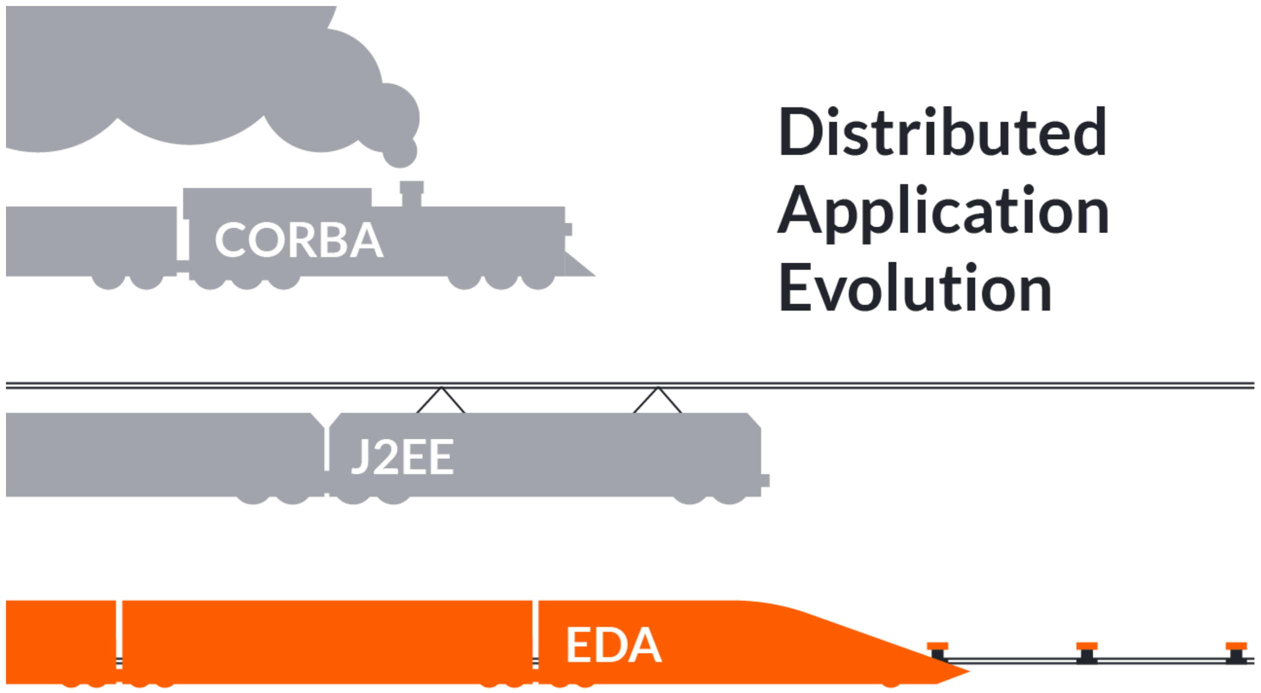
Next Generation Development
For Distributed Applications



If you've been around long enough, you will notice cyclical patterns in software development. Every so often an urgency for the same problems re-appear with new approaches, processes and, eventually, tools to assist with a solution.

In the late 90s, Sun Microsystems published a massive specification titled "Java™ 2 Platform Enterprise Edition Specification." It described a Java architecture to enable important quality attributes of an enterprise information system, namely security, high availability, reliability, scalability and so forth. According to the spec:

J2EE achieves these benefits by defining a standard architecture that is delivered as the following elements:

- J2EE BluePrints - A standard application model for developing multi-tier, thin-client services.
- J2EE Platform - A standard platform for hosting J2EE applications.
- J2EE Compatibility Test Suite - A suite of compatibility tests for verifying

that a J2EE platform product complies with the J2EE platform standard.
- J2EE Reference Implementation - A reference implementation for demonstrating the capabilities of J2EE and for providing an operational definition of the J2EE platform.

Having the J2EE BluePrints and Reference Implementation both served as means to create a well structured J2EE application. Eventually products like Weblogic, Websphere, and JBoss (known as application servers) implemented J2EE component containers, J2EE platform APIs, and other features defined in the specification. A J2EE application server became wildly popular because no enterprise would dare to implement the entire J2EE specification. The convenience of the application server proved the key to the growth of Java in the enterprise. So much so that today, the majority of 20+ year old Fortune 500 companies still have a substantial J2EE footprint in their application portfolio.

During this same time period, UML (Unified Modeling Language) became the standard for capturing the entities, data, relations, events, components, and sequences involved in a system. It was (is) a great way of documenting artifacts that both validate the intent of a business specification and help guide software development.

Where UML ends and J2EE begins, there is a massive void. J2EE is not model driven and therefore the J2EE specification fell short in offering guidance on how to translate requirement artifacts into code artifacts. Therefore, there was a mountain of heavy lifting to build the systems that power so many large enterprises today. The design of many of these systems were guided by the J2EE BluePrints. Even though the intent of the blue prints was admirable, the result of small and large systems was a monolithic architecture. Since these applications were “example-driven” and not “model-driven,” they often resulted in big balls of mud.

New Old Concepts

Event modeling, domain models, and event-driven architectures are the concepts that drive many of today’s newest software system designs. None of these concepts are new, especially events. Today the sheer number of events, and the decisions driven by those events, have brought these concepts to the forefront of system architecture and design.

Unlike J2EE, there is no standard specification for event driven systems. CQRS defines a pattern that lends itself nicely to event driven systems. Since it is a pattern, it does not offer a standard implementation. For better or worse, the predictability of J2EE guaranteed that Java code written according to the specification could run on a variety of vendor J2EE implementations (an application server).

If we consider these concepts a new means of development for distributed applications, we can define a new high level Event Driven Application Specification (EDAS) with the following elements:

- Model Driven - define the interesting things in our system.
- Event Driven - define the interesting occurrences in our system.
- CQRS Enabled - define a means of creating and handling commands and queries.
- Event Sourcing - define a storage method in which the state of a “business entity” or aggregate is stored as a series of changes (event), instead of storing the current state alone. The current state is instead rebuilt from these events when needed..
- Messages - the packaging and mechanism by which things communicate.
- Test Site - standard means of testing CRUD events.

With this specification, we will overcome the shortcomings of the J2EE Blueprints and define a way of turning models into functional aspects of our event driven system. These models already contain incredible detail and clues of the static and dynamic state of our event driven system.

Did you notice there is no mention of microservices? Although its qualities fit nicely into such a specification, we will resist the urge to make it mandatory. We will, however, reference them.
If your enterprise has moved from project to product based thinking, you are likely considering a vast # of microservices. 100s to maybe 1000s of them.

Implement Our Event Driven Application Specification

Suppose we have a well defined project with nicely written specifications. Let’s assume the event modeling is complete. All the events, commands, queries, interactions and so forth are clearly identified. We put the finishing touches on the domain model. Remember that unlike the J2EE spec, the EDAS specification has the ability to turn a domain model into event-driven code. More on this later.

Part of your non-functional requirements include the usual suspects:

- Backup/Failover
- Location Transparency
- High Availability
- Low Latency
- Messaging Infrastructure
- Observability
- Security

It is natural to first reach for things off the shelf. We quickly discover stitching so many technologies together for an enterprise level deployment is daunting.

We decide to look for something akin to a J2EE application server, but in the event-driven application space. We discover a framework and server called Axon, purpose built for event driven architectures. It provides an easy to use Java API for defining commands, queries, and events, each of which has its own complete bus topology. Out of the box Axon provides an event store for event sourcing. After a couple training videos, a read of a few technical blogs, and review of the docs, we are ready to go!

Well, not yet.

Enterprise Microservices Is Like A Railway

Time for an analogy.

Think of an enterprise grade microservices operational environment as a railway. A railway has two major components:

- The infrastructure (the permanent way, tracks, stations, freight facilities, viaducts, tunnels, etc.)
- The rolling stock (the locomotives, passenger coaches, freight cars, etc.)

Let's correlate this to our problem space.

- Axon is the railway infrastructure, complete with the facilities to handle movement throughout the railway.
- The domain model describes the "things" and the relationship of the "things" that are transported on the railway.
- The event model defines all the possible execution conditions (success and exception) of the railway.

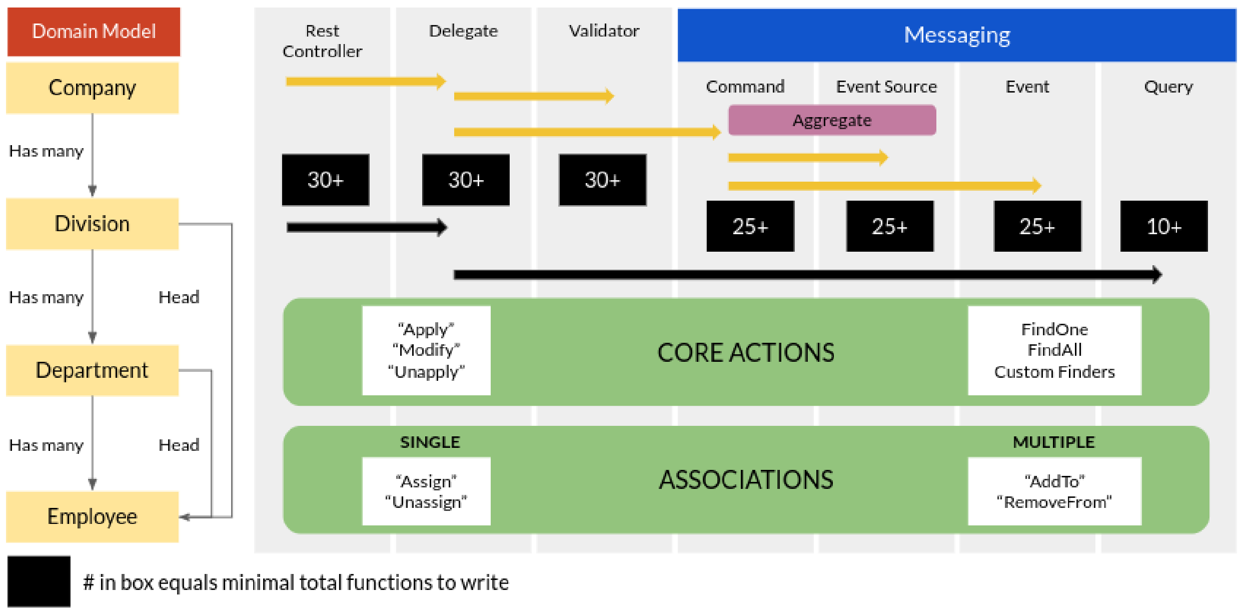
At this point we have a great amount at our disposal. Unfortunately, we do not yet have a functioning railway.

Through simple configuration, Axon instantiates the railway infrastructure. This means we are on the hook for creating (developing) the rolling stock. Even for a modest sized domain model, this will equate to 10s to 100s of 1000s of lines of code.

Recall that our EDAS guarantees more than "hello world," giving us a realization of our domain model within the context of an Axon led technology stack.

Jumpstart a Project Using The Domain Model

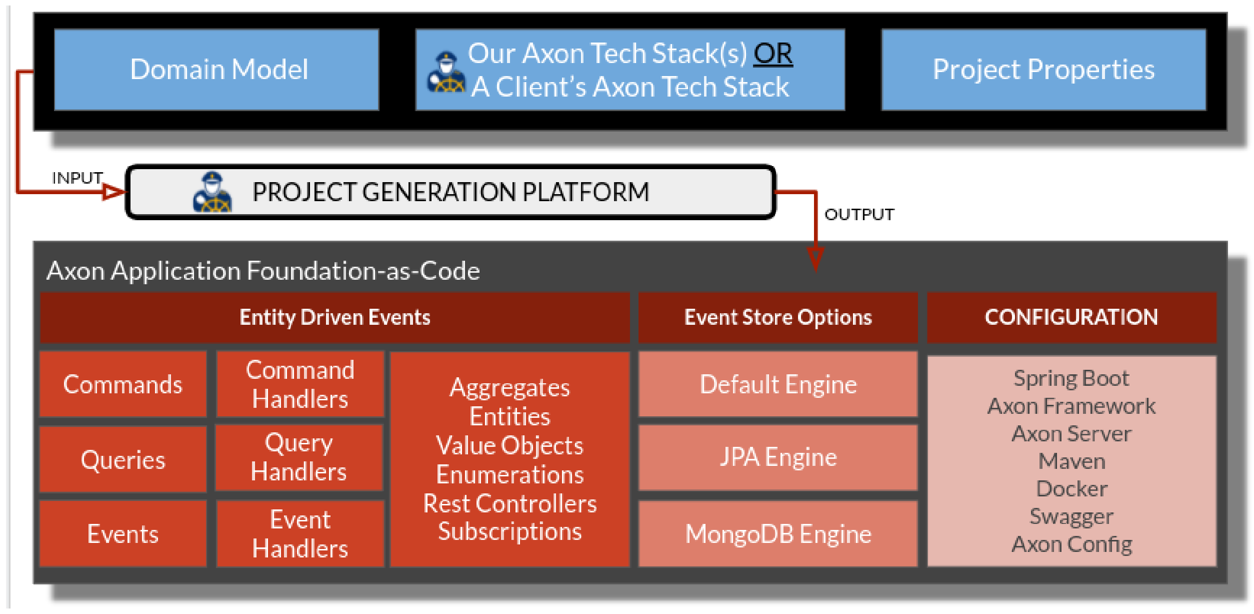
The event model defines all the execution conditions (good and bad) of the railroad system. Between it and our domain model, there is a lot of contextual information about our microservices ecosystem. We already learned Axon provides the API for using and extending the services needed to make our application come to life. The problem is that there is no standard way to quickly turn the things in the domain model into code and configuration to leverage our microservices tech stack, which includes Axon, Spring Boot, a datastore, and more. Even a small domain model requires a significant amount of work to simply represent its core capabilities. This is illustrated as follows:



Enter Harbormaster. Harbormaster is a platform that can ingest a domain model and output fully functional projects. These projects are based on many of today's most popular tech stacks, including an Axon/Spring Boot tech stack.

Instead of starting from scratch, read on to discover how you can start writing the innovation code from day one. Just in case you were wondering, Harbormaster is not a “no-code” or “low-code” platform. It is a “project generation” platform which, at no cost when used with Harbormaster Professional Services, instantly creates the code and files we would have written over the next month or two. If you don’t like what it creates, it’s all template based so everything is easily modifiable.

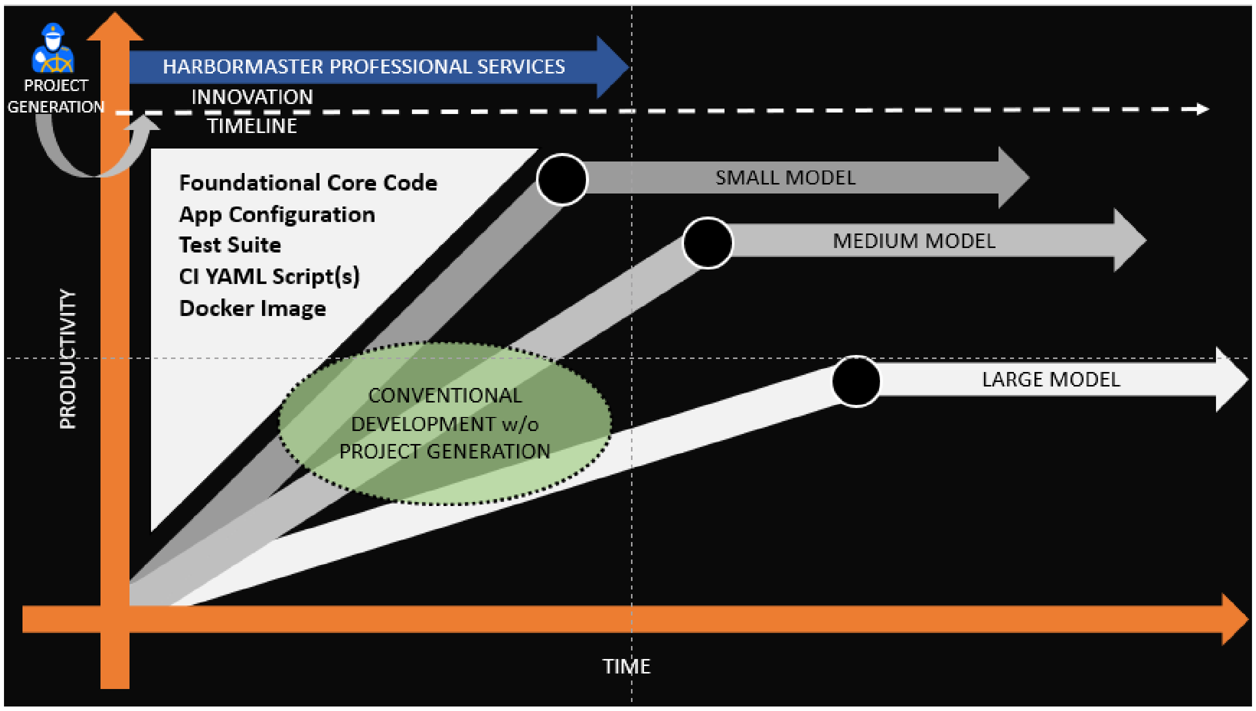
Axon + Harbormaster = Next Gen Dev For Distributed Apps



Speed To Innovation

Event Driven Architecture, Microservices, DDD, and CQRS all demonstrate clear benefits in helping create systems that have great benefits to the business. Unfortunately, there is still significant training and coding required before those benefits are realized.

For any project, the time it takes to start innovating is critical. Innovation is where a system expresses capabilities that make it unique. The innovative services of an application are often what makes the system different from a competitive system. Unfortunately, innovation cannot take place until the foundation aspects of the system have been developed. This usually means weeks to months, or more. Time truly is of the essence where weeks to months could mean the difference between success and failure.



Conclusion

The benefits of project generation on Axon are:

Innovate From The Start

When the core functionality of any application is generated, project development, and more, quickly start with the innovative features of the application. This means more time is spent innovating, failing faster, and so forth. This all leads to a more agile software delivery process, higher customer satisfaction, and more features delivered sooner.

Do More With Fewer Resources

A natural byproduct of project generation is the need for less development resources, especially in the beginning. The makeup of a development team changes with fewer specialists, which presents an opportunity for more generalists, all of which leads to a drastic reduction in cost.

Isolate Complexities

Concepts such as CQRS, event driven architectures, Microservice design patterns, event store and event sourcing all introduce a learning curve. Being able to generate the functional foundation atop a purpose built framework, like Axon, gives time for upskilling of development resources while building out the remaining parts of the application.

In summary, Axon is the first purpose built framework and server to address the needs of today's event driven architectures. Its support for CQRS, events and event sourcing are only a few of the capabilities that make Axon the only platform to address all the requirements of an enterprise grade microservices ecosystem. With Harbormaster, instantly bridge the gap between your model and an instantiation of your model running on a Axon-based technology stack..