissions in the information and shall not be liable for any loss, injury or damage of any kind caused by its use. The HSCIC does not guarantee the continued supply of the information.

# Foreword

*It would be nice to have a couple of paragraphs from Peter Counter!*

Peter Counter

Chief Technology Officer at the Health and Social Care Information Centre, UK

# Preface

## About This Book

This book is about CIAO – Care Integration and Orchestration - free open source software that allows care ICT systems to route, transform and work flow information between themselves and NHS England national care ICT services.

To be more precise CIAO is a library of free open source pre-built and tested Care Integration Patterns (CIPs) installable on a Linux platform for specific routing, transformation and work flow tasks. CIPs are packaged as Docker images so they can be easily deployed, configured and composed.

CIAO is also a development kit providing guidance and base code libraries for developing new CIPs.

CIAO originated as an open source project by a group of technologists; Adam Hatherly, Mike Kelly, Richard Dobson and Iestyn Evans, at the Health and Social Care Information Centre, UK. Open source projects are often poorly documented; therefore it was decided early on to write *CIAO the Book* as the definitive source of information about the project.

## Who Should Read This Book

For decision makers who decide on IT strategy, technology and product choices this book provides an overview of what problems CIAO seeks to solve and CIAO’s approach.

For architects who take an enterprise or solution level view of integration this book details the CIAO architecture and the library of developed CIPs.

For developers who want to design, build and test new CIPs or adapt existing ones this book gives guidance on software development.

For system administrators who want to deploy, operate and manage CIPs this book provides operational instructions.

## Navigating This Book

This book is divided into four parts.

*Part one* provides an introduction to CIAO and describes the architecture CIAO is built on.

*Part two* is the Care Integration Pattern catalogue. This contains detailed information for each of the individual CIPs developed by HSCIC. It also provides some examples of composing these CIPs into applications to solve specific care integration problems.

*Part three* gives instructions on how to deploy and run CIPs on the Linux platform.

*Part four* provides guidance on how to; design, code and test new CIPs.

## Conventions Used in This Book

The following typographical conventions are used in this book:

*Italic -* Indicates new terms, URLs, email addresses, filenames, and file extensions.

Constant width - Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

**Constant width bold -** Shows commands or other text that should be typed literally by the user.

*Constant width italic* - Shows text that should be replaced with user-supplied values or by values determined by context.

## Downloads

The CIAO source code for the software components developed by HSCIC can be found in the public repositories on GitHub at:

[*https://github.com/nhs-ciao*](https://github.com/nhs-ciao)

Jenkins is used as the continuous integration server to build the latest CIAO binaries for the software components developed by HSCIC. The binaries can be found at:

[*http://hcdn-build1.cloudapp.net/view/CIAO/*](http://hcdn-build1.cloudapp.net/view/CIAO/)

*[This may change to an non-Azure URL]*

To find a binary:

* select a project
* select workspace
* select target
* select binary file (.jar, .tar.gz or .zip) to download

Docker images for the CIPs developed by HSCIC and additional Docker images for supporting services packaged for CIAO can be found on Docker Hub at:

[*https://hub.docker.com/u/hscic/*](https://hub.docker.com/u/hscic/)

## Further Reading

Haloi, S. *Apache ZooKeeper Essentials.* Packt Publishing, 2015.

Hochstein, L. *Ansible: Up and Running.* O'Reilly Media, 2015.

Ibsen, C. & Anstey, J. *Camel in Action.* Manning Publications, 2010.

Newman, S. *Building Microservices.* O'Reilly Media, 2015.

Snyder, B., Bosanac, D. & Davies, R. *ActiveMQ in Action.* Manning Publications, 2011.

## Acknowledgments

The authors would like to thank the dragons in the internal HSCIC Innovation competition dragons den; Andy Williams, Tom Denwood, Beverley Bryant and [etal] for launching CIAO by choosing it as an innovation winner. Hopefully Beverley the lads have indeed “*cracked on*” as you requested.

Without the resources secured by Peter Counter moving CIAO from idea to working code would not have been possible.

We would also like to thank all our colleagues at HSCIC who have directly or indirectly contributed to CIAO. We would particularly like to thank; Marcus Fearnett, [etal]

## About the Cover Illustration

The cover illustration is from the specialist cancer care centre at University College London Hospital, England. In the Entrance hall a huge installation hangs from the ceiling with objects of significance in a variety of shapes and sizes; an apposite metaphor for the current state of ICT integration within the care and health sector.

# Part 1: Understanding CIAO

# 1. Introduction

CIAO came from a mixture of frustration, need and opportunity from a group of technologists (the authors) working in the Health and Social Care Information Centre (HSCIC) in the UK.

Frustration at how difficult it was to implement integration between ICT systems in the health and care sector.

Need to be able to do this quickly and cheaply to meet the growing challenges of the economic, efficient and effect delivery of health and care services.

Need to test, through implementation, the many integration specifications and standards the HSCIC create (in the words of Microsoft to “*eat our own dog food*”).

Opportunity of the emergence of containerisation and microservices within the IT industry to design bigger and better integration solutions.

Opportunity of resources made available through winning an internal innovation competition within the HSCIC.

Within England the NHS national strategy for patient centric care necessitates the need to connect and share care information between multiple care providers. These care providers use a variety of ICT systems to record and manage direct patient care. This gives a landscape of distributed, autonomous and multi-standard care ICT systems.

Standards are often posited as the “silver bullet” that moves you from integration to interoperability, if everyone uses the same standards then systems should just plug and play. In real life high level standards around the structure and meaning of care information are still underdeveloped, duplicated and sparsely used. Therefore pragmatically, for the foreseeable future, there is a need for system integration middleware technology to “*connect any*” ICT systems within health economies.

Middleware technology provides the familiar functions of routing, transformation and orchestration (work flow) of interactions and their content between disparate external ICT systems. There are many commercial and open source middleware products and frameworks available. Products usually provide a comprehensive service through including bundled design, management and monitoring tools. Frameworks just provide a codebase that allows you to build your own integration solution.

There are cost, learning time and skill barriers to using both commercial and open source middleware products and frameworks. CIAO aims to reduce these barriers while still maintaining flexibility and adaptability. To be more specific CIAO aims to accelerate integration by:

* Packaging integration components as microservices (CIPs)
* Making these CIPs easy to install, configure and manage
* Providing a set of common CIPs out of the box
* Providing a development kit to allow the building of new CIPs

Commoditise by:

* Provide CIPs as free open source software
* Developing a community of support for the use and development of CIPs
* Ensuring enterprise class resilience and scalability of CIPs
* Giving SME opportunities to offer CIAO run, support and development services

Evolve by:

* Moving from integration technology to integration patterns
* Moving from static to dynamic integration behaviours
* Building, seeding and curating a health and care integration ecology

# 2. Architecture

## 2.1. Microservices

Talk about how CIPs are realised as microservices. Although there are no agreed definitions around microservices, need to provide some context and show how the CIP key architectural principles map to it.

## 2.2. Containers

Talk about what containers (in the generic sense) are. So would cover traditional application servers, EJB, OSGi, Spring(?) and the newer LXC (Linux Containers) (Docker and Rocket). Need to carefully explain how containers relate to machines/VMs. Talk about models of deploying microservices to containers. Define our model, explain advantages and disadvantages:

* Container = LXC aka Docker
* Microservice = Linux application
* Each microservice is packaged and deployed in a single container

## 2.3. Care Integration Patterns

Explain the concept of a CIP. Define the key architectural principles CIPs implement and explain why. These will include:

* Interface Contracts – standard public interface definitions.
* Dependency Injection – just through configuration e.g. say a CIP needs to persist data to a message queue; then the location and name of that message queue would be a configuration item that is loaded at run time.
* Monitoring and management – being able to see what the CIP is doing (has done) and provide some external control over at least its lifecycle, but possibly its internal behaviour. Done through interface contracts.
* Diagnostics – essentially being able to ask a CIP to self-test itself. Done through interface contracts.
* Composing – being able to wire together different CIPs without the CIPs individually needing to be aware.

## 2.4. Applications

Talk about composing CIPs into Care apps.

## 2.5. Clouds

Our containers are deployed to essentially a cloud of real machines or VMs. Talk about the key issues to address at the cloud level (may not be able to actually provide answers for some of these yet as this is the bleeding edge of containers). These will include:

* Deployment of containers
* Configuration of containers
* Monitoring and management of containers
* Scalability and resilience of containers – this is around having multiple instances of the same container with load balancing in front, and having run policies e.g. I want 5 instances of this container operational at any one time
* Security – this is perimeter security in and out of the cloud, not necessary at the container to container level – although this is up for debate
* Composing/wiring containers together

Define our model and explain advantages and disadvantage. This may include going for Kubernetes (though Docker are rolling their own).

## 2.6. Product and Technology Stack

CIAO uses a variety of open source products and technologies. These provide the supporting services and infrastructure needed to run and manage CIPs.



*Figure 2.6-1. CIAO Technologies and Products*

### 2.6.1. Linux

CIAO needs to run on Linux to be able to use Docker. However as the CIPs are developed in Java they can be run as non-containerised programs on any operating system that provides a JVM. This will require you to find, deploy, configure and manage your own supporting services, rather than using those that have already been bundled with CIAO for use on Linux.

The version of Linux you use must meet the minimum requirements for Docker:

* 64-bit installation
* kernel must be 3.10 at minimum

We are going to use Ubuntu 14.04 LTS (Trusty) for all our examples, but you can use any Linux distribution that meets the Docker requirements.

Each CIAO Linux host must have the following services and applications installed and running:

* OpenSSH server
* curl
* [etal]

CIAO assumes that all hosts within a CIAO cloud are connected together in a secure network environment, so that network access to and from the CIAO cloud is controlled, but that IP networking between hosts within the CIAO cloud is unrestricted. It is important that all hosts within the CIAO cloud have Internet access so they can connect to Docker Hub to pull down CIAO Docker images.

### 2.6.2. Docker

To use the CIPs packaged as Docker containers you have to install the Docker daemon (Docker Engine) on each Linux host that will be part of a CIAO cloud.

For installation instructions see:

[*https://docs.docker.com/installation/*](https://docs.docker.com/installation/)

For example for Ubuntu Trusty, add a new gpg key:

$ sudo apt-key adv --keyserver hkp://pgp.mit.edu:80 \

-- recv-keys 58118E89F3A912897C070ADBF76221572C52609D

Open the /etc/apt/sources.list.d/docker.list file in an editor. If the file doesn’t exist, create it. Add a single entry:

deb https://apt.dockerproject.org/repo ubuntu-trusty main

Save and close the /etc/apt/sources.list.d/docker.list file. Update the apt package index:

$ sudo apt-get update

Install Docker:

$ sudo apt-get install docker-engine

Verify Docker is installed correctly:

$ sudo docker run hello-world

This command downloads a test image and runs it in a container. When the container runs, it prints an informational message. Then, it exits.

If you are new to Docker, then you can find comprehensive documentation at:

[*https://docs.docker.com/*](https://docs.docker.com/)

Some useful Docker commands to know are listed below.

To list all the Docker images on a host:

$ sudo docker images

To list all the Docker containers on a host:

$ sudo docker ps –a

To list all the running Docker containers on a host:

$ sudo docker ps

To connect a new TTY to a running container:

$ sudo docker exec –t –i *{container name}* /bin/bash

To stop all Docker containers on a host:

$ sudo docker stop $(sudo docker ps -q)

To remove all Docker containers on a host:

$ sudo docker rm $(sudo docker ps -a -q)

To remove all Docker images on a host:

$ sudo docker rmi $(sudo docker images -q)

### 2.6.3. etcd

Because CIPs can be deployed in a variety of different ways, there is a need for a consistent way of managing the configuration of CIPs even when they may be distributed across many containers and hosts. There are a number of approaches for managing configuration for distributed cloud solutions, including Zookeeper, Doozer, etcd.

The tool chosen for CIAO is etcd. This provides a lightweight, simple solution which allows configuration to be easily created and accessed via a REST API, and also provides replication to allow each CIP to have its configuration kept in-sync as new instances are created.

etcd has been produced as part of the CoreOS project, and more details can be found on their github page at:

[*https://github.com/coreos/etcd*](https://github.com/coreos/etcd)

Information about the Docker image of etcd used by CIAO can be found at:

[*https://github.com/coreos/etcd/blob/master/Documentation/docker\_guide.md*](https://github.com/coreos/etcd/blob/master/Documentation/docker_guide.md)

To provide resilience CIAO uses clustered etcd, where two or more etcd instances are clustered together. In an etcd cluster any changes to one cluster member is automatically replicated to the other cluster members (peers).

The etcd Docker image exposes the IP port 2380 to allow etcd peers to talk to each other in an etcd cluster, and the IP ports 2379 and 4001 to allow a client to talk to an etcd instance. By convention CIAO uses 4001. CIAO uses HTTP rather than HTTPS to talk to etcd, which means CIAO assumes it is operating in a secure network environment.



*Figure 2.6-2. Docker image for etcd interface*

To create or change a value in etcd within the secure network environment you can use curl to HTTP PUT a key and associated value, for example:

$ curl -L -X PUT http://localhost:4001/v2/keys/testkey \

-d value="testvalue"

To read the value from etcd use curl to HTTP GET the key value, for example:

$ curl http://localhost:4001/v2/keys/testkey

The result is returned in JSON format, for example:

{"action":"get","node":{"key":"/testkey", \

"value":"testvalue","modifiedIndex":10, \

"createdIndex":10}}

To test that etcd is replicating content successfully within an etcd cluster curl to HTTP GET the key value from a different etcd cluster member, for example:

$ curl http:// 10.210.162.28:4001/v2/keys/testkey

By convention CIAO deploys an etcd instance on each host that runs CIPs as well as the management host. To provide resilience an etcd cluster of at least two members is required. By convention CIAO names the etcd cluster etcd-cluster-ciao and names individual etcd instances etcd-*{host moniker}*. The Docker container running etcd is named ciao-etcd by convention.



*Figure 2.6-3. CIAO etcd cluster*

Currently etcd does not provide any monitoring functionality such as JMX. Therefore the only way to monitor the state of the etcd cluster is to monitor the state of each individual etcd Docker container in the cluster to see if it has stopped.

The host to container mappings for the default ciao-etcd deployment are:

|  |  |
| --- | --- |
| **Network Ports** | |
| **Container** | **Host** |
| 2379 | 2379 |
| 2380 | 2380 |
| 4001 | 4001 |

To help view and change values in etcd, rather than using curl a web based browser application called etcd-browser can be used. The Docker image for etcd-browser can be found at:

<https://hub.docker.com/r/buddho/etcd-browser/>

The etcd-browser Docker image exposes the IP port 8000 to allow a web browser to connect to it.



*Figure 2.6-4. Docker image for etcd-browser interface*

By convention CIAO deploys one instance of etc-browser on the management host. The Docker container running etcd-browser is named ciao-etcdbrowser by convention.

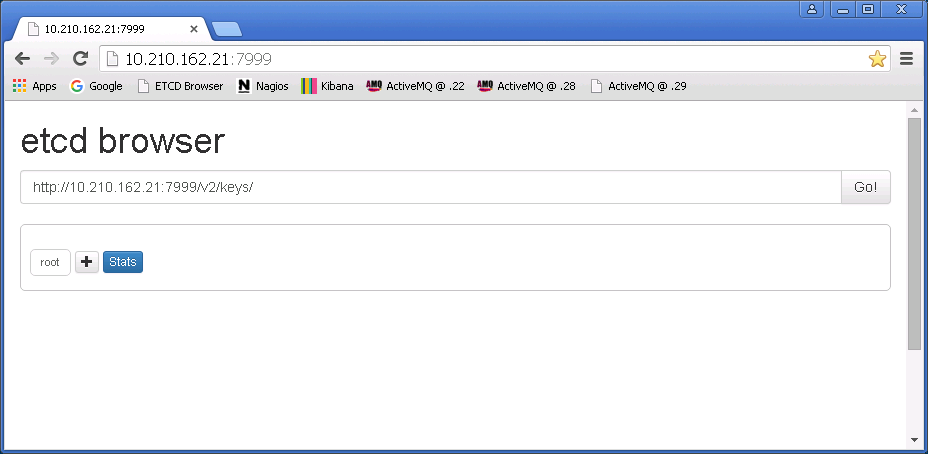
The host to container mappings for the default ciao-etcdbrowser deployment are:

|  |  |
| --- | --- |
| **Network Ports** | |
| **Container** | **Host** |
| 8000 | 7999 |

To connect to etcd-browser you can point your web browser at the URL:

http://*{network address of etcd-browser}*:7999

The etcd-browser console should be opened as shown below:



### 2.6.4. ZooKeeper

ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. CIAO requires it to support the resilient deployment of ActiveMQ as described later and so is a necessary evil (more stuff to be deployed unfortunately).

ZooKeeper is an Apache project, and more details can be found at:

[*https://zookeeper.apache.org/*](https://zookeeper.apache.org/)

Information about the Docker image of ZooKeeper used by CIAO can be found at:

[*https://hub.docker.com/r/jplock/zookeeper/*](https://hub.docker.com/r/jplock/zookeeper/)

For a reliable ZooKeeper service, ZooKeeper needs to be deployed in a cluster known as an ensemble. As long as a majority of the ensemble are up, the service will be available. Because Zookeeper requires a majority, it is best to use an odd number of hosts.

The ZooKeeper Docker image exposes the IP ports 2888 and 3888 to allow ZooKeeper peers to talk to each other in a ZooKeeper cluster, and the IP port 2181 to allow a client to talk to a ZooKeeper instance. By convention CIAO does not enable ZooKeeper authentication or authorisation but assumes it is operating in a secure network environment.

The ZooKeeper Docker image also provides two data volumes:

/opt/zookeeper/conf where configuration files are stored

/tmp/zookeeper where data and log files are stored



*Figure 2.6-5. Docker image for ZooKeeper interface*

You can monitor the health and status of individual ZooKeeper instances by using a set of four letter words that can be sent to the client port using telnet or nc. For example sending ruok checks whether the instance is running without any error. The instance will respond with imok if it is running. If the instance is in some error state, it will not respond to this command:

$ telnet localhost 2181

Trying ::1...

Connected to localhost.

Escape character is '^]'.

ruok

imokConnection closed by foreign host.

$

You can also use the stat command to see the status of a ZooKeeper instance:

$ telnet localhost 2181

Trying ::1...

Connected to localhost.

Escape character is '^]'.

stat

Zookeeper version: 3.4.6-1569965, built on 02/20/2014 09:09 GMT

Clients:

/172.17.42.1:33995[0](queued=0,recved=1,sent=0)

/10.210.162.28:36320[1](queued=0,recved=64369,sent=64376)

/10.210.162.22:40594[1](queued=0,recved=30075,sent=30075)

Latency min/avg/max: 0/0/56

Received: 94449

Sent: 94455

Connections: 3

Outstanding: 0

Zxid: 0x1900000025

Mode: follower

Node count: 8

Connection closed by foreign host.

$

By convention CIAO deploys a ZooKeeper instance on a minimum of three hosts. The Docker container running ZooKeeper is named ciao-zookeeper by convention.



*Figure 2.6-6. CIAO ZooKeeper cluster*

ZooKeeper provides monitoring and management capabilities with Java Management Extensions (JMX).

The host to container mappings for the default ciao-zookeeper deployment are:

|  |  |
| --- | --- |
| **Network Ports** | |
| **Container** | **Host** |
| 2181 | 2181 |
| 2888 | 2888 |
| 3888 | 3888 |
| **File Volumes** | |
| **Container** | **Host** |
| /opt/zookeeper/conf | /opt/ciao\_zookeeper/conf |
| /tmp/zookeeper | /var/lib/ciao\_zookeeper |

### 2.6.5. ActiveMQ

To ensure resilience of communication between CIPs, CIAO uses persisted messaging. There are a number of open source message brokers available. The product chosen for CIAO is ActiveMQ. This provides a message broker for remote communication between systems using the JMS (Java Message Service) specification.

ActiveMQ is an Apache project, and more details can be found at:

[*http://activemq.apache.org/*](http://activemq.apache.org/)

Information about the Docker image of ActiveMQ used by CIAO can be found at:

[*https://hub.docker.com/r/hscic/ciao-activemq/*](https://hub.docker.com/r/hscic/ciao-activemq/)

To provide resilience CIAO uses ActiveMQ clustered as Master/Slave. With Master/Slave messages are replicated to a slave broker so that if you have a failure of the master you get immediate failover to the slave with no message loss. The Master/Slave ActiveMQ configuration used by CIAO is the Replicated LevelDB Store. This uses Apache ZooKeeper to pick a master from a set of broker nodes configured to replicate a LevelDB Store (a high performance file based store ActiveMQ implements) and to pick a new master in case of failure. The Replicated LevelDB Store Master/Slave option allows you to have multiple slaves.

The ActiveMQ Docker image exposes the IP port 8161 to allow access to the ActiveMQ web console and the IP port 61616 to allow a client to talk to the ActiveMQ broker using the TCP transport. The IP port 61619 is used for replication within an ActiveMQ cluster. CIAO does not use SSL to secure TCP communication with ActiveMQ which means CIAO assumes it is operating in a secure network environment.

The ActiveMQ Docker image also provides two data volumes:

/opt/activemq/conf where configuration files are stored

/opt/activemq/data where data files are stored

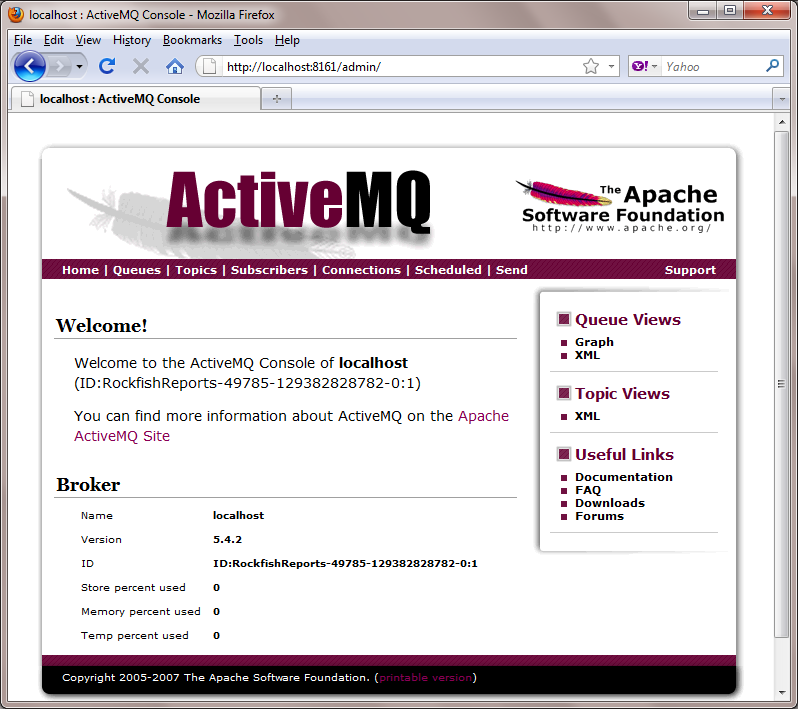


*Figure 2.6-7. Docker image for ActiveMQ interface*

To test that an ActiveMQ broker is running you can point your web browser at the URL:

http://*{network address of broker}*:8161/admin

The ActiveMQ web console should be opened as shown below. The default login username and password are admin and admin.



Note that within an ActiveMQ master/slave cluster, only the current master will be running the web console. This can be somewhat frustrating as you need to try and connect to the web console on each member of the cluster until you find a successful connection with the member who is the current master.

By convention CIAO deploys a minimum of three ActiveMQ brokers, one master and two slaves, to provide resilience. By convention CIAO names the ActiveMQ brokers within the cluster CiaoBroker. The Docker container running ActiveMQ is named ciao-activemq by convention.



*Figure 2.6-8. CIAO ActiveMQ Master/Slave*

ActiveMQ provides monitoring and management capabilities with Java Management Extensions (JMX).

The host to container mappings for the default ciao-activemq deployment are:

|  |  |
| --- | --- |
| **Network Ports** | |
| **Container** | **Host** |
| 8161 | 8161 |
| 61616 | 61616 |
| 61619 | 61619 |
| **File Volumes** | |
| **Container** | **Host** |
| /opt/activemq/conf | /opt/ciao\_activemq/conf |

### 2.6.6. Nagios

Nagios is an open source application to monitor servers, services and applications. In the event of a failure Nagios can alert support staff.

More details about Nagios can be found at:

<https://www.nagios.org/>

Information about the Docker image of Nagios used by CIAO can be found at:

<https://hub.docker.com/r/cpuguy83/nagios/>

The Nagios Docker image exposes the IP port 80 to allow access to the Nagios web console.

The Nagios Docker image also provides five data volumes:

/opt/nagios/var where log files are stored

/opt/nagios/etc where configuration files are stored

/opt/nagios/libexec where Nagios plugins are stored

/var/log/apache2 where apache2 log files are stored

/usr/share/snmp/mibs where SNMP MIBS are stores

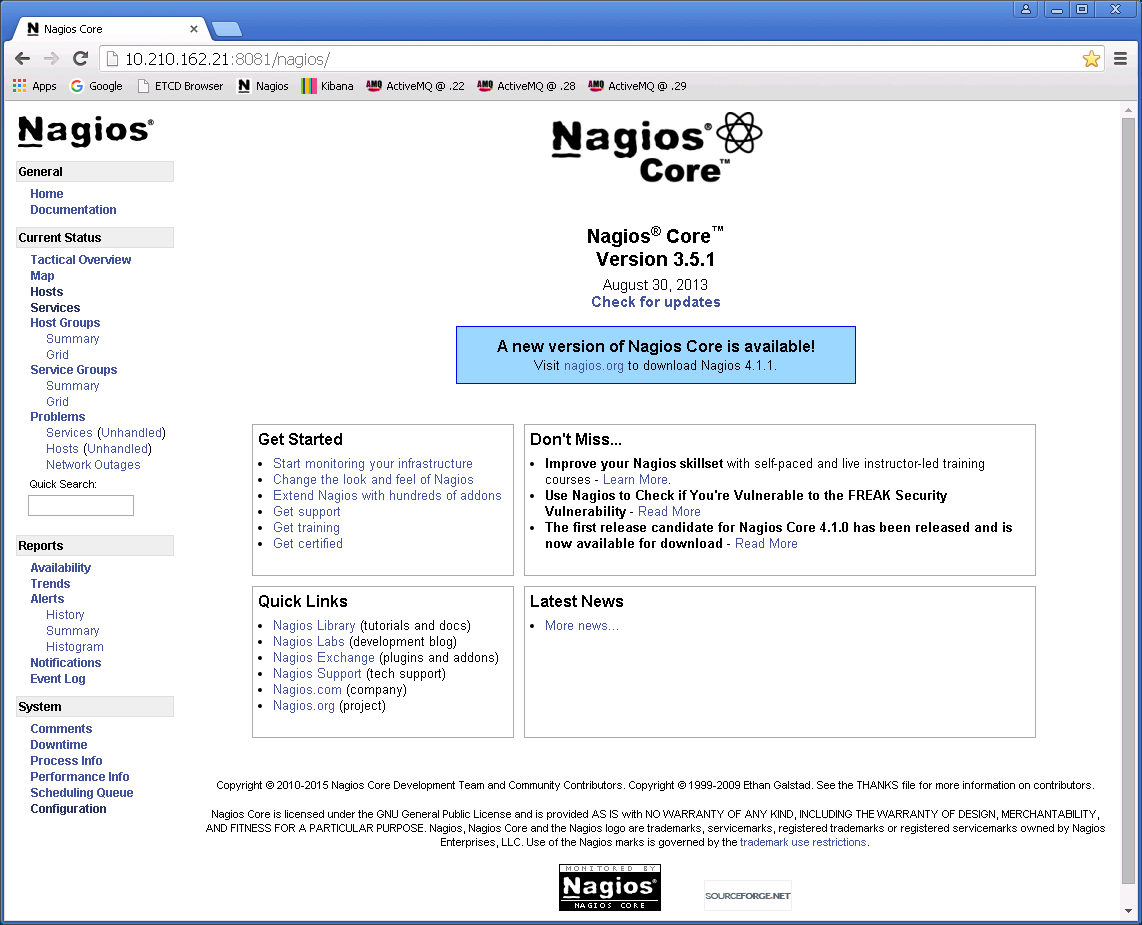


*2.6-9. Docker image for Nagios interface*

To test that Nagios is running you can point your web browser at the URL:

http*://{network address of Nagios host}*:8081/nagios/

The Nagios web console should be opened as shown below. The default login username and password are nagiosadmin and nagios.



By convention CIAO deploys one instance of Nagios on the management host. The Docker container running Nagios is named ciao-nagios by convention.

The host to container mappings for the default ciao-nagios deployment are:

|  |  |
| --- | --- |
| **Network Ports** | |
| **Container** | **Host** |
| 80 | 8081 |
| **File Volumes** | |
| **Container** | **Host** |
| /opt/nagios/etc | /opt/ciao\_nagios/etc |

The default deployment of CIAO will also create the following host and service configurations for monitoring:

* Check host alive
* Check SSH Server
* Check host load
* Check host swap space
* Check Zookeeper alive
* Check Logspout alive
* Check etcd alive
* Check ActiveMQ Web Console alive
* Check etcd Browser alive
* Check Logstash alive
* Check Elastic alive
* Check Kibana alive
* [etal]

### 2.6.7. Logspout

CIPs implement logging to record operational activity and error conditions. These log records need to be collected for analysis and viewing. CIAO starts this process by pulling log records out of all the running CIAO Docker containers. It uses Logspout, which is a log router for Docker containers. It runs in its own Docker container on a host and attaches to all containers on a host, then routes their logs to a configured destination.

Logspout has been developed by Gilder Labs, and more details can be found at:

[*https://github.com/gliderlabs/logspout*](https://github.com/gliderlabs/logspout)

Information about the Docker image of Logspout used by CIAO can be found at:

[*https://hub.docker.com/r/gliderlabs/logspout/*](https://hub.docker.com/r/gliderlabs/logspout/)

The Logspout Docker image exposes the IP port 8000 to allow a client to talk to a Logspout instance. Logspout provides no authentication or authorisation but assumes it is operating in a secure network environment.

The Logspout Docker image also provides a data volume:

/mnt/routes where routing configuration files are stored



*Figure 2.6-10. Docker image for Logspout interface*

You can monitor the log records Logspout streams by:

$ curl http://localhost:8000/logs

The log routing destination is set by passing the URI of a remote syslog as a command to the Docker container. By convention CIAO uses a single syslog (Logstash) running on the management host. The URI is:

syslog://*{hostaddress}*:514

Logspout does not provide any monitoring functionality such as JMX. Therefore the state of each individual Logspout Docker container has to be monitored to see if it has stopped.

By convention CIAO deploys Logspout to each host running CIPs, but not to the management host.

The host to container mappings for the default ciao-logspout deployment are:

|  |  |
| --- | --- |
| **Network Ports** | |
| **Container** | **Host** |
| 8000 | 8000 |

### 2.6.8. Logstash

CIAO uses Logstash as the service to collect log records routed by Logspout. Logstash is an open source data collection engine that can accept input from many different sources in many different formats, apply configurable filters before forwarding on to many different destinations.

Logstash details can be found at:

[*https://www.elastic.co/guide/en/logstash/current/index.html*](https://www.elastic.co/guide/en/logstash/current/index.html)

Information about the Docker image of Logstash used by CIAO can be found at:

[*https://hub.docker.com/r/willdurand/elk/*](https://hub.docker.com/r/willdurand/elk/)

This image in fact contains the ELK stack, an installation of Logstash, and installation of Elastic and an installation of Kibana.

### 2.6.9. Elastic

TO DO

### 2.6.10. Kibana

TO DO

curl <http://localhost:8080/>

### 2.6.11. File Sharing

Some CIPs may require access to directories on remote Linux, Windows or Macintosh servers to either read or write data files. For example the **ciao-docs-parser** reads files from a configured directory to parse them. This directory could reside for example on a Windows 2012 Server.

TO DO

* SAMBA Server
* Microsoft services for NFS
* Macintosh NFS
* Linux NFS Server

### 2.6.12. Port Map

The table below shows the ports that are used by a default CIAO supporting services and infrastructure deployment.

|  |  |
| --- | --- |
| **Port number** | **Used by** |
| 514 | ELK Logstash syslog |
| 2181 | ZooKeeper client |
| 2379 | etcd client |
| 2380 | etcd peer to peer |
| 2888 | ZooKeeper peer to peer |
| 3888 | ZooKeeper peer to peer |
| 4001 | etcd client |
| 7999 | Etcd browser |
| 8000 | Logspout |
| 8080 | Kibana web console |
| 8081 | Nagios web console |
| 8161 | ActiveMQ web console |
| 9200 | ELK Elastic |
| 61616 | ActiveMQ client TCP transport |
| 61619 | ActiveMQ peer to peer |

## 2.7. CIAO Conventions

### 2.7.1. CIP Name

CIPs should be named using the following convention:

ciao-*{function}*

Where *{function}* should describe the main thing the CIP acts on and the main action on that thing. Examples from the HSCIC CIP catalogue are:

ciao-docs-parser

ciao-cda-builder

ciao-transport-spine

A specific version of a CIP is named by appending a version number to the end of the generic CIP name, for example:

ciao-docs-parser-1.0.0-SNAPSHOT

### 2.7.2. CIP Directory Structure

By convention a CIP should be installed in the /opt Linux directory in its own subdirectory named after the CIP (called the CIP home directory), for example:

/opt/ciao-docs-parser-1.0.0-SNAPSHOT

Within the CIP home directory are the following files and directories.

Licence information:

LICENCE.txt

General read me information:

README.txt

The executable JAR file:

*{cip}*.jar

For example:

ciao-docs-parser-1.0.0-SNAPSHOT.jar

This can be run from the command line by:

java –jar *{cip}*.jar

For example:

$ java -jar ciao-docs-parser-1.0.0-SNAPSHOT.jar

There are several optional command line parameters that can be provided.

The URL of etcd in order to access CIP configuration information:

--etcdURL *{URL}*

For example:

--etcdURL http://127.0.0.1:4001

A classifier value for CIP configurations, to allow for multiple running CIPs to use different configurations if required:

--classifier=*{classifier name}*

For example:

--classifier=uat\_cluster

A path to the directory containing the CIP configuration file, this will override the default path (see the optional CIP configuration file below):

--configPath *{path}*

For example:

--configPath /etc/ciao

The etcdURL, classifier and configPath values can alternatively be set through environment variables:

CIAO\_ETCD\_URL

CIAO\_CONFIG\_PATH

CIAO\_CONFIG\_CLASSIFIER

Scripts (.bat for windows and .sh for Linux) to start the CIP with support for hawtio monitoring enabled via the Jolokia JVM agent:

run.bat

run.sh

A directory containing the supporting JAR files:

/lib

A directory containing configuration files:

/config

This will contain a configuration file for logging:

log4j.properties

This will contain a configuration file for the Jolokia JVM agent:

jolokia.properties

Within the logging configuration file, the convention is to set the logging directory to:

/logs

Outside of the CIP home directory an optional configuration file for the CIP itself called:

*{cip}*.properties

For example:

ciao-docs-parser-1.0.0-SNAPSHOT.properties

This can be created in the directory:

*{user-home}*/.ciao

Where *{user-home}* is the Linux home directory of the user who is running the CIP. For example if the user running the ciao-docs-parser CIP is root then a configuration file for the CIP would be:

/root/.ciao/ciao-docs-parser-1.0.0-SNAPSHOT.properties

### 2.7.3. CIP Log Record Structure

A CIP log record log has a common structure of:

*TIMESTAMP* *SEVERITY* *CLASSNAME* **- Camel** /

**{***CAMEL\_CONTEXT\_PARAMETERS***}** **–** *MESSAGE***. Values -->** /

*MESSAGE\_PARAMETERS*

Where:

*TIMESTAMP* is the date and time of the event being logged. This is in the format YYYY-MM-DD HH:MM:SS,milliseconds for example 2015-10-12 11:53:51,617

*SEVERITY* is the type of the event being logged. Valid values are:

**ERROR** - error events that might still allow the application to continue running

**WARN** - potentially harmful events

**INFO** - coarse-grained informational events

**DEBUG** - fine-grained informational events useful for debugging an application

**TRACE** - finer-grained informational events than the **DEBUG**

Typically you should only see **INFO** and above are from a production CIP.

*CAMEL\_CONTEXT\_PARAMETERS* is a list of key value pairs that relate to the Camel context involved in the processing of the message or interaction or function. This uses the format key1:value1 key2:value2 … for example RouteId:1234 ExchangeId:5678

*MESSAGE* is the message or interaction or function. For example Searching for SpineEndpointAdddress in cache

*MESSAGE\_PARAMETERS* is a list of key value pairs that are the parameter values for the message or interaction or function. This uses the format *key1=value1 key2=value2 …* for example Service=service Action=action ODSCode=ods-code

An example full log record is:

2015-10-12 11:53:51,617 INFO CachingEndpointAddressRepository - Camel {RouteId:1234 /

ExchangeId:5678 MessageId:9876 CorrelationId:5432} - Searching for SpineEndpointAdddress in /

cache. Values --> Service=service Action=action ODSCode=ods-code

### 2.7.4. Configuration Management

The CIAO preference is to store all CIP configuration information in etcd.

All configuration information in etcd is in the form of key-value pairs, and is created in a simple hierarchical directory structure. The convention used in CIAO is as follows:

/ciao/*{cipname}*/*{version}*/*{classifier}*

Within this directory, all CIPs will have a key called "version", which stores the version of the CIP that is running and an optional “classifier” , to allow for multiple running CIPs to use different configuration if required.

If you are unable to use etcd for some reason (or choose not to), CIPs will fall-back on local configuration files. These will by default be saved in the user's home directory in the path:

~/.ciao/<cipname>-<version>.properties

Obviously, the use of config files will mean that configuration will need to be managed individually on all running instances - there is no mechanism for keeping these configuration files in-sync.

The first time a CIP is run, it will attempt to connect to the etcd URL provided. If it is able to connect, it will check whether configuration already exists for this CIP. It will do this by looking for the /ciao/<cipname>/<version> or /ciao/<cipname>/<version>/<classifier> key. If config already exists, it will be used immediately. If no config exists, a default set of configuration values will be added to the etcd repository. If the CIP is unable to connect to the provided URL, an error will be returned.

If no etcd URL is provided, the CIP will look for a local configuration file (as described above). If a config file path is used, the CIP will look in that path, otherwise it will look in the home path (as described above). If a config file is found, it will be used, and if not, a default configuration file will be created.

As all CIPs will require the above configuration capabilities, a common CIAO code library has been created for this, which all CIPs can use. This is called ciao-configuration, and is available as a project in the ciao-utils repository.

# Part 2: Care Integration Pattern Catalogue

# 3. ciao-docs-parser

## 3.1. Description

The purpose of this CIP is to process an incoming binary document by parsing it and extracting a set of key/value properties before publishing the parsed document for further processing by other CIPs.

ciao-docs-parser is built on top of Apache Camel and Spring Framework, and can be run as a stand-alone Java application, or as a Docker container.

Each instance of the CIP can host multiple routes, where each route follows the following basic structure:



The input folder supports any document format recognised by the configured parsers and extractors.

The output queue format accepts a JSON-encoded representation of ParsedDocument.

The details of the JMS queues and document parsers are specified at runtime through a combination of ciao-configuration properties and Spring XML files.

The following document parsers are provided:

* TikaDocumentParser - A parser backed by Apache Tika. Tika is used to interpret the document file format and a configured PropertiesExtractor is used to extract key/value pairs from the text.
* MultiDocumentEnricher - A parser which sequentially delegates to multiple configured parsers until one succeeds or all fail to parse the document.

The following properties extractors are provided:

* RegexPropertiesExtractor - Properties are extracted through a series of regular expressions.
* SplitterPropertiesExtractor - Splits the document into multiple parts via selectors (e.g. XPathNodeSelector) and delegates extraction for that section to a configured property extractor.
* ciao-docs-parser-kings - Module which provides various parsers for Kings College University Hospital PDF and Word documents.

For more advanced usages, a custom document parser can be integrated by implementing parser Java interfaces and providing a suitable spring XML configuration on the classpath.

## 3.2. Download

To pull down the code, run:

git clone https://github.com/nhs-ciao/ciao-docs-parser.git

You can then compile the module via:

cd ciao-docs-parser-parent

mvn clean install -P bin-archive

This will compile a number of related modules - the main CIP module is ciao-docs-parser, and the full binary archive (with dependencies) can be found at ciao-docs-parser\target\ciao-docs-parser-{version}-bin.zip. To run the CIP, unpack this zip to a directory of your choosing and follow the instructions in the README.txt.

The CIP requires access to various file system directories and network ports (dependent on the selected configuration):

**etcd:**

* Connects to: localhost:2379

**ActiveMQ:**

* Connects to: localhost:61616

**Hazelcast:**

* Multicast discovery: 224.2.2.3:54327 (If enabled)
* Listens on: \*:5701 (If port is already taken, the port number is incremented until a free port is found)
* Connects to clustered nodes defined by the hazelcast.network.join.tcp\_ip.members property

**Filesystem:**

* If etcd is not available, CIAO properties will be loaded from: ~/.ciao/
* The default configuration creates/uses input, completed, and error directories in the CIP working directory. These can be altered by changing the CIAO properties configuration (via etcd, or the properties file in ~/.ciao/)

Docker download and run TODO.

## 3.3. Configuration

For further details of how ciao-configuration and Spring XML interact, please see ciao-core.

### 3.3.1. Spring XML

On application start-up, a series of Spring Framework XML files are used to construct the core application objects. The created objects include the main Camel context, input/output components, routes and any intermediate processors.

The configuration is split into multiple XML files, each covering a separate area of the application. These files are selectively included at runtime via CIAO properties, allowing alternative technologies and/or implementations to be chosen. Each imported XML file can support a different set of CIAO properties.

The Spring XML files are loaded from the classpath under the META-INF/spring package.

**Core:**

* beans.xml - The main configuration responsible for initialising properties, importing additional resources and starting Camel.

**Repositories:**

An `IdempotentRepository' is configured to enable multiple consumers access the same folder concurrently.

* repository/memory.xml - An in-memory implementation suitable for use when there is only a single consumer, or multiple-consumers are all contained within the same JVM instance.
* repository/hazelcast.xml - A grid-based implementation backed by Hazelcast. The component is hosted entirely within the JVM process and uses a combination of multicast and point-to-point networking to maintain a cross-server data grid.

**Processors:**

* processors/default.xml - Creates individual parsers from the ciao-docs-parser-kings module, and initialises an auto-detect parser to try each sequentially until a match is found.

**Messaging:**

* messaging/activemq.xml - Configures ActiveMQ as the JMS implementation for input/output queues.
* messaging/activemq-embedded.xml - Configures an internal embedded ActiveMQ as the JMS implementation for input/output queues. (For use during development/testing)

### 3.3.2. CIAO Properties

At runtime ciao-docs-parser uses the available CIAO properties to determine which Spring XML files to load, which Camel routes to create, and how individual routes and components should be wired.

**Camel Logging:**

* camel.log.mdc - Enables/disables Mapped Diagnostic Context in Camel. If enabled, additional Camel context properties will be made available to Log4J and Logstash.
* camel.log.trace - Enables/disables the Tracer interceptor for Camel routes.
* camel.log.debugStreams - Enables/disables debug logging of streaming messages in Camel.

**Spring Configuration:**

* repositoryConfig - Selects which repository configuration to load: repositories/${repositoryConfig}.xml
* processorConfig - Selects which processor configuration to load: processors/${processorConfig}.xml
* messagingConfig - Selects which messaging configuration to load: messaging/${messagingConfig}.xml

**Routes:**

* documentParserRoutes - A comma separated list of route names to build

The list of route names serves two purposes. Firstly it determines how many routes to build, and secondly each name is used as a prefix to specify the individual properties of that route.

**Route Configuration:**

For 'specific' properties unique to a single route, use the prefix:

documentParserRoutes.${routeName}

For 'generic' properties covering all routes, use the prefix:

documentParserRoutes

* inputFolder - Selects which folder to consume incoming documents from
* inProgressFolder - Selects which folder files should be moved to while they are being processing
* completedFolder - Selects which folder files should be moved to after they have processing has completed
* errorFolder - Selects which folder files should be moved to if they cannot be processed due to an unrecoverable error (e.g. unsupported file format)
* idempotentRepositoryId - The Spring ID of the IdempotentRepository used by the route. This enables support for the Competing Consumers Pattern.
* inProgressRepositoryId - The Spring ID of the in-progress IdempotentRepository used by the route. This enables support for the Competing Consumers Pattern.
* processorId - The Spring ID of the parser to use when parsing documents
* outputQueue - Selects which queue to publish parsed documents to

**Folder Configuration:**

The completedFolder and errorFolder route options can include Camel Simple Language expressions. The following additional headers can be referenced:

* CamelCorrelationId - A unique ID associated with the processing of the source document
* ciaoSourceFileName - The file name of the source document
* ciaoTimestamp - The time processing was started expressed as a Unix timestamp (i.e. milliseconds since 1970)

The inProgressFolder folder option does not support Simple expressions - instead this option should be specified as a standard file path (absolute or relative to the working directory). While a document is being processed, data relating to the processing will be stored in a sub-folder of inProgressFolder/{correlationId}.

For more details of the in-progress folder structure, see the state-machine documentation from ciao-docs-finalizer.

**Hazelcast Configuration:**

The following properties are applicable for repositoryConfig=hazelcast:

* hazelcast.group.name - Name of the hazelcast cluster group
* hazelcast.group.password - Password of the hazelcast cluster group
* hazelcast.network.port - The network port to use for the hazelcast server - if the port is already in use it will be incremented until a free port is found
* hazelcast.network.publicAddress – The (optional) public address of the hazelcast node – this can be used if nodes need to communicate over NAT.
* hazelcast.network.join.tcp\_ip - Comma separated list of static cluster members - if empty, multicast join should be enabled
* hazelcast.network.join.multicast.enabled - Boolean value specifying whether multicast join should be used to find cluster members - if false, static TCP-IP members should be specified
* hazelcast.network.join.multicast.group - Multicast address to use for finding cluster members
* hazelcast.network.join.multicast.port - Multicast port to use for finding cluster members

Example configuration file:

|  |
| --- |
| # Camel logging  camel.log.mdc=true  camel.log.trace=false  camel.log.debugStreams=false  # Select which processor config to use (via dynamic spring imports)  processorConfig=default  # Select which idempotent repository config to use (via dynamic spring imports)  #repositoryConfig=memory  repositoryConfig=hazelcast  # Select which messaging config to use (via dynamic spring imports)  messagingConfig=activemq  # messagingConfig=activemq-embedded  # ActiveMQ settings (if messagingConfig=activemq)  activemq.brokerURL=tcp://localhost:61616  activemq.userName=smx  activemq.password=smx  # Hazelcast settings (if repositoryConfig=hazelcast)  hazelcast.group.name=ciao-docs-parser  hazelcast.group.password=ciao-docs-parser-pass  hazelcast.network.port=5701  hazelcast.network.publicAddress=  hazelcast.network.join.tcp\_ip.members=  hazelcast.network.join.multicast.enabled=true  hazelcast.network.join.multicast.group=224.2.2.3  hazelcast.network.join.multicast.port=54327  # Setup route names (and how many routes to build)  documentParserRoutes=discharge-notification,ed-discharge,auto-detect  # Setup 'shared' properties across all-routes  documentParserRoutes.outputQueue=parsed-documents  documentParserRoutes.inProgressFolder=./in-progress  documentParserRoutes.idempotentRepositoryId=idempotentRepository  documentParserRoutes.inProgressRepositoryId=inProgressRepository  # Setup per-route properties (can override the shared properties)  documentParserRoutes.discharge-notification.inputFolder=./input/discharge-notifications  documentParserRoutes.discharge-notification.completedFolder=./completed/discharge-notifications/${date:now:yyyy-MM-dd}/${header.CamelCorrelationId}  documentParserRoutes.discharge-notification.errorFolder=./error/discharge-notifications/${date:now:yyyy-MM-dd}/${header.CamelCorrelationId}  documentParserRoutes.discharge-notification.processorId=dischargeNotificationProcessor  documentParserRoutes.ed-discharge.inputFolder=./input/ed-discharges  documentParserRoutes.ed-discharge.completedFolder=./completed/ed-discharges/${date:now:yyyy-MM-dd}/${header.CamelCorrelationId}  documentParserRoutes.ed-discharge.errorFolder=./error/ed-discharges/${date:now:yyyy-MM-dd}/${header.CamelCorrelationId}  documentParserRoutes.ed-discharge.processorId=edDischargeProcessor  documentParserRoutes.auto-detect.inputFolder=./input/auto-detect  documentParserRoutes.auto-detect.completedFolder=./completed/auto-detect/${date:now:yyyy-MM-dd}/${header.CamelCorrelationId}  documentParserRoutes.auto-detect.errorFolder=./error/auto-detect/${date:now:yyyy-MM-dd}/${header.CamelCorrelationId}  documentParserRoutes.auto-detect.processorId=autoDetectProcessor |

## 3.4. Log Events

TODO

## 3.5. Technical Narrative

The ciao-docs-parser CIP is built from the following Java projects:

* ciao-docs-parser-core
* ciao-docs-parser-kings
* ciao-docs-parser-model
* ciao-docs-parser-parent
* ciao-docs-parser

### 3.5.1. Parsed Document

The ciao-docs-parser-model provides a model (ParsedDocument.java) for representing a binary document and set of associated properties extracted from it.

The model is used internally and as a JSON transfer representation by multiple CIPs involved in a document upload process, including ciao-docs-parser, ciao-docs-enricher, ciao-cda-builder and ciao-transport-itk.

**Java Model**

The model is defined in the following class structure:

ParsedDocument:

* originalDocument: Document
* properties: Map

Document:

* name: String
* content: byte array
* mediaType: String

The properties Map may be flat or hierarchical. If included, the metadata property should form a (flat) Map of key, value pairs.

**JSON**

CIAO uses Jackson to perform the serialization between Java and JSON. The JSON representation of ParsedDocument follows the outlined structure. All keys in the structure should be strings and the values should be encoded following the standard JSON mappings.

originalDocument.content property is encoded in JSON as a Base64 string.

Example:

{

"originalDocument": {

"name": "hello.txt",

"content": "SGVsbG8gV29ybGQh",

"mediaType": "text/plain"

},

"properties": {

"metadata": {

"some-key": "some-value"

},

"extracted-property": "extracted value"

}

}

# 4. ciao-docs-enricher

## 4.1. Description

The purpose of this CIP is to process an incoming parsed document by enriching it with additional properties before publishing the enriched document for further processing by other CIPs.

ciao-docs-enricher is built on top of Apache Camel and Spring Framework, and can be run as a stand-alone Java application, or as a Docker container.

Each instance of the CIP can host multiple routes, where each route follows the following basic structure:



The input and output queues both use the JSON-encoded representation of ParsedDocument.

The details of the JMS queues and document enrichers are specified at runtime through a combination of ciao-configuration properties and Spring XML files.

Provided document enricher implementations:

* JsonResourceDocumentEnricher - An enricher which reads JSON content from the filesystem or classpath and merges it into the document properties. This can be used to include static content which cannot be obtained from the original source document. The resource should contain a JSON-encoded representation of a ParsedDocument.

Planned future document enricher implementations:

* PDSDocumentEnricher - An enricher which performs a PDS lookup based on properties previously extracted from the source document (e.g. NHS number), and merges the retrieved detail into the document properties.

For more advanced usages, a custom document enricher can be integrated by implementing one of the enricher Java interfaces and providing a suitable spring XML configuration on the classpath.

## 4.2. Download

To pull down the source code, run:

git clone https://github.com/nhs-ciao/ciao-docs-enricher.git

You can then compile the module via:

cd ciao-docs-enricher-parent

mvn clean install -P bin-archive

This will compile a number of related modules - the main CIP module is ciao-docs-enricher, and the full binary archive (with dependencies) can be found at ciao-docs-enricher\target\ciao-docs-enricher-{version}-bin.zip. To run the CIP, unpack this zip to a directory of your choosing and follow the instructions in the README.txt.

The CIP requires access to various file system directories and network ports (dependent on the selected configuration):

**etcd:**

* Connects to: localhost:2379

**ActiveMQ:**

* Connects to: localhost:61616

**Filesystem:**

* If etcd is not available, CIAO properties will be loaded from: ~/.ciao/
* The default configuration will load JSON files for the filesystem if any file:// URLs are specified in the staticJson.resourcePaths property. This can be altered by changing the CIAO properties configuration (via etcd, or the properties file in ~/.ciao/)
* If an incoming document cannot be converted, the CIP will write an event to the folder specified by the inProgressFolder property.

**Database:**

If the lookup-data processor is used:

* Connects to the URL defined by the database.url property

Docker download and run TO DO.

## 4.3. Configuration

For further details of how ciao-configuration and Spring XML interact, please see ciao-core.

### 4.3.1. Spring XML

On application start-up, a series of Spring Framework XML files are used to construct the core application objects. The created objects include the main Camel context, input/output components, routes and any intermediate processors.

The configuration is split into multiple XML files, each covering a separate area of the application. These files are selectively included at runtime via CIAO properties, allowing alternative technologies and/or implementations to be chosen. Each imported XML file can support a different set of CIAO properties.

The Spring XML files are loaded from the classpath under the META-INF/spring package.

**Core:**

* beans.xml - The main configuration responsible for initialising properties, importing additional resources and starting Camel.

**Processors:**

* processors/include-json.xml - Creates a single JsonResourceDocumentEnricher to load static content from the classpath or filesystem
* processors/lookup-json.xml – Creates a single DynamicDocumentEnricher to load JSON content from the classpath filesystem, and dynamically select what to include based on data in the incoming document properties
* processors/lookup-database.xml – Creates a single DynamicDocumentEnricher to load key/value pairs or embedded JSON from a database, and dynamically select what to include based on data in the incoming document properties

**Messaging:**

* messaging/activemq.xml - Configures ActiveMQ as the JMS implementation for input/output queues.
* messaging/activemq-embedded.xml - Configures an internal embedded ActiveMQ as the JMS implementation for input/output queues. (For use during development/testing)

### 4.3.2. CIAO Properties

At runtime ciao-docs-enricher uses the available CIAO properties to determine which Spring XML files to load, which Camel routes to create, and how individual routes and components should be wired.

**Camel Logging:**

* camel.log.mdc - Enables/disables Mapped Diagnostic Context in Camel. If enabled, additional Camel context properties will be made available to Log4J.
* camel.log.trace - Enables/disables the Tracer interceptor for Camel routes.
* camel.log.debugStreams - Enables/disables debug logging of streaming messages in Camel.

**Spring Configuration:**

* processorConfig - Selects which processor configuration to load: processors/${processorConfig}.xml
* messagingConfig - Selects which messaging configuration to load: messaging/${messagingConfig}.xml

**Routes:**

* documentEnricherRoutes - A comma separated list of route names to build

The list of route names serves two purposes. Firstly it determines how many routes to build, and secondly each name is used as a prefix to specify the individual properties of that route.

**Route Configuration:**

For 'specific' properties unique to a single route, use the prefix: documentEnricherRoutes.${routeName}

For 'generic' properties covering all routes, use the prefix:

documentEnricherRoutes

* inputQueue - Selects which queue to consume incoming documents from
* enricherId - The Spring ID of the enricher to use when enriching documents
* outputQueue - Selects which queue to publish enriched documents to

**In-progress Folder:**

Details of the in-progress folder structure are available in the ciao-docs-finalizer state machine documentation.

ciao-docs-parser provides the InProgressFolderManagerRoute class to support storing control and event files in the in-progress directory.

* inProgressFolder - Defines the root folder that document upload process events are written to.

**Include JSON Processor​:**

These properties only apply when using:

processorConfiguration=include-json

* json.resourcePaths - A comma separated list of JSON resources to include. Spring resource loader syntax is supported, e.g. classpath:, file: etc).

**Lookup JSON Processor​:**

These properties only apply when using:

processorConfiguration=lookup-json

* json.enrichablePropertiesSelectors - comma separated list of property selectors to selects which section or sections of the document should be enriched. The selected object must be a dynamic map - if empty the root document properties are used.
* json.lookupKeySelectors - comma separated list of property selectors to select a set of key/value pairs from the document to use as lookup keys for the dynamic data.
* json.resourcePath - An optional parent path to use when selecting which JSON resource to include. Spring resource loader syntax is supported, e.g. classpath:, file: etc).
* json.resourceNameSelector - optional property selector for choosing a dynamic file name based on the lookup keys
* json.resourceSuffix - Suffix to apply to dynamically selected resource names
* json.propertySelector - Optional property selector to select a section of the JSON resource to return (using incoming lookup keys). If empty, the entire JSON structure is returned.

**Lookup Database Processor​:**

These properties only apply when using:

processorConfiguration=lookup-database

* database.url - JDBC URL used to connect to the database
* database.username - The username to connect to the database with
* database.password - The password to connect to the database with
* database.enrichablePropertiesSelectors - comma separated list of property selectors to selects which section or sections of the document should be enriched. The selected object must be a dynamic map - if empty the root document properties are used.
* database.lookupKeySelectors - comma separated list of property selectors to select a set of key/value pairs from the document to use as lookup keys for the dynamic data.
* database.sqlQuery - The select query used to find the properties. A single named parameter (of the form :?id) should form part of the WHERE clause. The document property names can be configured by using SQL aliases.
* database.idParameter - The name of the SQL parameter included in the WHERE clause
* database.idSelector - The property selector for finding ID values from the incoming lookup keys - the resulting value forms the dynamic part of the SQL WHERE clause
* database.jsonColumn - Optional name of a single returned column containing data as an embedded JSON string

**Property Selectors:**

PropertySelector is used to find source properties. Property selectors support addressing nested properties by key and index:

* nested keys: root.child
* nested arrays: root[0]
* wildcard keys: root.\*
* wildcard arrays: root[\*]

Selectors can be combined (including multiple wildcards): root[\*].child[2].\*.

Special characters [ ] . \* \ must be delimited by a \ prefix:

* D\.O\.B
* first\\last

Example configuration file:

|  |
| --- |
| # Config name/version  cip.name=ciao-docs-enricher  cip.version=1.0.0-SNAPSHOT  # Camel logging  camel.log.mdc=true  camel.log.trace=false  camel.log.debugStreams=false  # Select which processor config to use (via dynamic spring imports)  processorConfig=include-json  #processorConfig=lookup-json  #processorConfig=lookup-database  # Select which messaging config to use (via dynamic spring imports)  messagingConfig=activemq  # messagingConfig=activemq-embedded  # ActiveMQ settings (if messagingConfig=activemq)  activemq.brokerURL=tcp://localhost:61616  activemq.userName=smx  activemq.password=smx  # Setup route names (and how many routes to build)  documentEnricherRoutes=default  # Setup 'shared' properties across all-routes  documentEnricherRoutes.outputQueue=enriched-documents  # Setup per-route properties (can override the shared properties)  documentEnricherRoutes.default.enricherId=enricher  documentEnricherRoutes.default.inputQueue=parsed-documents  inProgressFolder=./in-progress  # JSON include options (if processorConfig=include-json)  json.resourcePaths=classpath:/json/extra-detail.json  # JSON lookup options (if processorConfig=lookup-json)  json.enrichablePropertiesSelectors=  json.lookupKeySelectors=documentId  json.resourcePath=classpath:/json/dynamic/  json.resourceNameSelector=documentId  json.resourceSuffix=.json  json.propertySelector=  # Database lookup options (if processorConfig=lookup-database)  database.url=jdbc:derby:memory:example;create=true  database.username=DB\_USER  database.password=DB\_PASS  database.enrichablePropertiesSelectors=  database.lookupKeySelectors=documentId  database.sqlQuery=SELECT \* FROM EXAMPLES WHERE ID = ?:id  database.idParameter=id  database.idSelector=documentId  database.jsonColumn= |

## 4.4. Log Events

TODO

## 4.5. Technical Narrative

TODO

# 5. ciao-docs-transformer

## 5.1. Description

The purpose of this CIP is to process an incoming parsed document by transforming the properties before publishing the enriched document for further processing by other CIPs.

ciao-docs-enricher is built on top of Apache Camel and Spring Framework, and can be run as a stand-alone Java application, or as a Docker container.

Each instance of the CIP can host multiple routes, where each route follows the following basic structure:



The input and output queues both use the JSON-encoded representation of ParsedDocument.

DocumentTransformer uses a delegate PropertiesTransformation instance to transform the incoming document.getProperties() map.

The details of the JMS queues and document transformers are specified at runtime through a combination of ciao-configuration properties and Spring XML files.

**Provided properties transformations:**

* PropertiesTransformer - Acts as a factory / DSL for specifying a list of specific transformations to perform
* NestedPropertiesTransformer - A special-case PropertiesTransformer which performs transformations on a child property found from the incoming properties.
* RenamePropertyTransformation - Moves / copies a property to a new name
* SplitPropertyTransformation - Splits a property value into multiple values and assigns each to a specified target property
* SplitListPropertyTransformation - Splits a property value into a list of values (using the specified pattern) and assigns the resulting list to the target property
* CombinePropertiesTransformation - Combines a series of input properties into a combined output property. The default combined value is an HTML two-column table.
* FormatDatePropertyTransformation - Parses and reformats a date property
* FindAndFormatDatePropertiesTransformation - Finds all string properties matching the specified date pattern and reformats them to use the specified output pattern

For more advanced usages, custom transformations can be integrated by implementing the PropertiesTransformation Java interface and providing a suitable spring XML configuration on the classpath.

**Property Selectors:**

The provided properties transformations use PropertyName to identify source and target properties.

Property names support addressing nested properties by key and index:

* nested keys: root.child
* nested arrays: root[0]

Names can be combined: root.child[2].name.firstName.

Special characters [ ] . \* \ must be delimited by a \ prefix:

* D\.O\.B
* first\\last

## 5.2. Download

To pull down the source code, run:

git clone https://github.com/nhs-ciao/ciao-docs-transformer.git

You can then compile the module via:

mvn clean install -P bin-archive

This will compile a number of related modules - the main CIP module is ciao-docs-transformer, and the full binary archive (with dependencies) can be found at target\ciao-docs-transformer-{version}-bin.zip. To run the CIP, unpack this zip to a directory of your choosing and follow the instructions in the README.txt.

The CIP requires access to various file system directories and network ports (dependent on the selected configuration):

**etcd:**

* Connects to: localhost:2379

**ActiveMQ:**

* Connects to: localhost:61616

**Filesystem:**

* If etcd is not available, CIAO properties will be loaded from: ~/.ciao/
* The default configuration will load JSON files for the filesystem if any file:// URLs are specified in the staticJson.resourcePaths property. This can be altered by changing the CIAO properties configuration (via etcd, or the properties file in ~/.ciao/)
* If an incoming document cannot be converted, the CIP will write an event to the folder specified by the inProgressFolder property.

Docker download and run TO DO.

## 5.3. Configuration

For further details of how ciao-configuration and Spring XML interact, please see ciao-core.

### 5.3.1. Spring XML

On application start-up, a series of Spring Framework XML files are used to construct the core application objects. The created objects include the main Camel context, input/output components, routes and any intermediate processors.

The configuration is split into multiple XML files, each covering a separate area of the application. These files are selectively included at runtime via CIAO properties, allowing alternative technologies and/or implementations to be chosen. Each imported XML file can support a different set of CIAO properties.

The Spring XML files are loaded from the classpath under the META-INF/spring package.

**Core:**

* beans.xml - The main configuration responsible for initialising properties, importing additional resources and starting Camel.

**Processors:**

* processors/default.xml - Creates the following DocumentTransformer instances: kingsWordDischargeNotificationTransformer and kentEDNTransformer

**Messaging:**

* messaging/activemq.xml - Configures ActiveMQ as the JMS implementation for input/output queues.
* messaging/activemq-embedded.xml - Configures an internal embedded ActiveMQ as the JMS implementation for input/output queues. (For use during development/testing)

### 5.3.2. CIAO Properties

At runtime ciao-docs-transformer uses the available CIAO properties to determine which Spring XML files to load, which Camel routes to create, and how individual routes and components should be wired.

**Camel Logging:**

* camel.log.mdc - Enables/disables Mapped Diagnostic Context in Camel. If enabled, additional Camel context properties will be made available to Log4J.
* camel.log.trace - Enables/disables the Tracer interceptor for Camel routes.
* camel.log.debugStreams - Enables/disables debug logging of streaming messages in Camel.

**Spring Configuration:**

* processorConfig - Selects which processor configuration to load: processors/${processorConfig}.xml
* messagingConfig - Selects which messaging configuration to load: messaging/${messagingConfig}.xml

**Routes:**

* documentTransformerRoutes - A comma separated list of route names to build

The list of route names serves two purposes. Firstly it determines how many routes to build, and secondly each name is used as a prefix to specify the individual properties of that route.

**Route Configuration:**

For 'specific' properties unique to a single route, use the prefix: documentTransformerRoutes.${routeName}

For 'generic' properties covering all routes, use the prefix:

documentTransformerRoutes

* inputQueue - Selects which queue to consume incoming documents from
* transformerId - The Spring ID of the transformer to use when enriching documents
* outputQueue - Selects which queue to publish enriched documents to

**In-progress Folder:**

Details of the in-progress folder structure are available in the ciao-docs-finalizer state machine documentation.

ciao-docs-parser provides the InProgressFolderManagerRoute class to support storing control and event files in the in-progress directory.

* inProgressFolder - Defines the root folder that document upload process events are written to.

Example configuration file:

|  |
| --- |
| # Camel logging  camel.log.mdc=true  camel.log.trace=false  camel.log.debugStreams=false  # Select which processor config to use (via dynamic spring imports)  processorConfig=default  # Select which messaging config to use (via dynamic spring imports)  messagingConfig=activemq  # messagingConfig=activemq-embedded  # ActiveMQ settings (if messagingConfig=activemq)  activemq.brokerURL=tcp://localhost:61616  activemq.userName=smx  activemq.password=smx  # Setup route names (and how many routes to build)  documentTransformerRoutes=kings,kent  # Setup 'shared' properties across all-routes  documentTransformerRoutes.outputQueue=transformed-documents  # Setup per-route properties (can override the shared properties)  documentTransformerRoutes.kings.inputQueue=parsed-kings-documents  documentTransformerRoutes.kings.transformerId=kingsWordDischargeNotificationTransformer  documentTransformerRoutes.kent.inputQueue=parsed-kent-documents  documentTransformerRoutes.kent.transformerId=kentEDNTransformer  # Global properties  inProgressFolder=./in-progress |

## 5.4. Log Events

TODO

## 5.5. Technical Narrative

TODO

# 6. ciao-cda-builder

## 6.1. Description

The purpose of this CIP is to convert an incoming parsed document into one of the ITK-specified CDA document formats.

ciao-cda-builder is built on top of Apache Camel and Spring Framework, and can be run as a stand-alone Java application, or as a Docker container. The ITK Payloads library is used to build the outgoing CDA documents.

Each instance of the CIP can host multiple routes, where each route follows the following basic structure:



* The input and output queues both use the JSON-encoded representation of ParsedDocument. In the output queue, ParsedDocument.originalDocument contains the constructed CDA document.

The details of the JMS queues and document enrichers are specified at runtime through a combination of ciao-configuration properties and Spring XML files.

Supported CDA types:

Incoming documents can specify which type of CDA document to build by include the corresponding interaction id using the itkHandlingSpec property on the incoming document.

|  |  |  |
| --- | --- | --- |
| **Specification** | **Input Format** | **Interaction ID** |
| Transfer of care | transfer-of-care-json | urn:nhs-itk:interaction:primaryRecipienteDischargeInpatientDischargeSummaryDocument-v1-0 |

## 6.2. Download

To pull down the code, run:

git clone https://github.com/nhs-ciao/ciao-cda-builder.git

You can then compile the module via:

mvn clean install -P bin-archive

This will compile a number of related modules - the main CIP module is ciao-cda-builder, and the full binary archive (with dependencies) can be found at target\ciao-cda-builder-{version}-bin.zip. To run the CIP, unpack this zip to a directory of your choosing and follow the instructions in the README.txt.

The CIP requires access to various file system directories and network ports (dependent on the selected configuration):

**etcd:**

* Connects to: localhost:2379

**ActiveMQ:**

* Connects to: localhost:61616

**Filesystem:**

* If etcd is not available, CIAO properties will be loaded from: ~/.ciao/
* If an incoming document cannot be converted, the CIP will write an event to the folder specified by the inProgressFolder property.

Docker download and run TODO.

## 6.3. Configuration

For further details of how ciao-configuration and Spring XML interact, please see ciao-core.

### 6.3.1 Spring XML

On application start-up, a series of Spring Framework XML files are used to construct the core application objects. The created objects include the main Camel context, input/output components, routes and any intermediate processors.

The configuration is split into multiple XML files, each covering a separate area of the application. These files are selectively included at runtime via CIAO properties, allowing alternative technologies and/or implementations to be chosen. Each imported XML file can support a different set of CIAO properties.

The Spring XML files are loaded from the classpath under the META-INF/spring package.

**Core:**

* beans.xml - The main configuration responsible for initialising properties, importing additional resources and starting Camel.

**Processors:**

* processors/default.xml - Creates a single JsonToCDADocumentTransformer to convert incoming documents into CDA format.

**Messaging:**

* messaging/activemq.xml - Configures ActiveMQ as the JMS implementation for input/output queues.
* messaging/activemq-embedded.xml - Configures an internal embedded ActiveMQ as the JMS implementation for input/output queues. (For use during development/testing)

### 6.3.2. CIAO Properties

At runtime ciao-cda-builder uses the available CIAO properties to determine which Spring XML files to load, which Camel routes to create, and how individual routes and components should be wired.

**Camel Logging:**

* camel.log.mdc - Enables/disables Mapped Diagnostic Context in Camel. If enabled, additional Camel context properties will be made available to Log4J and Logstash.
* camel.log.trace - Enables/disables the Tracer interceptor for Camel routes.
* camel.log.debugStreams - Enables/disables debug logging of streaming messages in Camel.

**Spring Configuration:**

* processorConfig - Selects which processor configuration to load: processors/${processorConfig}.xml
* messagingConfig - Selects which messaging configuration to load: messaging/${messagingConfig}.xml

**Routes:**

* cdaBuilderRoutes - A comma separated list of route names to build

The list of route names serves two purposes. Firstly it determines how many routes to build, and secondly each name is used as a prefix to specify the individual properties of that route.

**Route Configuration:**

For 'specific' properties unique to a single route, use the prefix: cdaBuilderRoutes.${routeName}

For 'generic' properties covering all routes, use the prefix:

cdaBuilderRoutes

* inputQueue - Selects which queue to consume incoming documents from
* processorId - The Spring ID of the processor to use when converting documents
* outputQueue - Selects which queue to publish the constructed CDA documents to

**In-progress Folder:**

Details of the in-progress folder structure are available in the ciao-docs-finalizer state machine documentation.

ciao-docs-parser provides the InProgressFolderManagerRoute class to support storing control and event files in the in-progress directory.

* inProgressFolder - Defines the root folder that document upload process events are written to.

**Default Processor​:**

The default processor configuration does not currently support any additional properties.

Example configuration file:

|  |
| --- |
| # Camel logging  camel.log.mdc=true  camel.log.trace=false  camel.log.debugStreams=false  # Select which processor config to use (via dynamic spring imports)  processorConfig=default  # Select which messaging config to use (via dynamic spring imports)  messagingConfig=activemq  #messagingConfig=activemq-embedded  # ActiveMQ settings (if messagingConfig=activemq)  activemq.brokerURL=tcp://localhost:61616  activemq.userName=smx  activemq.password=smx  # Setup route names (and how many routes to build)  cdaBuilderRoutes=default  # Setup 'shared' properties across all-routes  cdaBuilderRoutes.outputQueue=cda-documents  cdaBuilderRoutes.processorId=processor  # Setup per-route properties (can override the shared properties)  cdaBuilderRoutes.default.inputQueue=enriched-documents  inProgressFolder=./in-progress |

## 6.4. Log Events

TODO

## 6.5. Technical Narrative

TODO

# 7. ciao-transport-spine

## 7.1. Description

Transferring documents using the Interoperability Toolkit (ITK) specifications results in a layered system of:

* Business Message Layer - e.g. CDA documents
* ITK Layer - Distribution Envelope and Acknowledgement Framework
* Transport Layer - e.g. Spine or DTS/MESH

This CIP includes the ITK Layer (by importing ciao-transport-itk) and the Transport Layer by providing Spine-specific functionality.

## 7.2. Download

To pull down the code, run:

git clone https://github.com/nhs-ciao/ciao-transport-itk.git

You can then compile the module via:

Cd ciao-transport-itk-parent

mvn clean install -P bin-archive

This will compile a number of related modules - the main CIP module for Spine is ciao-transport-spine, and the full binary archive (with dependencies) can be found at ciao-transport-spine\target\ciao-transport-spine-{version}-bin.zip. To run the CIP, unpack this zip to a directory of your choosing and follow the instructions in the README.txt.

The CIP requires access to various file system directories and network ports (dependent on the selected configuration):

**etcd:**

* Connects to: localhost:2379

**ActiveMQ:**

* Connects to: localhost:61616

**Hazelcast:**

* Multicast discovery: 224.2.2.3:54327 (If enabled)
* Listens on: \*:5701 (If port is already taken, the port number is incremented until a free port is found)
* Connects to clustered nodes defined by the hazelcast.network.join.tcp\_ip.members property

**Spine:**

* Connects to the HTTP/HTTPS server specified by spine.toUri
* Connects to the LDAP server specified by sds.url
* Listens to the interface/port specified by spine.fromUri

**Filesystem:**

* If etcd is not available, CIAO properties will be loaded from: ~/.ciao/
* For key events in the document upload lifecycle, the CIP will write an event to the folder specified by the inProgressFolder property.
* If static addresses are enabled, the files specified by the addressing.staticFiles property will be read from the file system. Relative paths are resolved relative to CIP working directory.
* If SSL/TLS is enabled, the files specified by the TRUST\_STORE and KEY\_STORE properties will be read from the file system.

Docker download and run TODO.

## 7.3. Configuration

For further details of how ciao-configuration and Spring XML interact, please see ciao-core.

### 7.3.1. Spring XML

On application start-up, a series of Spring Framework XML files are used to construct the core application objects. The created objects include the main Camel context, input/output components, routes and any intermediate processors.

The configuration is split into multiple XML files, each covering a separate area of the application. These files are selectively included at runtime via CIAO properties, allowing alternative technologies and/or implementations to be chosen. Each imported XML file can support a different set of CIAO properties.

The Spring XML files are loaded from the classpath under the META-INF/spring package.

**Core:**

* beans.xml - The main configuration responsible for initialising properties, importing additional resources and starting Camel.

**Repositories:**

An `IdempotentRepository' is configured to enable multiple consumers access the same folder concurrently.

* repository/memory.xml - An in-memory implementation suitable for use when there is only a single consumer, or multiple-consumers are all contained within the same JVM instance.
* repository/hazelcast.xml - A grid-based implementation backed by Hazelcast. The component is hosted entirely within the JVM process and uses a combination of multicast and point-to-point networking to maintain a cross-server data grid.

**Processors:**

* processors/default.xml - Currently a NOOP

**Messaging:**

* messaging/activemq.xml - Configures ActiveMQ as the JMS implementation for input/output queues.
* messaging/activemq-embedded.xml - Configures an internal embedded ActiveMQ as the JMS implementation for input/output queues. (For use during development/testing)

**Addressing:**

* addressing/static.xml - Configures the CIP to resolve ITK to Spine addresses using the static values defined via the staticJson.resourcePaths property.
* addressing/sds.xml - Configures the CIP to resolve ITK to Spine addresses using the Spine Directory Service (SDS). Additionally address caching (via Hazelcast) is enabled, and static values can also be supplied via the staticJson.resourcePaths property.

**Spine SSL:**

* ssl/vanilla.xml - Configures the CIP to use standard non-TLS sockets for connecting to Spine.
* ssl/tls.xml - Configures the CIP to use TLS-enabled sockets for connecting to Spine. This configuration requires a key store and a trust store to be specified via the KEY\_STORE and TRUST\_STORE properties.

### 7.3.2. CIAO Properties

At runtime ciao-transport-spine uses the available CIAO properties to determine which Spring XML files to load, which Camel routes to create, and how individual routes and components should be wired.

**Camel Logging:**

* camel.log.mdc - Enables/disables Mapped Diagnostic Context in Camel. If enabled, additional Camel context properties will be made available to Log4J and Logstash.
* camel.log.trace - Enables/disables the Tracer interceptor for Camel routes.
* camel.log.debugStreams - Enables/disables debug logging of streaming messages in Camel.

**Spring Configuration:**

* repositoryConfig - Selects which repository configuration to load: repositories/${repositoryConfig}.xml
* processorConfig - Selects which processor configuration to load: processors/${processorConfig}.xml
* messagingConfig - Selects which messaging configuration to load: messaging/${messagingConfig}.xml
* addressingConfig - Selects which addressing configuration to load: addressing/${addressingConfig}.xml
* sslConfig - Selects which SSL configuration to load: ssl/${sslConfig}.xml

**Spine Configuration:**

* spine.toUri - URI for sending outgoing messages to Spine.
* spine.fromUri - URI of HTTP/HTTPS server for receiving incoming messages from Spine.
* spine.replyUri - URI of JMS topic for processing asynchronous Spine ebXml acknowledgements.
* sds.url - URI of the SDS LDAP server.
* sds.authentication - Type of LDAP authentication used when connecting to SDS.
* sds.principal - LDAP principal / user used when connecting to SDS.
* sds.credentials - LDAP credentials / password used when connecting to SDS.

Spine and SDS connection URIs depend on the selected SSL configuration. If TLS is enabled, then the URIs should include the https scheme, otherwise http should be used.

**Distribution Envelope Configuration:**

* senderItkService - The ITK service added to outgoing distribution envelopes
* senderODSCode - The sender ODS code added to outgoing distribution envelopes
* auditODSCode - The audit ODS code added to outgoing distribution envelopes (if this property is not defined, senderODSCode is used).

**EbXml/HL7 Configuration:**

* senderService - The ebXml service added to outgoing ebXml messages and SOAPAction headers
* senderAction - The ebXml action added to outgoing ebXml messages and SOAPAction headers
* senderPartyId - The sender PartyId added to outgoing ebXml messages
* senderAsid - The sender ASID added to outgoing HL7 messages

**Queue Configuration:**

* itkDocumentSenderQueue - JMS queue for processing outgoing ITK documents
* multipartMessageSenderQueue - JMS queue for processing outgoing Spine multipart messages
* multipartMessageResponseQueue - JMS queue for processing incoming Spine multipart messages
* distributionEnvelopeReceiverQueue - JMS queue for processing incoming ITK Distribution Envelopes
* itkMessageReceiverQueue - JMS queue for processing incoming ITK messages

The main input queue (itkDocumentSenderQueue) uses the JSON-encoded representation of ParsedDocument. The originalDocument property contains the document to send, while the properties section contains additional details such as the recipient ODS code.

**Address Resolution Configuration:**

* addressing.staticFiles - A comma-separated list of static files which provide static JSON-encoded SpineEndpointAddress values.
* addressing.sdsCacheUri - Defines the Hazelcast distributed map used to cache resolved endpoint addresses.

Configuration of the cache (e.g. time to live, cache size) is specified in the repositories\hazelcast.xml spring file.

**In-progress Folder:**

Details of the in-progress folder structure are available in the ciao-docs-finalizer state machine documentation.

ciao-docs-parser provides the InProgressFolderManagerRoute class to support storing control and event files in the in-progress directory.

* inProgressFolder - Defines the root folder that document upload process events are written to.

**Hazelcast Configuration:**

The following properties are applicable for repositoryConfig=hazelcast:

* hazelcast.group.name - Name of the hazelcast cluster group
* hazelcast.group.password - Password of the hazelcast cluster group
* hazelcast.network.port - The network port to use for the hazelcast server - if the port is already in use it will be incremented until a free port is found
* hazelcast.network.publicAddress – The (optional) public address of the hazelcast node - this can be used if nodes need to communicate over NAT.
* hazelcast.network.join.tcp\_ip - Comma separated list of static cluster members - if empty, multicast join should be enabled
* hazelcast.network.join.multicast.enabled - Boolean value specifying whether multicast join should be used to find cluster members - if false, static TCP-IP members should be specified
* hazelcast.network.join.multicast.group - Multicast address to use for finding cluster members
* hazelcast.network.join.multicast.port - Multicast port to use for finding cluster members

Example configuration file:

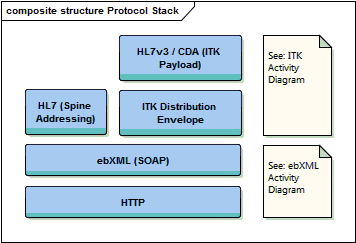
|  |
| --- |
| # Camel logging  camel.log.mdc=true  camel.log.trace=false  camel.log.debugStreams=false  # Select which processor config to use (via dynamic spring imports)  processorConfig=default  # Select which idempotent repository config to use (via dynamic spring imports)  repositoryConfig=hazelcast  # repositoryConfig=memory  # Select which messaging config to use (via dynamic spring imports)  messagingConfig=activemq  #messagingConfig=activemq-embedded  # Select which addressing config to use (via dynamic spring imports)  addressingConfig=static  #addressingConfig=sds  # Select which ssl config to use (via dynamic spring imports)  sslConfig=vanilla  # ActiveMQ settings (if messagingConfig=activemq)  activemq.brokerURL=tcp://localhost:61616  activemq.userName=smx  activemq.password=smx  spine.toUri=http://localhost:8123/  spine.fromUri=jetty:http://localhost:8122/  spine.replyUri=jms2:topic:document-ebxml-acks  # Spine SSL settings  TRUST\_STORE=/opt/keystores/SpineDEVCerts.keystore  TRUST\_STORE\_PW=password  KEY\_STORE=/opt/keystores/SpineCiaoTest1.keystore  KEY\_STORE\_PW=password  KEY\_PASSWORD=password  # Spine SDS settings  sds.url=ldap://localhost:1234  sds.authentication=simple  sds.principal=cn=Manager,dc=example,dc=com  sds.credentials=passw0rd  # Common JMS/ActiveMQ settings (if messagingConfig=activemq or activemq-embedded)  jms.concurrentConsumers=20  jms2.concurrentConsumers=2  # Hazelcast settings (if repositoryConfig=hazelcast)  hazelcast.group.name=ciao-transport-spine  hazelcast.group.password=ciao-transport-spine-pass  hazelcast.network.port=5701  hazelcast.network.publicAddress=  hazelcast.network.join.tcp\_ip.members=  hazelcast.network.join.multicast.enabled=true  hazelcast.network.join.multicast.group=224.2.2.3  hazelcast.network.join.multicast.port=54327  # Common addressing settings (if addressingConfig=static or sds)  addressing.staticFiles=  # SDS addressing settings (if addressingConfig=sds)  addressing.sdsCacheUri=hazelcast:map:spine-endpoint-addresses  senderPartyId=!REQUIRED!  senderAsid=!REQUIRED!  senderODSCode=!REQUIRED!  # auditODSCode=optional - defaults to senderODSCode  senderItkService=urn:nhs-itk:services:201005:sendDistEnvelope  senderService=urn:nhs:names:services:itk  senderAction=COPC\_IN000001GB01  itkDocumentSenderQueue=cda-documents  multipartMessageSenderQueue=multipart-message-sender  multipartMessageResponseQueue=multipart-message-responses  distributionEnvelopeReceiverQueue=distribution-envelope-receiver  itkMessageReceiverQueue=itk-message-receiver  inProgressFolder=./in-progress |

## 7.4. Log Events

TODO

## 7.5. Technical Narrative

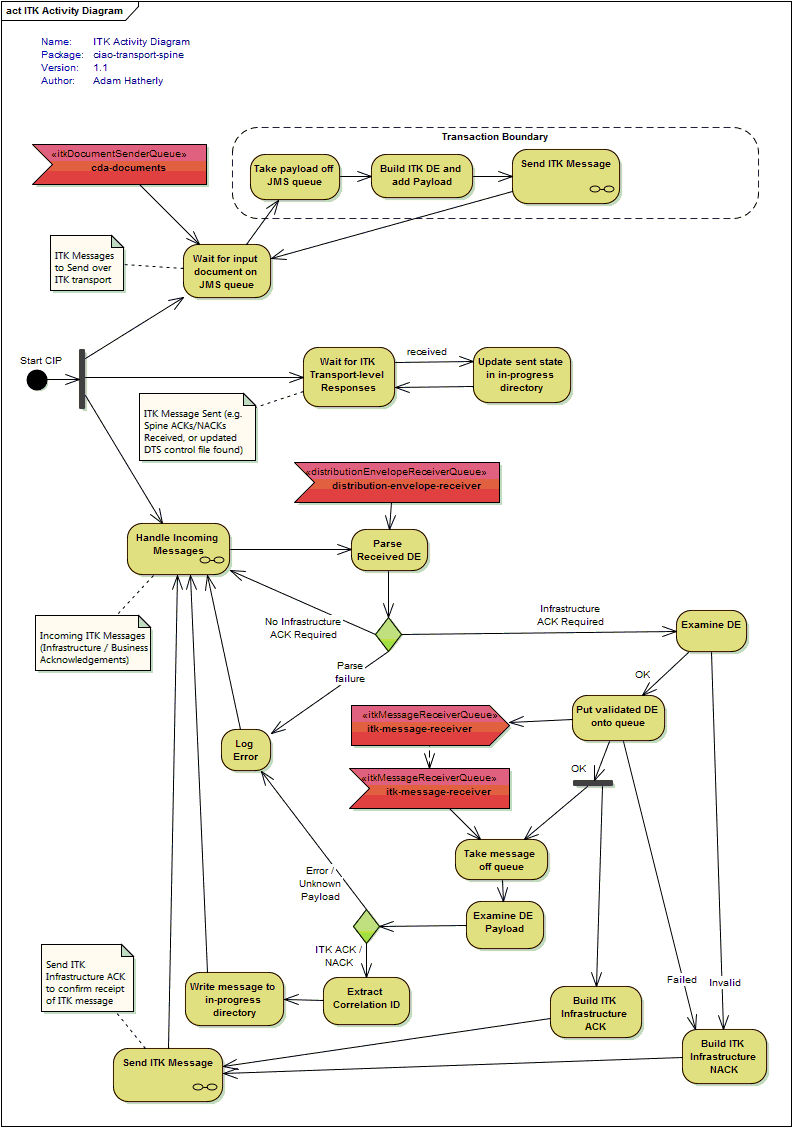
In order to understand how this CIP sends documents over the Spine, we first need to understand the different protocols in use:



If we were to consider the full set of activities to send a document over the Spine using the below protocols, the process would appear quite complex. By splitting the activities into the two main layers (ITK and ebXML), the processing required is much easier to understand.

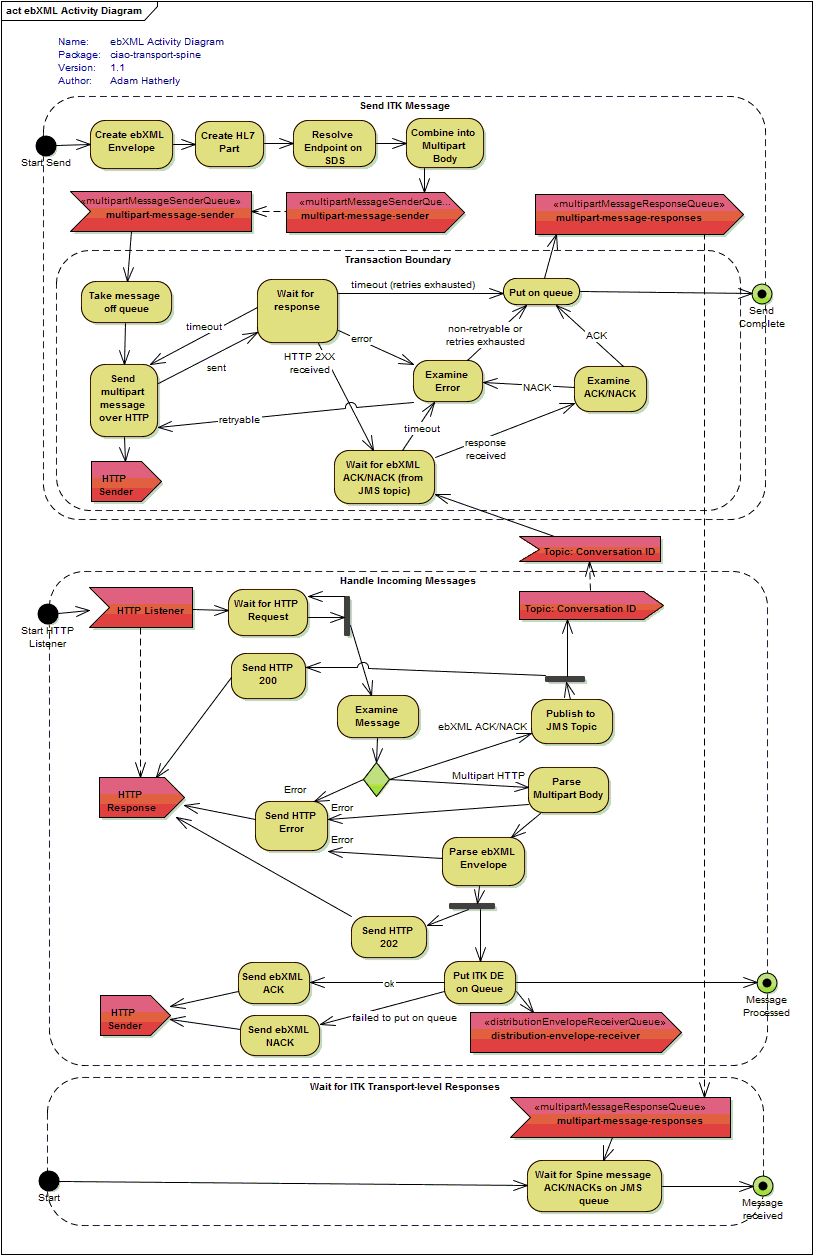
The ITK layer includes the basic sending, coupled with a simple acknowledgement framework which allows the sender to request an Infrastructure Acknowledgement (to confirm the ITK message has reached its destination), and a Business Acknowledgement (to confirm "business receipt" - the exact meaning of which is specific to the type of document being sent).

The below activity diagram shows the high level activities involved in processing the overall message and ITK-layer interactions:



Some of the individual boxes on the ITK diagram above represent the lower level ebXML interactions - these lower level protocol activities are shown in the below ebXML activity diagram.

The ebXML layer deals with sending these higher level ITK interactions over the Spine. It uses a multipart HTTP message as defined in the ebXML specification, and each individual ebXML message is Asynchronously acknowledged with a corresponding ebXML Acknowledgement (or error).



**Models**

Several message types are used when sending messages over Spine and acknowledging receipt. The ciao-transport-spine CIP provides Java representations of these types, along with serialization to/from XML, and integration with Apache Camel's type conversion system.

**EbXml**

During part of the message flow, the ebXml message is used to describe one or more payloads, along with details of the sender, intended receiver, and a manifest describing the payload. Later in the message flow, an asynchronous ebXml message is used to acknowledge receipt (or failure).

ebXml provides extension elements to a SOAP XML message.

**Java Classes:**

* EbxmlEnvelope provides a bean-like representation of a ebXml envelope.
* EbxmlEnvelopeParser parses an XML serialized ebXml envelope to object form.
* EbxmlEnvelopeSerializer - serializes an ebXml envelope object into XML.
* EbxmlEnvelopeTypeConverter - Integrates the ebXml envelope parser and serializer with Camel.

**Creating, parsing and serializing ebXml envelopes:**

|  |
| --- |
| // Serializer/parser configuration (reusable objects)  EbxmlEnvelopeParser parser = new EbxmlEnvelopeParser();  EbxmlEnvelopeSerializer serializer = new EbxmlEnvelopeSerializer();  // Creating an envelope  EbxmlEnvelope prototype = new EbxmlEnvelope();  prototype.setAction("action");  prototype.setService("service");  prototype.setFromParty("from-party");  prototype.setAckRequested(true);  // Applying default values for non-specified fields (e.g. timestamp)  prototype.applyDefaults();  // Parsing an envelope  InputStream in = new FileInputStream("example-envelope.xml");  EbxmlEnvelope envelope = parser.parse(in);  // Merging / copying properties between envelopes  boolean overwrite = false;  envelope.copyFrom(prototype, overwrite);  // Serializing an envelope  String xml = serializer.serialize(envelope); |

**Camel type conversion:**

|  |
| --- |
| public class ExampleRoute extends RouteBuilder {  @Override  public void configure() throws Exception {  from("jms:queue:input")  // the ebxml type converter is automatically registered in Camel  .convertBodyTo(EbxmlEnvelope.class)  .log("Found ebxml envelope from ${body.fromParty} with messageId: ${body.messageData.messageId}")  // Use / modify the envelope in some way  .bean(new EbxmlEnvelopeProcessor())  // Serialize the updated envelope as XML  .convertBodyTo(String.class)  .log("Converted the envelope to XML: ${body}")  .end();  }  } |

**HL7**

In multipart messages, an HL7 part is included providing details of the sender/receiver ASIDs for Spine.

This message type/part only applies to outgoing multi-part messages - not asynchronous acknowledgements.

**Java Classes:**

* HL7Part provides a bean-like representation of a HL7 message part.
* HL7PartParser parses an XML serialized HL7 message part to object form.
* HL7PartSerializer - serializes an HL7 message part object into XML.
* HL7PartTypeConverter - Integrates the HL7 message part parser and serializer with Camel.

**Creating, parsing and serializing HL7 parts:**

|  |
| --- |
| // Serializer/parser configuration (reusable objects)  HL7PartParser parser = new HL7PartParser();  HL7PartSerializer serializer = new HL7PartSerializer();  // Creating an HL7 part  HL7Part prototype = new HL7Part();  prototype.setInteractionId("interaction-id");  prototype.setSenderAsid("sender-asid");  // Applying default values for non-specified fields (e.g. timestamp)  prototype.applyDefaults();  // Parsing an HL7 part  InputStream in = new FileInputStream("example-part.xml");  HL7Part part = parser.parse(in);  // Merging / copying properties between HL7 parts  boolean overwrite = false;  part.copyFrom(prototype, overwrite);  // Serializing an HL7 part  String xml = serializer.serialize(part); |

**Camel type conversion:**

|  |
| --- |
| public class ExampleRoute extends RouteBuilder {  @Override  public void configure() throws Exception {  from("jms:queue:input")  // the HL7 type converter is automatically registered in Camel  .convertBodyTo(HL7Part.class)  .log("Found HL7 part from ${body.senderAsid} with interactionId: ${body.interactionId}")  // Use / modify the part in some way  .bean(new HL7PartProcessor())  // Serialize the updated part as XML  .convertBodyTo(String.class)  .log("Converted the HL7 part to XML: ${body}")  .end();  }  } |

**Multipart Message**

Spine uses the Multipart/Related Content-Type to bundle multiple message parts into a single message representation. The bundled message typically includes an ebXml part, an HL7 part, and a payload (which for itk-transport-spine is an ITK Distribution Envelope).

**Java Classes:**

* MultipartBody provides a bean-like representation of a multipart message.
* Part provides a bean-like representation of an individual part of a multipart message. Additionally this class integrates with Camel's Message interface.
* MultipartParser - parses multipart messages between object and serialized text form.
* MultipartTypeConverter - Integrates the multipart parser with Camel.

**Creating, parsing and serializing multipart messages:**

|  |
| --- |
| // Serializer/parser configuration (reusable objects)  MultipartParser parser = new MultipartParser();  // Creating a multi-part body  MultipartBody body = new MultipartBody();  // Creating and adding parts to the body  Part part = new Part();  part.setContentType("text/plain");  part.setBody("content of the first part\n");  body.addPart(part);  // A short-hand way of adding a part  body.addPart("text/xml", "<root>content of the second part</root>\n");  // Serializing a multi-part body  FileOutputStream out = new FileOutputStream("multipart.xml");  body.write(out);  // Parsing a multi-part body  // The content type (with boundary) is typically provided as a message header (e.g. via HTTP)  String contentType = "multipart/related; boundary=\_\_some-boundary\_\_";  InputStream in = new FileInputStream("example-multipart.xml");  body = parser.parse(contentType, in);  // Accessing a part (when the id is known)  part = body.findPartByContentId("1234567"); |

**Camel type conversion:**

|  |
| --- |
| public class ExampleRoute extends RouteBuilder {  @Override  public void configure() throws Exception {  from("jms:queue:input")  // the multipart type converter is automatically registered in Camel  .convertBodyTo(MultipartBody.class)  .log("Found multipart body with ${body.getParts.size} parts - boundary: ${body.boundary}")  // Use the first part  .setProperty("multipart-body").body()  .setBody().spel("#{body.parts[0].body}")  .convertBodyTo(EbxmlEnvelope.class) // example part conversion  .log("First part of multipart body: ${body}")  // Serialize the body as a string  .setBody().property("multipart-body")  .convertBodyTo(String.class)  .log("Converted the multipart body to XML: ${body}")  .end();  }  } |

Part is actually a subclass of Camel's DefaultMessage - allowing Part instances to be integrated easily in Camel routes. However, for the type conversion to operate as expected part.setExchange(exchange) should be called to ensure the type converter has access to the current CamelContext. The MultipartParser automatically handles this for parsed objects (when the appropriate method is used), and instances converted inside a Camel route will have the exchange set. However, manually created objects will need the exchange to be provided.

**Spine Directory Service**

ciao-transport-spine can handle ITK to Spine address resolution by querying the Spine Directory Service (SDS).

The SDS LDAP directory contains many object classes, only some of which are relevant to this CIP. The ciao-spine-sds library provides Java models for these classes, and an associated query-builder layer for finding matching objects in the LDAP tree.

# 8. ciao-transport-dts

## 8.1. Description

Transferring documents using the Interoperability Toolkit (ITK) specifications results in a layered system of:

* Business Message Layer - e.g. CDA documents
* ITK Layer - Distribution Envelope and Acknowledgement Framework
* Transport Layer - e.g. Spine or DTS/MESH

This CIP includes the ITK Layer (by importing ciao-transport-itk) and the Transport Layer by providing DTS/MESH-specific functionality.

## 8.2. Download

To pull down the code, run:

git clone https://github.com/nhs-ciao/ciao-transport-itk.git

You can then compile the module via:

Cd ciao-transport-itk-parent

mvn clean install -P bin-archive

This will compile a number of related modules - the main CIP module for DTS/MESH is ciao-transport-dts, and the full binary archive (with dependencies) can be found at ciao-transport-dts\target\ciao-transport-dts-{version}-bin.zip. To run the CIP, unpack this zip to a directory of your choosing and follow the instructions in the README.txt.

The CIP requires access to various file system directories and network ports (dependent on the selected configuration):

**etcd:**

* Connects to: localhost:2379

**ActiveMQ:**

* Connects to: localhost:61616

**Hazelcast:**

* Multicast discovery: 224.2.2.3:54327 (If enabled)
* Listens on: \*:5701 (If port is already taken, the port number is incremented until a free port is found)
* Connects to clustered nodes defined by the hazelcast.network.join.tcp\_ip.members property

**DTS:**

* Files are read from / written to the main DTS folder structure defined by the dts.rootFolder property.
* Additional directories (relative to the main DTS folder) defined by the dts.temporaryFolder, dts.errorFolder, and dts.completedFolder properties will also be used.

**Filesystem:**

* If etcd is not available, CIAO properties will be loaded from: ~/.ciao/
* For key events in the document upload lifecycle, the CIP will write an event to the folder specified by the inProgressFolder property.
* If static addresses are enabled, the files specified by the addressing.staticFiles property will be read from the file system. Relative paths are resolved relative to CIP working directory.

Docker download and run TODO.

## 8.3. Configuration

For further details of how ciao-configuration and Spring XML interact, please see ciao-core.

### 8.3.1. Spring XML

On application start-up, a series of Spring Framework XML files are used to construct the core application objects. The created objects include the main Camel context, input/output components, routes and any intermediate processors.

The configuration is split into multiple XML files, each covering a separate area of the application. These files are selectively included at runtime via CIAO properties, allowing alternative technologies and/or implementations to be chosen. Each imported XML file can support a different set of CIAO properties.

The Spring XML files are loaded from the classpath under the META-INF/spring package.

**Core:**

* beans.xml - The main configuration responsible for initialising properties, importing additional resources and starting Camel.

**Repositories:**

An `IdempotentRepository' is configured to enable multiple consumers access the same folder concurrently.

* repository/memory.xml - An in-memory implementation suitable for use when there is only a single consumer, or multiple-consumers are all contained within the same JVM instance.
* repository/hazelcast.xml - A grid-based implementation backed by Hazelcast. The component is hosted entirely within the JVM process and uses a combination of multicast and point-to-point networking to maintain a cross-server data grid.

**Processors:**

* processors/default.xml - Currently a NOOP

**Messaging:**

* messaging/activemq.xml - Configures ActiveMQ as the JMS implementation for input/output queues.
* messaging/activemq-embedded.xml - Configures an internal embedded ActiveMQ as the JMS implementation for input/output queues. (For use during development/testing)

**Addressing:**

* addressing/static.xml - Configures the CIP to resolve ITK to DTS addresses using the static values defined via the staticJson.resourcePaths property.

### 8.3.2. CIAO Properties

At runtime ciao-transport-spine uses the available CIAO properties to determine which Spring XML files to load, which Camel routes to create, and how individual routes and components should be wired.

**Camel Logging:**

* camel.log.mdc - Enables/disables Mapped Diagnostic Context in Camel. If enabled, additional Camel context properties will be made available to Log4J and Logstash.
* camel.log.trace - Enables/disables the Tracer interceptor for Camel routes.
* camel.log.debugStreams - Enables/disables debug logging of streaming messages in Camel.

**Spring Configuration:**

* repositoryConfig - Selects which repository configuration to load: repositories/${repositoryConfig}.xml
* processorConfig - Selects which processor configuration to load: processors/${processorConfig}.xml
* messagingConfig - Selects which messaging configuration to load: messaging/${messagingConfig}.xml
* addressingConfig - Selects which addressing configuration to load: addressing/${addressingConfig}.xml

**DTS Configuration:**

* dts.rootFolder - The root folder of the DTS file structure
* dts.temporaryFolder - Temporary folder used to initially write files into before moving them into the ${dts.rootFolder}/OUT folder.
* dts.filePrefix - An optional filename prefix added to all outgoing files in the ${dts.rootFolder}/OUT folder. This can be used to distinguish which application added the file and to filter the corresponding sent notifications from the ${dts.rootFolder}/SENT folder. This property is useful in deployments where multiple applications share the same DTS mailbox.
* dts.errorFolder - Folder where incoming control/data file pairs are moved to if an error occurs during processing. If this property is left blank, the files are deleted instead. This applies to reading the ${dts.rootFolder}/IN and ${dts.rootFolder}/SENT folders.
* dts.completedFolder - Folder where incoming control/data file pairs are moved to if an error occurs during processing. If this property is left blank, the files are deleted instead. This applies to reading the ${dts.rootFolder}/IN and ${dts.rootFolder}/SENT folders.

If relative file-paths are specified, they are resolved relative to the relevant sub-folder under dts.rootFolder. For example, to register a file for sending it needs to be stored in ${dts.rootFolder}/OUT but the file is first written into dts.temporaryFolder then moved to the final destination. In this case dts.temporaryFolder is resolved relative to ${dts.rootFolder}/IN.

**Distribution Envelope Configuration:**

* senderItkService - The ITK service added to outgoing distribution envelopes
* senderODSCode - The sender ODS code added to outgoing distribution envelopes
* auditODSCode - The audit ODS code added to outgoing distribution envelopes (if this property is not defined, senderODSCode is used).

**Control File Configuration:**

* dts.workflowId - Workflow ID added to outgoing control files (may be overridden for response messages where the original workflow id is echoed back)
* dts.senderMailbox - Send DTS mailbox (From\_DTS) added to outgoing control files
* dts.receiverWorkflowIds - Comma separated list of workflow IDs which will be accepted and handled on incoming messages

If incoming control files do not match a valid combination of dts.senderMailbox and dts.receiverWorkflowIds, the control file is not processed and is left in the directory. The DTS directory may be shared with other applications, and incoming messages may be intended for them.

**Queue Configuration:**

* itkDocumentSenderQueue - JMS queue for processing outgoing ITK documents
* distributionEnvelopeReceiverQueue - JMS queue for processing incoming ITK Distribution Envelopes
* itkMessageReceiverQueue - JMS queue for processing incoming ITK messages

The main input queue (itkDocumentSenderQueue) uses the JSON-encoded representation of ParsedDocument. The originalDocument property contains the document to send, while the properties section contains additional details such as the recipient ODS code.

**Address Resolution Configuration:**

* addressing.staticFiles - A comma-separated list of static files which provide static JSON-encoded DTSEndpointAddress values.

Configuration of the cache (e.g. time to live, cache size) is specified in the repositories\hazelcast.xml spring file.

**In-progress Folder:**

Details of the in-progress folder structure are available in the ciao-docs-finalizer state machine documentation.

ciao-docs-parser provides the InProgressFolderManagerRoute class to support storing control and event files in the in-progress directory.

* inProgressFolder - Defines the root folder that document upload process events are written to.

**Hazelcast Configuration:**

The following properties are applicable for repositoryConfig=hazelcast:

* hazelcast.group.name - Name of the hazelcast cluster group
* hazelcast.group.password - Password of the hazelcast cluster group
* hazelcast.network.port - The network port to use for the hazelcast server - if the port is already in use it will be incremented until a free port is found
* hazelcast.network.publicAddress – The (optional) public address of the hazelcast node - this can be used if nodes need to communicate over NAT.
* hazelcast.network.join.tcp\_ip - Comma separated list of static cluster members - if empty, multicast join should be enabled
* hazelcast.network.join.multicast.enabled - Boolean value specifying whether multicast join should be used to find cluster members - if false, static TCP-IP members should be specified
* hazelcast.network.join.multicast.group - Multicast address to use for finding cluster members
* hazelcast.network.join.multicast.port - Multicast port to use for finding cluster members

Example configuration file:

|  |
| --- |
| # Camel logging  camel.log.mdc=true  camel.log.trace=false  camel.log.debugStreams=false  # Select which processor config to use (via dynamic spring imports)  processorConfig=default  # Select which messaging config to use (via dynamic spring imports)  messagingConfig=activemq  #messagingConfig=activemq-embedded  # Select which idempotent repository config to use (via dynamic spring imports)  repositoryConfig=hazelcast  # repositoryConfig=memory  # Select which addressing config to use (via dynamic spring imports)  addressingConfig=static  # ActiveMQ settings (if messagingConfig=activemq)  activemq.brokerURL=tcp://localhost:61616  activemq.userName=smx  activemq.password=smx  # Common JMS/ActiveMQ settings (if messagingConfig=activemq or activemq-embedded)  jms.concurrentConsumers=20  # Hazelcast settings (if repositoryConfig=hazelcast)  hazelcast.group.name=ciao-transport-dts  hazelcast.group.password=ciao-transport-dts-pass  hazelcast.network.port=5701  hazelcast.network.publicAddress=  hazelcast.network.join.tcp\_ip.members=  hazelcast.network.join.multicast.enabled=true  hazelcast.network.join.multicast.group=224.2.2.3  hazelcast.network.join.multicast.port=54327  # Common addressing settings (if addressingConfig=static)  addressing.staticFiles=  senderODSCode=!REQUIRED!  # auditODSCode=optional - defaults to senderODSCode  senderItkService=urn:nhs-itk:services:201005:sendDistEnvelope  itkDocumentSenderQueue=cda-documents  distributionEnvelopeReceiverQueue=distribution-envelope-receiver  itkMessageReceiverQueue=itk-message-receiver  inProgressFolder=./in-progress  # DTS settings  dts.rootFolder=./dts  dts.temporaryFolder=../dts-temp  dts.filePrefix=!REQUIRED! ${siteid}${APP}  dts.workflowId=TOC\_DISCH\_DMS  dts.senderMailbox=!REQUIRED!  dts.receiverWorkflowIds=TOC\_DISCH\_DMS\_ACK  # DTS housekeeping options - if the property is empty the file is deleted  dts.errorFolder=./dts-error  dts.completedFolder= |

## 8.4. Log Events

TODO

## 8.5. Technical Narrative

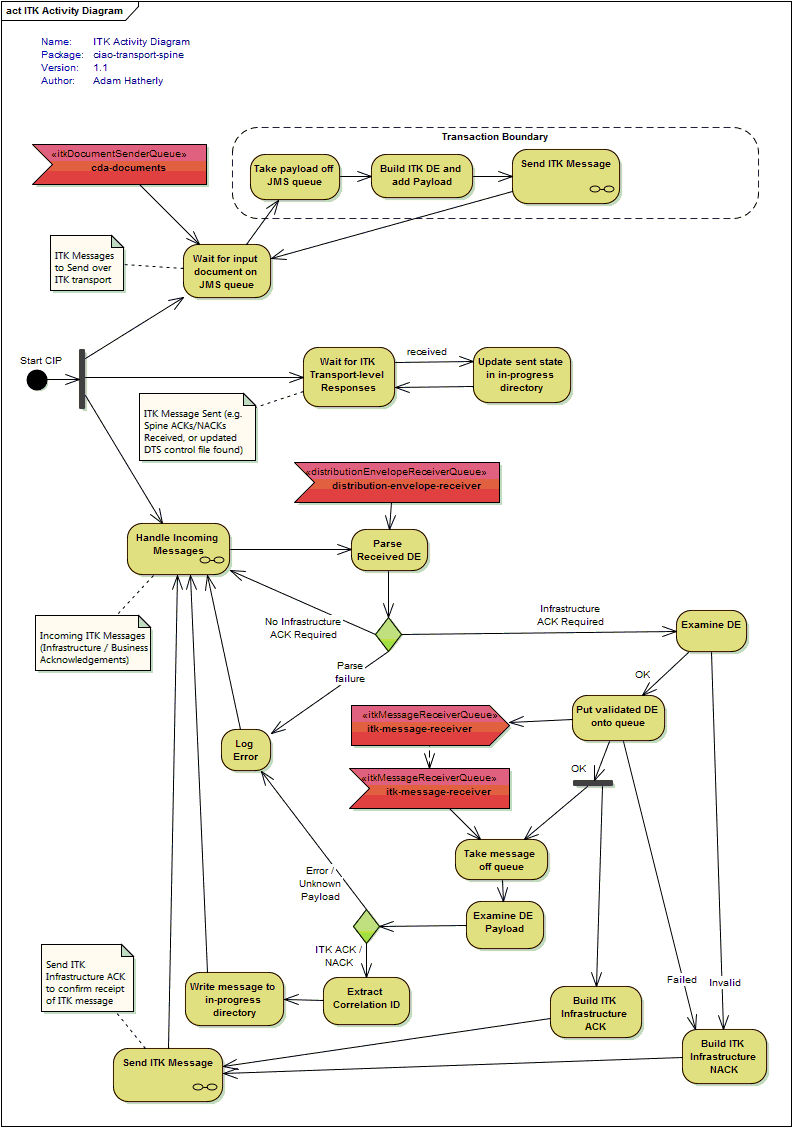
In order to understand how this CIP sends documents over DTS, we first need to understand the different protocols in use:

TODO: add diagram with DTS components in the bottom layer

If we were to consider the full set of activities to send a document over DTS using the below protocols, the process would appear quite complex. By splitting the activities into the two main layers (ITK and DTS), the processing required is much easier to understand.

The ITK layer includes the basic sending, coupled with a simple acknowledgement framework which allows the sender to request an Infrastructure Acknowledgement (to confirm the ITK message has reached its destination), and a Business Acknowledgement (to confirm "business receipt" - the exact meaning of which is specific to the type of document being sent).

The below activity diagram shows the high level activities involved in processing the overall message and ITK-layer interactions:



Some of the individual boxes on the ITK diagram above represent the lower level DTS interactions - these lower level protocol activities are shown in the below DTS activity diagram.

The DTS layer deals with sending these higher level ITK interactions over DTS/MESH. Messages are sent/received by writing/reading pairs of files on the filesystem. A DTS/MESH client process is responsible for handling the interactions with the central server.

* The control file (\*.ctl) specifies details such as sender address, receiver address, and correlation id.
* The corresponding data file (\*.dat) contains the payload message (in any format).

TODO: add DTS specific activity diagram

**Models**

The Control File message type is used when sending messages over DTS. The ciao-dts library provides a Java representation of this types, along with serialization to/from XML, and integration with Apache Camel's type conversion system.

**Control File**

Control files are used both when sending/receiving data messages over DTS/MESH, and when receiving reports. DTS control files are encoded using an XML format.

**Java Classes:**

* ControlFile - provides a bean-like representation of a DTS control file.
* ControlFileParser - parses an XML serialized control file to object form.
* ControlFileSerializer - serializes a control file object into XML.
* ControlFileTypeConverter - Integrates the control file parser and serializer with Camel.

# 9. ciao-docs-finalizer

## 9.1. Description

The purpose of this CIP is to monitor an in-progress folder which contains events associated with a document upload (typically originating from ciao-docs-parser). The CIP maintains an internal state machine for each document upload, and once completion is detected (success or otherwise) a configurable action is performed.

ciao-docs-finalizer is built on top of Apache Camel and Spring Framework, and can be run as a stand-alone Java application, or as a Docker container.

Each application hosts a Camel route, following the structure:



The details of the monitored folder and completion actions are specified at runtime through a combination of ciao-configuration properties and Spring XML files.

**Supported completion actions:**

* MoveToCompletedFolder - Moves the document's directory to the folder specified in the ./control/completed-folder file
* MoveToErrorFolder - Moves the document's directory to the folder specified in the ./control/error-folder file

For more advanced usages, a custom action can be integrated by implementing the TransitionListener Java interface and providing a suitable spring XML configuration on the classpath.

**In-progress Folder:**

The event files use a common naming format written by the CIPs involved in a document upload process. The contents and format of the event files is determined by the component writing the event and is not interpreted by this CIP.

A detailed description of the in-progress folder and associated state machine is available in the technical narrative.

## 9.2. Download

To pull down the code, run:

git clone https://github.com/nhs-ciao/ciao-docs-finalizer.git

You can then compile the module via:

mvn clean install -P bin-archive

This will compile a number of related modules - the main CIP module is ciao-docs-finalizer, and the full binary archive (with dependencies) can be found at target\ciao-docs-finalizer-{version}-bin.zip. To run the CIP, unpack this zip to a directory of your choosing and follow the instructions in the README.txt.

The CIP requires access to various file system directories and network ports (dependent on the selected configuration):

**etcd:**

* Connects to: localhost:2379

**ActiveMQ:**

* Connects to: localhost:61616

**Hazelcast:**

* Multicast discovery: 224.2.2.3:54327 (If enabled)
* Listens on: \*:5701 (If port is already taken, the port number is incremented until a free port is found)
* Connects to clustered nodes defined by the hazelcast.network.join.tcp\_ip.members property

**Filesystem:**

* If etcd is not available, CIAO properties will be loaded from: ~/.ciao/
* The CIP will monitor event files in the folder specified by the inProgressFolder property.

Docker download and run TODO.

## 9.3. Configuration

For further details of how ciao-configuration and Spring XML interact, please see ciao-core.

### 9.3.1. Spring XML

On application start-up, a series of Spring Framework XML files are used to construct the core application objects. The created objects include the main Camel context, input/output components, routes and any intermediate processors.

The configuration is split into multiple XML files, each covering a separate area of the application. These files are selectively included at runtime via CIAO properties, allowing alternative technologies and/or implementations to be chosen. Each imported XML file can support a different set of CIAO properties.

The Spring XML files are loaded from the classpath under the META-INF/spring package.

**Core:**

* beans.xml - The main configuration responsible for initialising properties, importing additional resources and starting Camel.

**Repositories:**

An `IdempotentRepository' is configured to enable multiple consumers access the same folder concurrently.

* repository/memory.xml - An in-memory implementation suitable for use when there is only a single consumer, or multiple-consumers are all contained within the same JVM instance.
* repository/hazelcast.xml - A grid-based implementation backed by Hazelcast. The component is hosted entirely within the JVM process and uses a combination of multicast and point-to-point networking to maintain a cross-server data grid.

**Processors:**

* processors/default.xml - Creates a file poller for the in-progress folder and an associated state-machine to handle detected event transitions.

**Messaging:**

* messaging/activemq.xml - Configures ActiveMQ as the JMS implementation for input/output queues.
* messaging/activemq-embedded.xml - Configures an internal embedded ActiveMQ as the JMS implementation for input/output queues. (For use during development/testing)

### 9.3.2. CIAO Properties

At runtime ciao-docs-finalizer uses the available CIAO properties to determine which Spring XML files to load, which Camel routes to create, and how individual routes and components should be wired.

**Camel Logging:**

* camel.log.mdc - Enables/disables Mapped Diagnostic Context in Camel. If enabled, additional Camel context properties will be made available to Log4J and Logstash.
* camel.log.trace - Enables/disables the Tracer interceptor for Camel routes.
* camel.log.debugStreams - Enables/disables debug logging of streaming messages in Camel.

**Spring Configuration:**

* repositoryConfig - Selects which repository configuration to load: repositories/${repositoryConfig}.xml
* processorConfig - Selects which processor configuration to load: processors/${processorConfig}.xml
* messagingConfig - Selects which messaging configuration to load: messaging/${messagingConfig}.xml

**Route Configuration:**

* inProgressFolderPollPeriod - Time in millis between polling attempts on the in-progress folder.
* inProgressFolder - The folder to monitor for document upload progress events.
* documentPreparationTimeout - Maximum time in millis to wait for a document to complete the preparation stage (prior to sending) before raising a timeout event.
* documentSendTimeout - Maximum time in millis to wait for confirmation that the document has been sent before raising a timeout event.
* infResponseTimeout - Maximum time in millis to wait for an infrastructure response before raising a timeout event.
* busResponseTimeout - Maximum time in millis to wait for a business response before raising a timeout event.
* idempotentActions - Boolean flag which selects if an idempotent checks should be performed before performing state transition actions. This is used to ensure that only one node in a cluster performs the action.
* actions - Specifies which action to perform when a document upload transitions to a particular state. The format is one mapping per line, where a mapping has the form: to={EVENT\_NAME} > {ACTION\_NAME}.

**Hazelcast Configuration:**

The following properties are applicable for repositoryConfig=hazelcast:

* hazelcast.group.name - Name of the hazelcast cluster group
* hazelcast.group.password - Password of the hazelcast cluster group
* hazelcast.network.port - The network port to use for the hazelcast server - if the port is already in use it will be incremented until a free port is found
* hazelcast.network.publicAddress – The (optional) public address of the hazelcast node – this can be used if nodes need to communicate over NAT.
* hazelcast.network.join.tcp\_ip - Comma separated list of static cluster members - if empty, multicast join should be enabled
* hazelcast.network.join.multicast.enabled - Boolean value specifying whether multicast join should be used to find cluster members - if false, static TCP-IP members should be specified
* hazelcast.network.join.multicast.group - Multicast address to use for finding cluster members
* hazelcast.network.join.multicast.port - Multicast port to use for finding cluster members

**Default Processor​:**

The default processor configuration does not currently support any additional properties.

Example configuration file:

|  |
| --- |
| # Camel logging  camel.log.mdc=true  camel.log.trace=false  camel.log.debugStreams=false  # Select which processor config to use (via dynamic spring imports)  processorConfig=default  # Select which idempotent repository config to use (via dynamic spring imports)  repositoryConfig=hazelcast  # repositoryConfig=memory  # Hazelcast settings (if repositoryConfig=hazelcast)  hazelcast.group.name=ciao-docs-finalizer  hazelcast.group.password=ciao-docs-finalizer-pass  hazelcast.network.port=5701  hazelcast.network.join.tcp\_ip.members=  hazelcast.network.join.multicast.enabled=true  hazelcast.network.join.multicast.group=224.2.2.3  hazelcast.network.join.multicast.port=54327  hazelcast.network.publicAddress=  inProgressFolderPollPeriod=5000  inProgressFolder=./in-progress  documentPreparationTimeout=60000  documentSendTimeout=180000  infResponseTimeout=300000  busResponseTimeout=17280000  idempotentActions=true  actions=\  to=SUCCEEDED > MoveToCompletedFolder \n\  to=FAILED > MoveToErrorFolder |

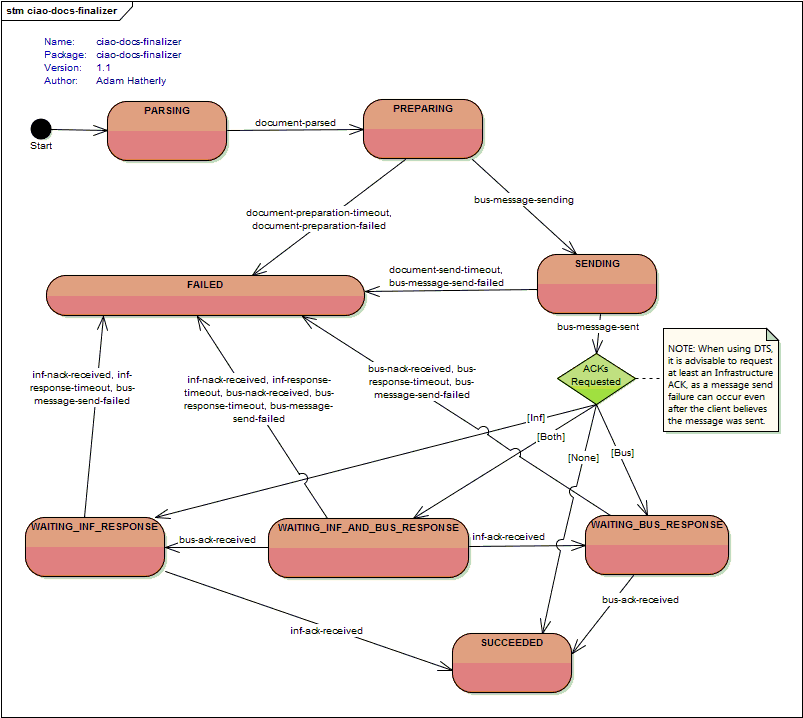
## 9.4. Log Events

TODO

## 9.5. Technical Narrative

**Document Upload State Machine**

ciao-docs-finalizer tracks the ongoing state of a document upload process via an internal state machine. The state machine is updated by monitoring an in-progress folder for event files which in turn trigger an event transition. This is represented in the below state transition diagram:



The event files are stored in the in-progress directory by other CIPs involved in the upload progress. The contents/format of the event files are determined by the CIP doing the writing, however the directory structure and naming of the event files is common across all CIPs.

**Structure of the In-Progress Folder**

The root folder contains multiple sub-folder, where each sub-folder corresponds to an on-going document upload process.

The name of the sub-folder matches the unique id of the document upload process. For example:

* ${ROOT\_FOLDER}/
  + 37eef483-f506-41e1-9058-cd4cd64a5280/
  + 69ade68e-28ea-496c-8324-8cc3f186e96d/
  + 760aadb6-7744-4fef-867e-6078a07b5cbd/
  + etc.

Each document upload process folder contains:

* input/
  + ${SOURCE\_FILE\_NAME}
* control/
  + completed-folder
  + error-folder
  + wants-inf-ack
  + wants-bus-ack
* events/
  + ${TIMESTAMP}-${EVENT\_TYPE}
  + ${TIMESTAMP}-${EVENT\_TYPE}
  + ${TIMESTAMP}-${EVENT\_TYPE}
  + etc.

Files are expected to be written by a single process and not subsequently updated.

ciao-docs-parser provides the InProgressFolderManagerRoute class to support storing control and event files in the in-progress directory.

**Input Folder**

The input folder is used to store a copy of the original input file prior to any transformations. The contents of this folder are not directly used or interpreted by ciao-docs-finalizer.

**Control Folder**

The control folder is used to store control/configuration details of the upload process.

completed-folder and error-folder are single-line text files defining where the in-progress folder should be moved to on completion and on error (subject to suitable completion actions being configured in ciao-docs-finalizer).

wants-inf-ack and wants-bus-ack are optional files defining whether an ITK infrastructure response and/or an ITK business response has been requested. The file contents are not read - the existence of the files is enough to determine if the acknowledgements have been requested. These files are associated with ITK transports and are added by the transport if applicable.

**Events Folder**

The events folder represents a log/journal of key events throughout the document upload process and is used to drive the ciao-docs-finalizer state machine.

Each event file is names using the pattern: ${TIMESTAMP}-${EVENT\_TYPE}.

The timestamp format is a modified version of ISO-8601 (suitable for file names):

* yyyyMMdd-HHmmssSSS - \*\*\* (in UTC timezone) \*\*\*

Event files are processed in order of increasing ${TIMESTAMP}. Only the filename is interpreted by ciao-docs-finalizer, the CIP responsible for storing the event determines the file contents.

The recognised event types are:

* document-parsed
* document-preparation-timeout
* document-preparation-failed
* bus-message-sending
* document-send-timeout
* bus-message-send-failed
* bus-message-sent
* inf-response-timeout
* inf-ack-received
* inf-nack-received
* bus-ack-received
* bus-nack-received
* bus-response-timeout

Additional details of the event types are available in the Event class.

**States**

The possible states of a document upload process are:

* PARSING
* PREPARING
* SENDING
* WAITING\_INF\_AND\_BUS\_RESPONSE
* WAITING\_INF\_RESPONSE
* WAITING\_BUS\_RESPONSE
* FAILED
* SUCCEEDED

Additional details of the states are available in the State class.

# 10. tkw-spine

## 10.1. Description

TODO

## 10.2. Download

TODO

## 10.3. Configuration

TODO

## 10.4. Log Events

TODO

## 10.5. Technical Narrative

TODO

# 11. Example Applications

## 11.1. Send Discharge Summary via Spine

This application takes a discharge summary produced by a hospital in Microsoft Word or PDF format, transforms it into a CDA document that conforms to the CDA schema X and sends it to the patient’s GP using the Spine TMS transport.

The logical CIP composition of this application is shown below:



*Figure 11.1-1 Logical Application*

Discharge summary sources place discharge summary documents into an input folder. The ciao-docs-parser CIP picks these documents up and parses their content into a JSON file which is placed on a queue. Documents that cannot be parsed correctly are put in an error folder. Need to explain in-progress and completed folders. The ciao-cda-builder CIP picks these JSON documents off the queue and builds them into a CDA document which is then placed on another queue. The ciao-transport-spine CIP picks the CDA documents from this queue and sends them to the NHS Spine using the TMS transport, requesting Spine to deliver them to a Primary Care System that can talk Spine TMS. Spine will route them through to the Primary Care System. Spine will send back to the ciao-transport-spine CIP various acknowledgements. The ciao-docs-finalizer CIP ensures the correct actions are taken when Spine doesn’t return expected acknowledgements. This needs explaining better.

# Part 3: Deploy and Run

# 12. Deploying the Cloud

A cloud can vary from a single machine to many. Talk about installing and configuring the full portfolio of infrastructure software e.g. OS, Docker, Kubernetes, system tools etc. Illustrate with an example deployment.

# 13. Operating the Cloud

Talk about system management of the cloud infrastructure. Illustrate with an example deployment.

# 14. Deploying CIPs

As CIPs are packaged as Docker images they can be easily deployed, configured and wired together on Docker enabled hosts. This can either be done manually or through configuration management tools. Previously when we talked about the pets versus cattle attitude to servers we placed the most likely uses of CIAO with the pet owners rather than the cattle ranchers, so you might think that doing deployment manually is the easiest and quickest option as you don’t have to spend any time learning how to use a configuration management tool. However as CIAO is architected around microservices, even with just a couple of hosts there are a lot of moving parts to deploy and wire-up to create a working integration. You might start by doing this by hand so you can try things out, but once you have a production ready deployment you will need to document in detail what you have done, so the next time you or someone else needs to do it (and there always will be a next time) they can reproduce the deployment successfully. Instead of writing your detailed deployment steps down in a document that will be misread and mistyped, a configuration management tool allows you to record your instructions in some form of machine readable script that it will then run to accurately carry out the deployment. We would therefore strongly recommend that you use a configuration management tool even for the simplest of CIAO deployments.

So which configuration management tool should you use? There are many open source tools such as Chef, Puppet and Salt. Within the container community there are also many evolving tools for deployment and composition, for example Google Kubernetes and Docker Compose. Our preference is to use Ansible. This is an established open source configuration management tool that is simple to use and scales down to the typical CIAO pet centred deployment.

Ansible allows you to deploy stuff to hosts, orchestrate that deployment across multiple hosts so things happen in a particular order, configure the stuff once deployed and even provision hosts from cloud providers.

Ansible uses a DSL (Domain Specific Language) that allows you to write your deployment and configuration instructions in files called *playbooks*. The DSL is expressed in YAML, which is an easy to read JSON like notation. A playbook essentially contains a list of plays. Each play has a list of hosts to configure and the list of tasks to be run on these hosts. The Ansible software runs on a single Linux host, reading your playbook, for each play it connects via SSH to each host in the associated host list simultaneously, it then runs each task in the task list in a defined order on all the hosts in parallel. To run a task Ansible writes some python script which it then pushes to a host over SSH to execute. One of the nice things about Ansible is this push model, unlike other configuration management tools you don’t need to install any special agent software on remote hosts.

The rest of our discussion on deploying CIPs will be based around Ansible. If you want to use another configuration tool then you will need to translate what follows into your tool’s own DSL. Note that the Ansible DSL uses commands (called methods) that are specific to a Linux distro. So for example you need to use the APT method when installing packages on Ubuntu but the YUM method in CentOS.

To find out more about Ansible browse the Ansible documentation at:

[*https://docs.ansible.com/ansible/index.html*](https://docs.ansible.com/ansible/index.html)

Or read the excellent book *Ansible: Up and Running* by Lorin Hochstein.

To illustrate the deployment of CIPs using Ansible we will use an example CIAO cloud consisting of four hosts:



All the hosts are running Ubuntu 14.04.3 LTS. The host chico will be used as the management host to run Ansible itself and all the event and log monitoring software. The three other hosts; harpo, groucho and zeppo, will be used to run CIPs and the supporting CIAO base services.

## 14.1 Install Ansible

You will need to install Ansible on a Linux host that will act as the configuration management controller. This host can also be one of the hosts on which you install CIPs, but we would recommend you install it on a separate dedicated control (management) machine (host).

Installation instructions for different flavours of Linux can be found at:

[*http://docs.ansible.com/ansible/intro\_installation.html*](http://docs.ansible.com/ansible/intro_installation.html)

We will install Ansible on the chico host. Login to the host with a user account that has sudo privileges. Then install Ansible using the apt-get utility:

$ sudo apt-get install software-properties-common

$ sudo apt-add-repository ppa:ansible/ansible

$ sudo apt-get update

$ sudo apt-get install ansible

## 14.2 Playbooks Setup

Create a directory structure to store your playbooks and associated files:

$ mkdir ciao-playbooks

$ cd ciao-playbooks

$ mkdir files

$ mkdir templates

You need to create an Ansible inventory file that lists the hosts that you are going to deploy to. In our example we are going to deploy to four hosts, so we create an inventory file called hosts in the ciao-playbooks directory which contains:

|  |
| --- |
| chico ansible\_ssh\_host=10.210.162.21  harpo ansible\_ssh\_host=10.210.162.22  groucho ansible\_ssh\_host=10.210.162.28  zeppo ansible\_ssh\_host=10.210.162.29 |

Each line in the file represents a single host. The first value is a name for the host that you use to refer to it in your playbook. The ansible\_ssh\_host value gives either the IP address or domain name of the host. There are additional key value pairs you can add to a host line including:

ansible\_ssh\_port – The ssh port number, if not 22.

ansible\_ssh\_user – The default ssh user name to use.

ansible\_ssh\_pass - The ssh password to use (this is insecure).

ansible\_ssh\_private\_key\_file - Private key file used by ssh.

Each of the hosts needs a SSH server running and a user account with sudo privileges that you can use. Normally you would setup the same user account across each host as this makes life simpler. However you can use a different user account for each host in which case you will need to add the relevant additional ssh key value pairs to each host line in the inventory file.

With the same user account across each host you can define this in an Ansible configuration file named ansible.cfg created in the playbooks directory:

|  |
| --- |
| [defaults]  inventory = hosts  remote\_port = 22  remote\_user = ciaoadmin  ask\_pass = True  nocows = 1  ask\_sudo\_pass= True |

This defines the inventory file of hosts to use, the ssh port to use for all hosts and the ssh user name to use for all hosts. For simplicity this example doesn’t use a ssh private key but will ask for a ssh password. Enabling nocows turns off the use of the cowsay program (which generates ASCII pictures of a cow with a message) to format Ansible logging (the author’s preference). Enabling ask\_sudo\_pass will ask for a sudo password.

We can now test that Ansible is working and that both the inventory and configuration files are correct by asking Ansible to ping all the hosts in the inventory.

$ ansible all –m ping

SSH password:

SUDO password[defaults to SSH password]:

chico | success >> {

"changed": false,

"ping": "pong"

}

harpo | success >> {

"changed": false,

"ping": "pong"

}

groucho | success >> {

"changed": false,

"ping": "pong"

}

zeppo | success >> {

"changed": false,

"ping": "pong"

}

$

Ansible first prompts for the SSH password, it then does an Ansible ping of each host and reports that a pong has been received. As this is not changing any state on a host, changed is reported as false.

If you get error messages then run the Ansible command again but add the –vvvv connection debugging verbose mode flag at the end to see more detailed error information.

## 14.3 CIAO Base Services

Irrespective of which CIPs you plan to deploy there is a set of standard CIAO base services that need to be deployed first. These include:

* A python package that is a workaround for Ansible issues with installing Docker images
* Logspout to route log data from Docker containers to Logstash
* Logstash to collect log data
* Elastic to store the log data
* Kibana to provide a web console to view the log data
* Nagios to provide a web console to view events
* Etcd-browser to provide a web console to view or change configuration values
* A etcd cluster to support configuration management
* A ZooKeeper cluster to support the clustering of ActiveMQ
* A ActiveMQ cluster for messaging

For our example CIAO cloud we are going to first deploy the Ansible python fix to all the hosts. Then we are going to deploy ELK and Nagios to the management host.



Within the inventory file we can define groups of hosts. Each group is given a meaningful group name and the hosts within the group are listed. Any host can belong to as many groups as you like. Below is our updated hosts file with groups added to match our deployment plan.

|  |
| --- |
| chico ansible\_ssh\_host=10.210.162.21  harpo ansible\_ssh\_host=10.210.162.22  groucho ansible\_ssh\_host=10.210.162.28  zeppo ansible\_ssh\_host=10.210.162.29  [elk]  chico  [nagios]  chico  [etcdbrowser]  chico  [logspout]  harpo  groucho  zeppo  [etcd]  chico  harpo  groucho  zeppo  [zookeeper]  harpo  groucho  zeppo  [activemq]  harpo  groucho  zeppo |

We next need to define the Ansible playbook to deploy our CIAO base services. This is called ciao-base.yml. You can download it with all its associated files and templates by:

$ git clone http://github.com/makelly/ciao-playbooks.git

Need to update url once repository moves.

To run the playbook:

$ ansible-playbook ciao-base.yml

Note Ansible issue in pull Docker images from v1 repositories.

## 14.4. Application to Send Discharge Summary via Spine

We are going to deploy this application on our example CIAO cloud in the following composition:



We are going to have a ciao-docs-parser, ciao-docs-enricher, ciao-cda-builder and ciao-docs-finalizer running on three CIP hosts. There will be shared folders on the management host chico that these hosts can access. There will be a single instance of ciao-transport-spine also running on one of the management host [Need to work out how this can be scaled out] . To simulate a Spine we run a single instance of tkw-spine on the management host.

To setup the shared folders we are going to use sshfs. To do this you first need to install sshfs on each host:

$ sudo apt-get install sshfs

You must then uncomment the user\_allow\_other line in /etc/fuse.conf file. On chico, which will host the shared folders, create the shared directories:

$ sudo mkdir –m 777 /opt/ciao-share

$ sudo mkdir –m 777 /opt/ciao-share/input

$ sudo mkdir –m 777 /opt/ciao-share/error

$ sudo mkdir –m 777 /opt/ciao-share/in-progress

$ sudo mkdir –m 777 /opt/ciao-share/completed

Then on each of the other hosts that will be running the CIPs, create a local directory where the shared folders will be mounted and then mount them using sshfs:

$ sudo mkdir /opt/ciao

$ sudo sshfs <username>@<chico address>:/opt/ciao-share /opt/ciao \

–o allow\_other

If you need to un-mount the share on a host then:

$ fusermount –u /opt/ciao

|  |  |
| --- | --- |
| **Port number** | **Used by** |
|  |  |
| 514 | ELK Logstash syslog |
| 2181 | ZooKeeper client |
| 2379 | etcd client |
| 2380 | etcd peer to peer |
| 2888 | ZooKeeper peer to peer |
| 3888 | ZooKeeper peer to peer |
| 4001 | etcd client |
| 4002 | Tkw-spine |
| 5701 | Hazelcast ciao-docs-parser |
| 5702 | Hazelcast ciao-docs-finalizer |
| 7999 | Etcd browser |
| 8000 | Logspout |
| 8080 | Kibana web console |
| 8081 | Nagios web console |
|  |  |
| 8161 | ActiveMQ web console |
| 9200 | ELK Elasticsearch |
| 61616 | ActiveMQ client TCP transport |
| 61619 | ActiveMQ peer to peer |
|  |  |



# 15. Operating CIPs

Talk about system management of CIPs. Illustrate with an example cloud deployment and exemplar CIP.

# Part 4: Building a new CIP

# 16. Design

TO DO

## 16.1. Principles

Talk about the key design principles that should be used when designing a new CIP and how to apply them at the code level, also any patterns or conventions. Also talk about initially assessing if a candidate CIP needs disaggregating into separate independent CIPs. Illustrate with our exemplar CIP (:-) suggest we use this as a single example that is carried through the rest of the text).

## 16.2. Methods and Tools

Talk about useful design methods and tools to use.

## 16.3. Outputs

Talk about the design outputs – illustrate with exemplar CIP.

# 17. Environments and Tools

Talk about setting up environments based on our technology stack (this will be covered in Part 5, so maybe pointers to there).

Talk about development and testing tools.

Talk about source control and also container control (Docker Hub)

# 18. Testing

Talk about unit testing etc. Illustrate with exemplar CIP.

# 19. Development

Talk about the actual coding. Illustrate with exemplar CIP.

CIAO Cheat Sheet

# ciao-docs-parser

ciao-docs-parser creates routes using the following structure:

Input Folder -> DocumentParser (java class) -> Output Folder

TikaDocumentParser is currently the only implementation of DocumentParser. It converts a binary input stream into an XHTML DOM, then calls a PropertiesExtractor to find properties in the DOM.

New ‘parsers’ are likely to be an instance of TikaDocumentParser backed by either RegexPropertiesExtractor or SplitterPropertiesExtractor.

Examples of using both are in the ciao-docs-parser-kings factory classes.

## Configuring a new parser route

* Add a new route name to the documentParserRoutes CIAO property
  + documentParserRoutes=example1,example2
* Configure the input folder, parser ID and output queue using the documentParserRoutes.{routeName} prefix
  + documentParserRoutes.example2.processorId=example2Parser

## Adding a new type of parser

The actual parser is likely to be an instance of TikaDocumentParser backed by a configured PropertiesExtractor.

* Create a new Java class to act as a factory: e.g. ExamplePropertyExtractorFactory
* Add a new static method which returns a PropertiesExtractor <Document>: e.g. createWordEDNPropertiesExtractor
* Select the PropertiesExtractor implementation – SplitterPropertiesExtractor is the most flexible and dynamic
* Fill the method in with the required mappings – these are likely to be static strings like XPath selectors, property names, regex patterns etc – although it is also possible to create new custom classes
* Add or update a spring XML config file to call the factory class and associate the parser with a registry ID (see the META-INF/spring/processor examples in ciao-docs-parser)
* Make sure the new factory class and spring config is on the classpath
* Configure CIAO properties to start the route (see previous)

## Creating mappings

While creating the mappings, it is useful to create a dummy PropertiesExtractor class which takes the intermediate Tika XHTML DOM and writes it out to the console/filesystem. This can then be used to find the patterns / sections in the document and any associated regex or XPath expressions. Creating a small test-rig which takes some input, runs the extractor, and prints output is useful while creating the extractor rules: compile, run, check output, repeat…

# ciao-docs-transformer

ciao-docs-transformer creates routes using the following structure:

Input Queue -> DocumentTransformer (java class) -> Output Folder

DocumentTransformer chooses whether transformations are done in-place and keeps track of which properties have been transformed. The transformations are performed by a delegate PropertiesTranformation object.

Sseveral types of transformation are available but the standard starting point is to use PropertiesTransformer which in turn provides a DSL for the rest of the transformations.

Examples of using them are in the ciao-docs-transformer factory classes.

## Configuring a new transformer route

* Add a new route name to the documentTransformerRoutes CIAO property
  + documentTransformerRoutes=example1,example2
* Configure the input queue, transformer ID and output queue using the documentTransformerRoutes.{routeName} prefix
  + documentTransformerRoutes.example2.transformerId=example2Transformer

## Adding a new type of transformer

* Create a new Java class to act as a factory: e.g. ExamplePropertyTransformationFactory
* Add a new static method which returns a PropertiesTranformation: e.g. createExamplePropertiesTransformation
* Select the PropertiesTranformation implementation – PropertyTransfomer is the likely starting point and provides a DSL for building other transformations
* Fill the method in with the required mappings – these are likely to be static strings like selectors, property names, regex patterns, date formats etc – although it is also possible to create new custom classes
* Add or update a spring XML config file to call the factory class and associate the transformer with a registry ID (see the META-INF/spring/processor examples in ciao-docs-transformer)
* Make sure the new factory class and spring config is on the classpath
* Configure CIAO properties to start the route (see previous)

# 20. Contribution

Talk about contributing both the source code and the container as open source – the how. Talk about documentation. Talk about the CIAO Hub – need something that people can go to discover what CIPs are out there, and somewhere developers can signpost their new CIP (this could be just a section of nhs developer network or whatever).

# 21. Assurance

Talk about when a CIP needs some sort of assurance (or accreditation), who does this, what it is, where this gets recorded.