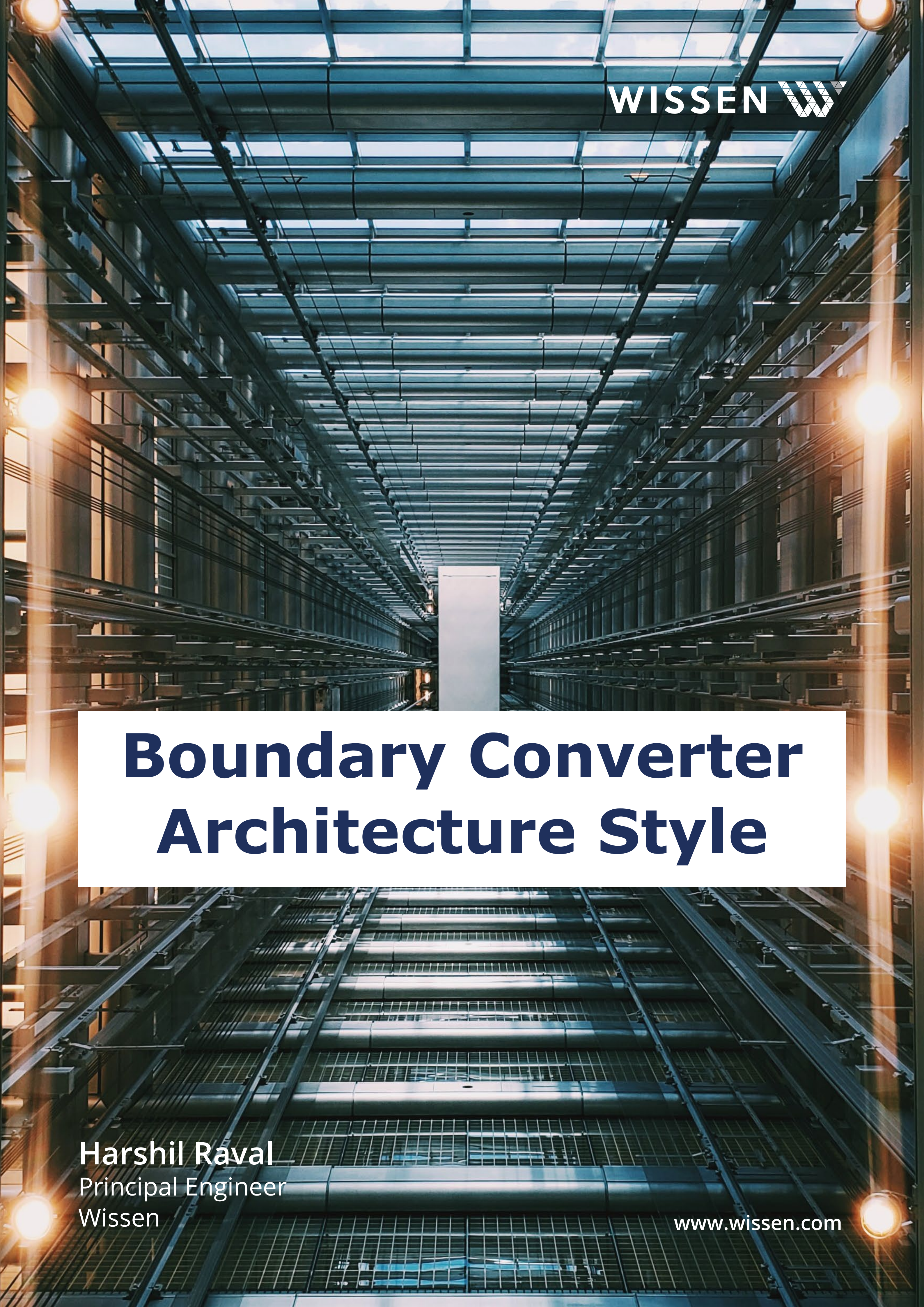


WISSEN

Boundary Converter Architecture Style

Harshil Raval
Principal Engineer
Wissen

www.wissen.com

Table of Contents

Overview

In Distributed applications, where there are multiple services, storage systems, batch processors and messaging components – which interact with each other to provide business value. For a global firm, it is usual to deal with location specific / operation specific / regulatory specific / usage specific units and formats on a daily basis. This requirement leads to multiple formats and units being processed and stored across components. This need – to deal with multiple formats and units – leads to complex issues. This paper discusses such potential issues and provides a generic solution in the form of an architecture style.

Issues

- Maintaining multiple formats / units across components and storage systems.
- Every Component must check every request for format / unit consistency along with other necessary validations.
- While debugging, it would not be clear which format / unit is being used unless the developer has end to end knowledge of the system or that format / unit information is part of the context.
- Each request has to bear the overhead of keeping extra info related to format and unit.
- The whole system is at very high risk of wrong data processing.

Examples

Here are some of the examples of formats and units which could lead to above issues in distributed applications :

- Multiple Date Formats and Time-zones
- Currencies
- Units of measurement
- Domain specific formats / units.

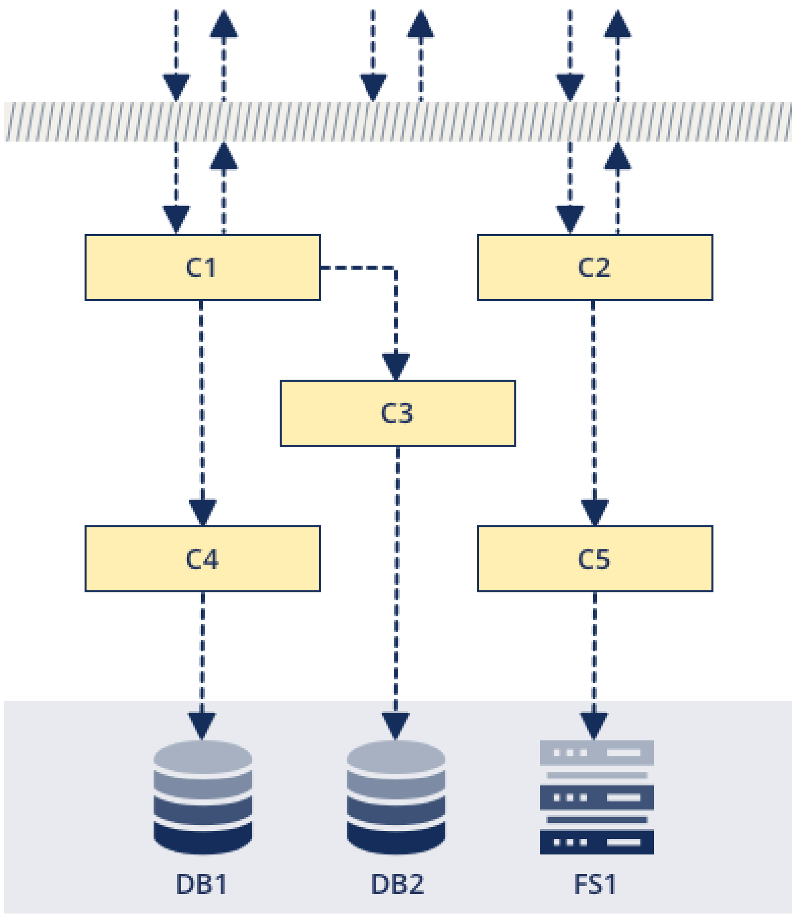
Here we have taken an example of Multiple Date Formats and Time-zones to describe in detail – how it creates issues in distributed application, what are the pain points and how to address these issues with Boundary Converter Architecture.

Problem

What are the issues of having multiple date formats in the system?

Imagine a distributed system where there are multiple storage systems including multiple dbs and file systems and there are multiple components which interact with each other

to provide some functionality. Some of the components also work as standalone.



df1 : dd/mm/yyyy
df1 : mm/dd/yyyy
df1 : dd/mm/yy
df1 : yyyy/mm/dd

TZ1 : IST
TZ2 : GMT
TZ1 :EST

now, suppose c1 uses date format df1,
c2 -> df2,
c3 -> df3,
c4 -> df4,
c5 -> df2 and df3,
db1 -> df1 and df2,
db2 -> df2,
fs1 -> df1, df2, df3, df4

If the system is designed to handle dates in this manner then there are following issues which can occur:

(i-1): Maintaining multiple formats across components and storage systems
(i-2): Every time a request comes to a component, it needs to check whether the format is consistent locally. If not, the date has to be converted to local format.
(i-3): While debugging it would not be clear which format needs to be used or being used unless the developer has end to end knowledge of the system.
(i-4): If the system uses multiple time-zones in the date then each request needs to keep track of respective time-zones.
(i-5): the system is at very high risk of wrong date processing.

Why does this issue need attention?

A system with Multiple dates and time-zones (MDaTZ) is prone to inconsistencies and wrong processing.

Use Case 1:

There is an order placing system which is a standalone service and takes following inputs to create order:

```
Order_name, //name of the order
Order_items, //order details
Order_customer, //customer details
Order_date, //order placed date (mm/dd/yy)
```

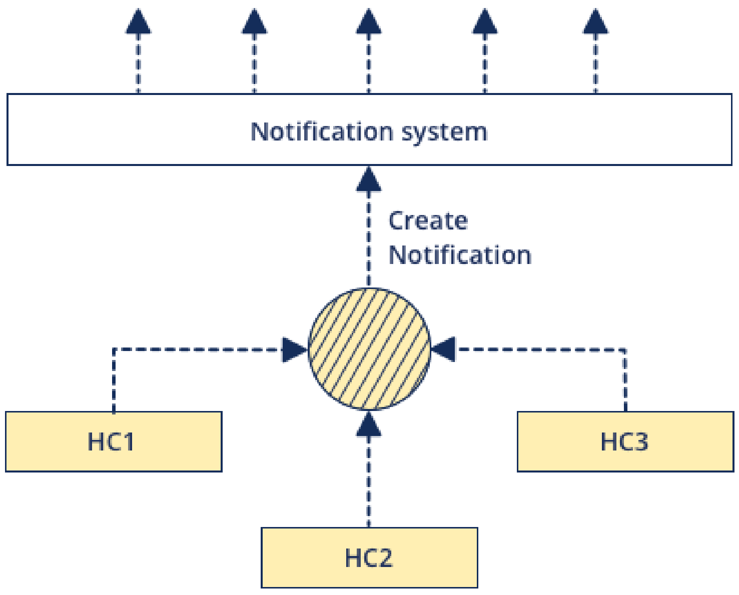
Some other component – X calls place_order() service. Now, this external component – X uses dd/mm/yy as their date format. If while calling place_order(), X passes the order_date as:01/02/03 which means for X the date is 1 st Feb 2003 but for place_order() service the date is 2 nd Jan 2003. In this case even if there are validations to check the date, the date will pass them as both the dates: 1 st Feb 2003 and 2 nd Jan 2003 are valid. But, the system is in an inconsistent state.

The obvious solution to this, is to know what you are calling. Meaning X must know that place_order() is expecting a date in mm/dd/yy format. So before calling place_order() convert the date from dd/mm/yy to mm/dd/yy. Since we write a lot of components and they need to interact with each other to provide different functionalities, does it make sense to write multiple date and timezone converters in each component to make sure it is consistent with the date format of the requesting component? That will lead to code repetition, which comes with additional cost of developer's time.

When will having MDaTZ would create a major problem?

Use Case 2:

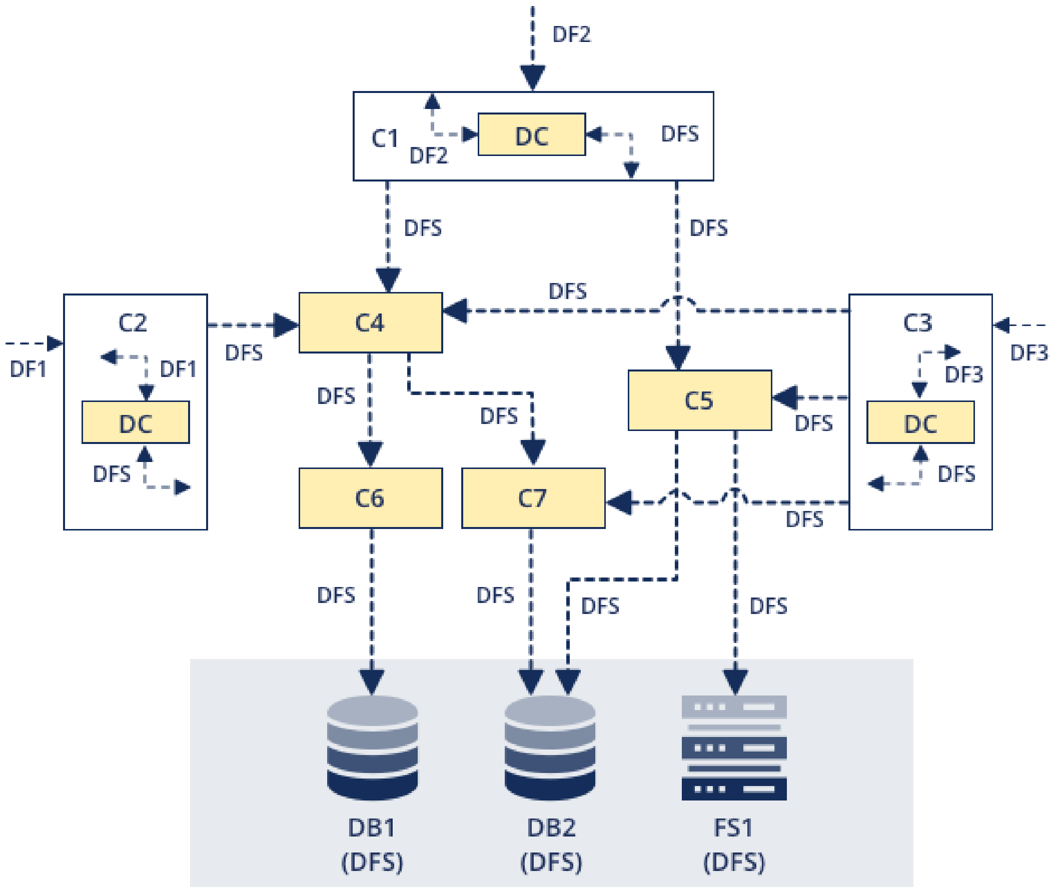
Suppose you have created a notification system which sends alerts and notifications to users. This is a high availability system so multiple healthcare firms prefer to use this to send their patients notification and alerts globally.



Notification System (NS) uses date format dd/mm/yyyy with GMT as their timezone. So every caller HC1, HC2, HC3,....HCN should first convert the notification date to dd/mm/yyyy format and GMT time zone for the notification to be sent at an accurate time. Suppose a caller HCX did not convert the notification time to GMT zone. This could result in early or delay notification.Suppose a caller HCY did not convert the format from mm/dd/yyyy and passed 01/02/2019. In this case, the system will send the notification on 1 st Feb 2019 instead of 2 nd Jan 2019.This delayed notification/alert could be fatal as the receiver could be a patient fighting against a major illness.

Solution

How the Boundary Converter Architecture style will resolve these issues?



Where components [C1], [C2] and [C3] are boundary components and components [C4], [C5], [C6], [C7] are internal components. Also, storage [db1], [db2] and [fs1] are considered as internal components.In this system whenever a request comes to either of [C1], [C2] or [C3] the date part will first be converted to (dfs) and then the request will be sent for further processing. This wayall the internal components will have to deal with only one date format and timezone ==>(dfs)

In the response, before sending it to the external system, the boundary component will convert the date part of response to local format of the external component.

i.e.,

in case of [C1]'s response, (dfs) will be converted to (df2),

in [C2]'s response, (dfs) will be converted to (df1), and

in [C3]'s response, (dfs) will be converted to (df3)

Also, in this example we are considering storage as an internal component. So, [db1], [db2] and [fs1] will use (dfs) for all date fields. This will liberate system designers and developers from maintaining MDaTZ in the system. A lot of calls to convert date format and time-zones will be eliminated. While debugging, a person will always be sure which format is being used for the date.

While adding a new component, the architect will only have to decide whether the component is Boundary Component [BC] or Internal Component [IC]. If it is a Boundary Component [BC] she/he can add Date Converter {DC} in the component to convert incoming date to (dfs). If it is an Internal Component [IC] then code can be written to process only (dfs).

The component can be designed to be [IC] and later if the system needs the [IC] can be converted to [BC] by just adding {DC} to it. This solution will minimize the risk of processing the wrong date and wrong time zone mentioned in Use Case 1 and Use Case 2.

Where do we need to make changes to adopt Boundary Converter Architecture style?

Use Case 3: Existing System:

If the system is already established and if architect decides to adopt Date-Boundary architecture style then following changes are necessary:

1> Identify a date format and timezone with which internal components will deal.

2> Identify [BC] and [IC] in the system

3> Write {DC} at each boundary component.

4> Remove all {DC} from internal components.

5> If the storage has any data then change the date format of all the stored dates.This is a one time effort which requires the system to be down and needs to carry out with utmost accuracy.

Use Case 4: New System:

Adopting Boundary Converter Architecture in a new system is relatively easy as the tricky task of changing date format and timezone in data storage is not involved. Here is the list of tasks which needs to be carried out:

1> Identify a date format and timezone with which [IC]s will deal.

2> Identify [BC] and [IC] in the system.

3> Write {DC} at each [BC].

Conclusion

For any distributed system, consistency is key to provide business value. This paper discusses potential issues with multiple formats and units in a system and provides a generic solution in terms of Boundary Converter Architecture Style.

About Wissen

Established in the year 2000 in the US, Wissen is an Information Technology company headquartered in Bangalore, India. With global offices in US, India, UK, Australia, Mexico and Canada, Wissen is an end-to-end solution provider for companies in sectors such as Banking and Financial Services, Telecom, Healthcare, Manufacturing and Energy verticals.

With best in class infrastructure and development facilities spread across the globe, the company has successfully delivered $650 million worth of projects for more than 20 of the Fortune 500 companies. Wissen's 1400+ highly skilled professionals, a strong leadership team, and technology expertise help clients build enterprise systems, implement a modern digital strategy, and gain a competitive advantage with business transformation.

Our service offerings

- Application Development
- Artificial Intelligence and Machine Learning
- Big Data and Analytics
- Visualization and Business Intelligence
- Robotic Process Automation
- Cloud and Mobility
- Agile and DevOps
- Quality Assurance and Test Automation
- Infrastructure Management

hello@wissen.com

+91 982 042 6895 | US +1 847 868 9595

WISSEN

www.wissen.com

USA | CANADA | UK | INDIA | AUSTRALIA

Great Place To Work® Certified SEP 2020–AUG 2021 INDIA