33. C4: context, containers, components and classes

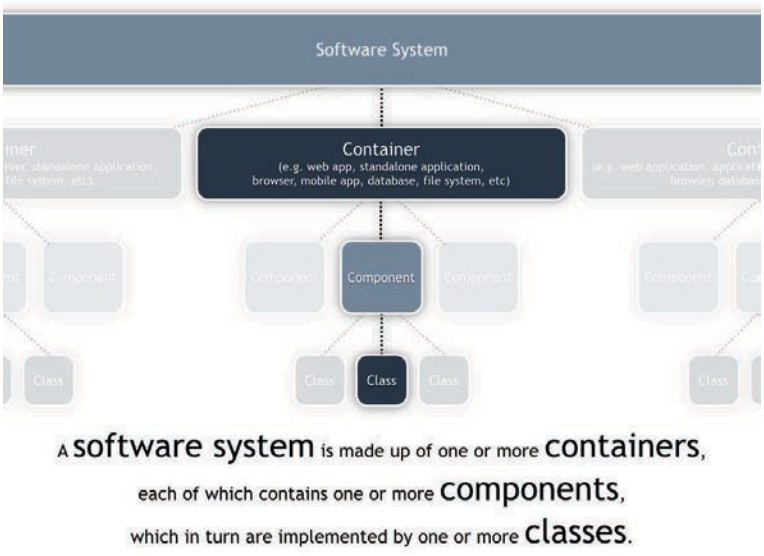
The code for any software system is where most of the focus remains for the majority of the software development life cycle, and this makes sense because the code is the ultimate deliverable. But if you had to explain to somebody how that system worked, would you start with the code?

Unfortunately the code doesn't tell the whole story and, in the absence of documentation, people will typically start drawing boxes and lines on a whiteboard or piece of paper to explain what the major building blocks are and how they are connected. When describing software through pictures, we have a tendency to create a single uber-diagram that includes as much detail as possible at every level of abstraction simultaneously. This may be because we're anticipating questions or because we're a little too focussed on the specifics of how the system works at a code level. Such diagrams are typically cluttered, complex and confusing. Picking up a tool such as Microsoft Visio, Rational Software Architect or Sparx Enterprise Architect usually adds to the complexity rather than making life easier.

A better approach is to create a number of diagrams at varying levels of abstraction. A number of simpler diagrams can describe software in a much more effective way than a single complex diagram that tries to describe *everything*.

A common set of abstractions

If software architecture is about the structure of a software system, it's worth understanding what the major building blocks are and how they fit together at differing levels of abstraction.



A simple model of architectural constructs

Assuming an OO programming language, the way that I like to think about structure is as follows … a software system is made up of a number of containers, which themselves are made up of a number of components, which in turn are implemented by one or more classes. It's a simple hierarchy of logical building blocks that can be used to model most software systems.

- **Classes**: for most of us in an OO world, classes are the smallest building blocks of our software systems.
- **Components**: a component can be thought of as a logical grouping of one or more classes. For example, an audit component or an authentication service that is used by other components to determine whether access is permitted to a specific resource. Components are typically made up of a number of collaborating classes, all sitting behind a higher level contract.
- **Containers**: a container represents something in which components are executed or where data resides. This could be anything from a web or application server through to a rich client application or database. Containers are typically executables that are started as a part of the overall system, but they don't have to be separate processes in their own right. For example, I treat each Java EE web application or .NET website as a separate container regardless of whether they are running in the same physical web server process. The key thing about understanding a software system from a containers

perspective is that any inter-container communication is likely to require a remote interface such as a SOAP web service, RESTful interface, Java RMI, Microsoft WCF, messaging, etc.
- **Systems**: a system is the highest level of abstraction and represents something that delivers value to somebody. A system is made up of a number of separate containers. Examples include a financial risk management system, an Internet banking system, a website and so on.

It's easy to see how we could take this further, by putting some very precise definitions behind each of the types of building block and by modelling the specifics of how they're related. But I'm not sure that's particularly useful because it would constrain and complicate what it is we're trying to achieve here, which is to simply understand the structure of a software system and create a simple set of abstractions with which to describe it.

Summarising the static view of your software

Visualising this hierarchy is then done by creating a collection of system context, container, component and (optionally) class diagrams to summarise the static structure of a software system:

1. **Context**: A high-level diagram that sets the scene; including key system dependencies and actors.
2. **Container**: A container diagram shows the high-level technology choices, how responsibilities are distributed across them and how the containers communicate.
3. **Component**: For each container, a component diagram lets you see the key logical components and their relationships.
4. **Classes**: This is an *optional* level of detail and I will draw a small number of high-level UML class diagrams if I want to explain how a particular pattern or component will be (or has been) implemented. The factors that prompt me to draw class diagrams for parts of the software system include the complexity of the software plus the size and experience of the team. Any UML diagrams that I do draw tend to be sketches rather than comprehensive models.

Common abstractions over a common notation

This simple sketching approach works for me and many of the software teams that I work with, but it's about providing some organisational ideas and guidelines rather than creating

a prescriptive standard. The goal here is to help teams communicate their software designs in an effective and efficient way rather than creating another comprehensive modelling notation.

UML provides both a common set of abstractions **and** a common notation to describe them, but I rarely find teams who use either effectively. I'd rather see teams able to discuss their software systems with a common set of abstractions in mind rather than struggling to understand what the various notational elements are trying to show. For me, a common set of abstractions is more important than a common notation.

Most maps are a great example of this principle in action. They all tend to show roads, rivers, lakes, forests, towns, churches, etc but they often use different notation in terms of colour-coding, line styles, iconography, etc. The key to understanding them is exactly that - a key/legend tucked away in a corner somewhere. We can do the same with our software architecture diagrams.

It's worth reiterating that informal boxes and lines sketches provide flexibility at the expense of diagram consistency because you're creating your own notation rather than using a standard like UML. My advice here is to be conscious of colour-coding, line style, shapes, etc and let a consistent notation evolve naturally within your team. Including a simple key/legend on each diagram to explain the notation will help. Oh, and if naming really *is* the hardest thing in software development, try to avoid a diagram that is simply a collection of labelled boxes. Annotating those boxes with responsibilities helps to avoid ambiguity while providing a nice "at a glance" view.

Diagrams should be simple and grounded in reality

There seems to be a common misconception that "architecture diagrams" must only present a high-level conceptual view of the world, so it's not surprising that software developers often regard them as pointless. Software architecture diagrams should be grounded in reality, in the same way that the software architecture process should be about coding, coaching and collaboration rather than ivory towers. Including technology choices (or options) is usually a step in the right direction and will help prevent diagrams looking like an ivory tower architecture where a bunch of conceptual components magically collaborate to form an end-to-end software system.

A single diagram can quickly become cluttered and confused, but a collection of simple diagrams allows you to effectively present the software from a number of different levels of abstraction. This means that illustrating your software can be a quick and easy task that

requires little ongoing effort to keep those diagrams up to date. You never know, people might even understand them too.

34. Context diagram

A context diagram can be a useful starting point for diagramming and documenting a software system, allowing you to step back and look at the big picture.

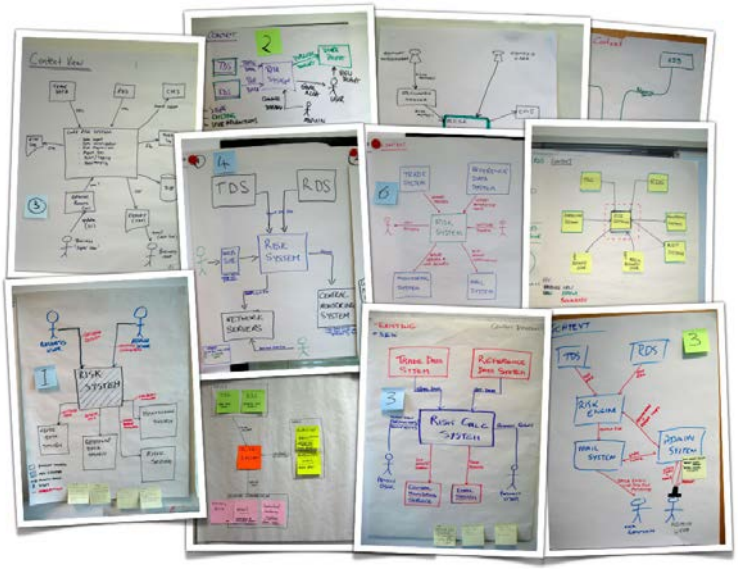
Intent

A context diagram helps you to answer the following questions.

1. What is the software system that we are building (or have built)?
2. Who is using it?
3. How does it fit in with the existing IT environment?

Structure

Draw a simple block diagram showing your system as a box in the centre, surrounded by its users and the other systems that it interfaces with. For example, if you were diagramming a solution to the financial risk system, you would draw the following sort of diagram. Detail isn't important here as it's your wide angle view showing a big picture of the system landscape. The focus should be on people and systems rather than technologies and protocols.



Example context diagrams for the financial risk system (see appendix)

These example diagrams show the risk system sitting in the centre, surrounded by its users and the other IT systems that the risk system has a dependency on.

Users, actors, roles, personas, etc

These are the users of the system. There are two main types of user for the risk system:

- Business user (can view the risk reports that are generated)
- Admin user (can modify the parameters used in the risk calculation process)

IT systems

Depending on the environment and chosen solution, the other IT systems you might want to show on a context diagram for the risk system include:

- Trade Data System (the source of the financial trade data)
- Reference Data System (the source of the reference data)
- Central Monitoring System (where alerts are sent to)
- Active Directory or LDAP (for authenticating and authorising users)

- Microsoft SharePoint or another content/document management system (for distributing the reports)
- Microsoft Exchange (for sending e-mails to users)

Interactions

It's useful to annotate the interactions (user <-> system, system <-> system, etc) with some information about the purpose rather than simply having a diagram with a collection of boxes and ambiguous lines connecting everything together. For example, when I'm annotating user to system interactions, I'll often include a short bulleted list of the important use cases/user stories to summarise how that particular type of user interacts with the system.

Motivation

You might ask what the point of such a simple diagram is. Here's why it's useful:

- It makes the context explicit so that there are no assumptions.
- It shows what is being added (from a high-level) to an existing IT environment.
- It's a high-level diagram that technical and non-technical people can use as a starting point for discussions.
- It provides a starting point for identifying who you potentially need to go and talk to as far as understanding inter-system interfaces is concerned.

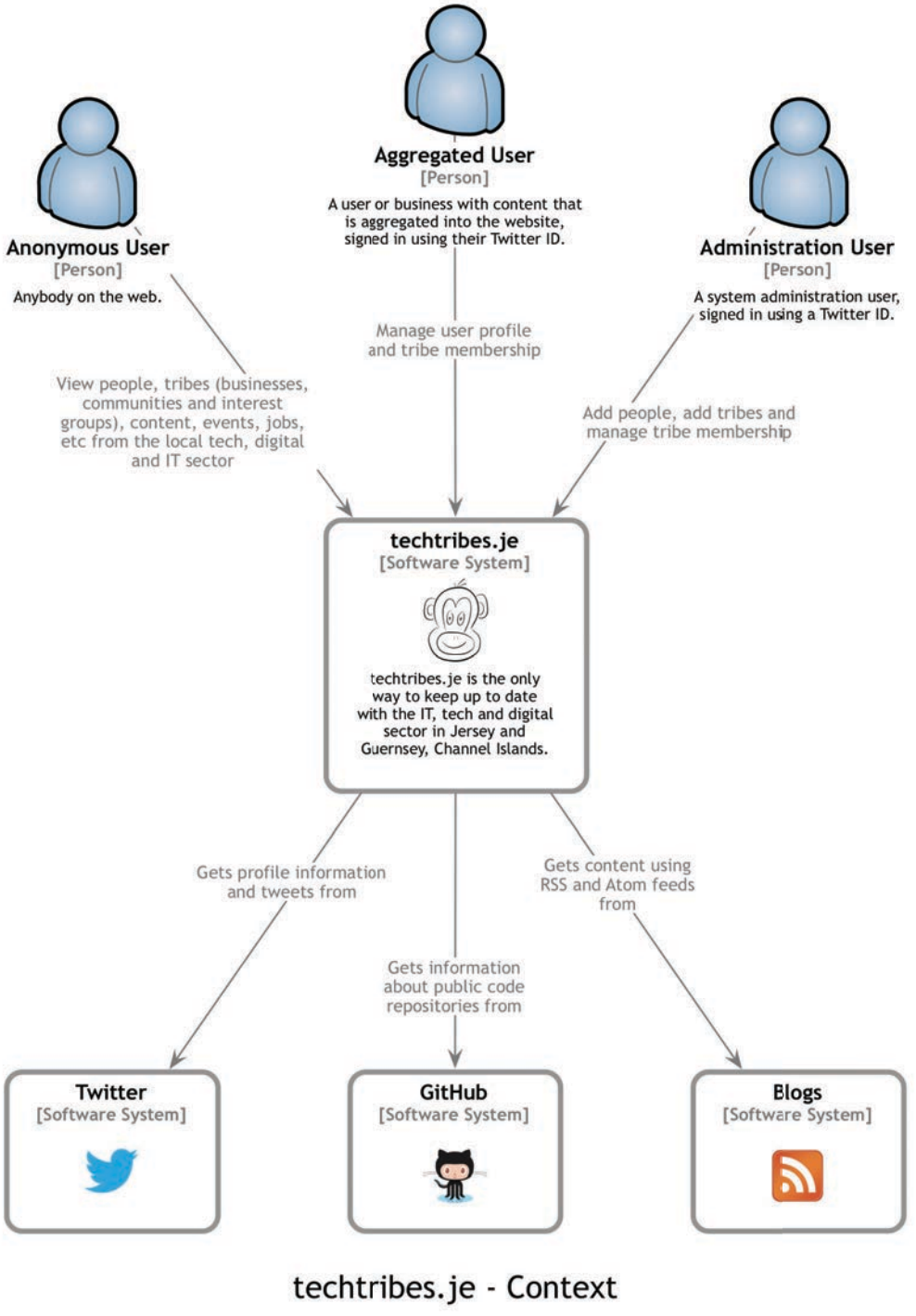
A context diagram doesn't show much detail but it does help to set the scene and is a starting point for other diagrams. Finally, a context diagram should only take a couple of minutes to draw, so there really is no excuse not to do it.

Audience

- Technical and non-technical people, inside and outside of the immediate software development team.

Example

Let's look at an example. The techtribes.je website provides a way to find people, tribes (businesses, communities, interest groups, etc) and content related to the tech, IT and digital sector in Jersey and Guernsey, the two largest of the Channel Islands. At the most basic level, it's a content aggregator for local tweets, news, blog posts, events, talks, jobs and more. Here's a context diagram that provides a visual summary of this.



techtribes.je - Context

Again, detail isn't important here as this is your zoomed out view. The focus should be on people (actors, roles, personas, etc) and software systems rather than technologies, protocols and other low-level details.

35. Container diagram

Once you understand how your system fits in to the overall IT environment with a context diagram, a really useful next step can be to illustrate the high-level technology choices with a container diagram.

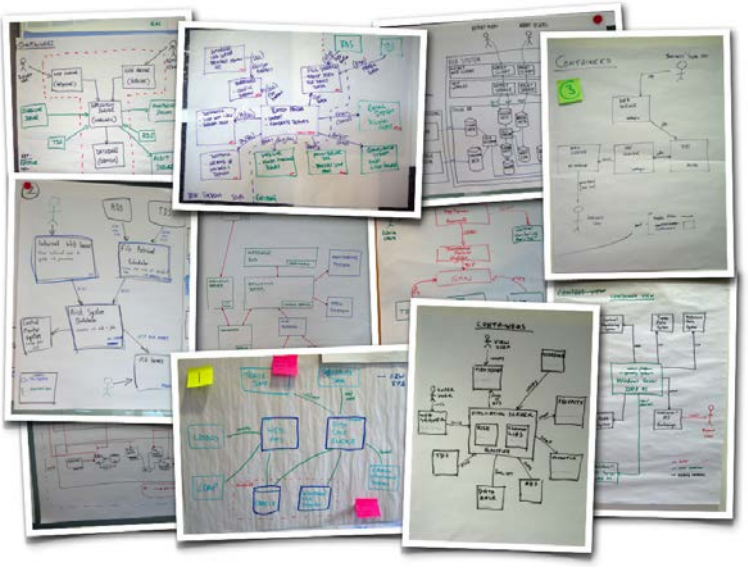
Intent

A container diagram helps you answer the following questions.

1. What is the overall shape of the software system?
2. What are the high-level technology decisions?
3. How are responsibilities distributed across the system?
4. How do containers communicate with one another?
5. As a developer, where do I need to write code in order to implement features?

Structure

Draw a simple block diagram showing your key technology choices. For example, if you were diagramming a solution to the financial risk system, depending on your solution, you would draw the following sort of diagram.



Example container diagrams for the financial risk system (see appendix)

These example diagrams show the various web servers, application servers, standalone applications, databases, file systems, etc that make up the risk system. To enrich the diagram, often it's useful to include some of the concepts from the context diagram diagram, such as users and the other IT systems that the risk system has a dependency on.

Containers

By "containers", I mean the *logical* executables, applications or processes that make up your software system; such as:

- Web servers and applications[1] (e.g. Apache HTTP Server, Apache Tomcat, Microsoft IIS, WEBrick, etc)
- Application servers (e.g. IBM WebSphere, BEA/Oracle WebLogic, JBoss AS, etc)
- Enterprise service buses and business process orchestration engines (e.g. Oracle Fusion middleware, etc)
- SQL databases (e.g. Oracle, Sybase, Microsoft SQL Server, MySQL, PostgreSQL, etc)
- NoSQL databases (e.g. MongoDB, CouchDB, RavenDB, Redis, Neo4j, etc)

[1]If multiple Java EE web applications or .NET websites are part of the same software system, they are usually executed in separate classloaders or AppDomains so I show them as separate containers because they are independent and require inter-process communication (e.g. remote method invocation, SOAP, REST, etc) to collaborate.

- Other storage systems (e.g. Amazon S3, etc)
- File systems (especially if you are reading/writing data outside of a database)
- Windows services
- Standalone/console applications (i.e. "public static void main" style applications)
- Web browsers and plugins
- cron and other scheduled job containers

For each container drawn on the diagram, you could specify:

- **Name**: The logical name of the container (e.g. "Internet-facing web server", "Database", etc)
- **Technology**: The technology choice for the container (e.g. Apache Tomcat 7, Oracle 11g, etc)
- **Responsibilities**: A very high-level statement or list of the container's responsibilities. You could alternatively show a small diagram of the key components that reside in each container, but I find that this usually clutters the diagram.

If you're struggling to understand whether to include a box on a containers diagram, simply ask yourself whether that box will be (or can be) deployed on a separate piece of physical or virtual hardware. Everything that you show on a containers diagram *should* be deployable separately. This doesn't mean that you must deploy them on separate infrastructure, but they should be able to be deployed separately.

Interactions

Typically, inter-container communication is inter-process communication. It's very useful to explicitly identify this and summarise how these interfaces will work. As with any diagram, it's useful to annotate the interactions rather than simply having a diagram with a collection of boxes and ambiguous lines connecting everything together. Useful information to add includes:

- The purpose of the interaction (e.g. "reads/writes data from", "sends reports to", etc).
- Communication method (e.g. Web Services, REST, Java Remote Method Invocation, Windows Communication Foundation, Java Message Service).
- Communication style (e.g. synchronous, asynchronous, batched, two-phase commit, etc)
- Protocols and port numbers (e.g. HTTP, HTTPS, SOAP/HTTP, SMTP, FTP, RMI/IIOP, etc).

System boundary

If you do choose to include users and IT systems that are outside the scope of what you're building, it can be a good idea to draw a box around the appropriate containers to explicitly demarcate the system boundary. The system boundary corresponds to the single box that would appear on a context diagram (e.g. "Risk System").

Motivation

Where a context diagram shows your software system as a single box, a container diagram opens this box up to show what's inside it. This is useful because:

- It makes the high-level technology choices explicit.
- It shows where there are relationships between containers and how they communicate.
- It provides a framework in which to place components (i.e. so that all components have a home).
- It provides the often missing link between a very high-level context diagram and (what is usually) a very cluttered component diagram showing all of the logical components that make up the entire software system.

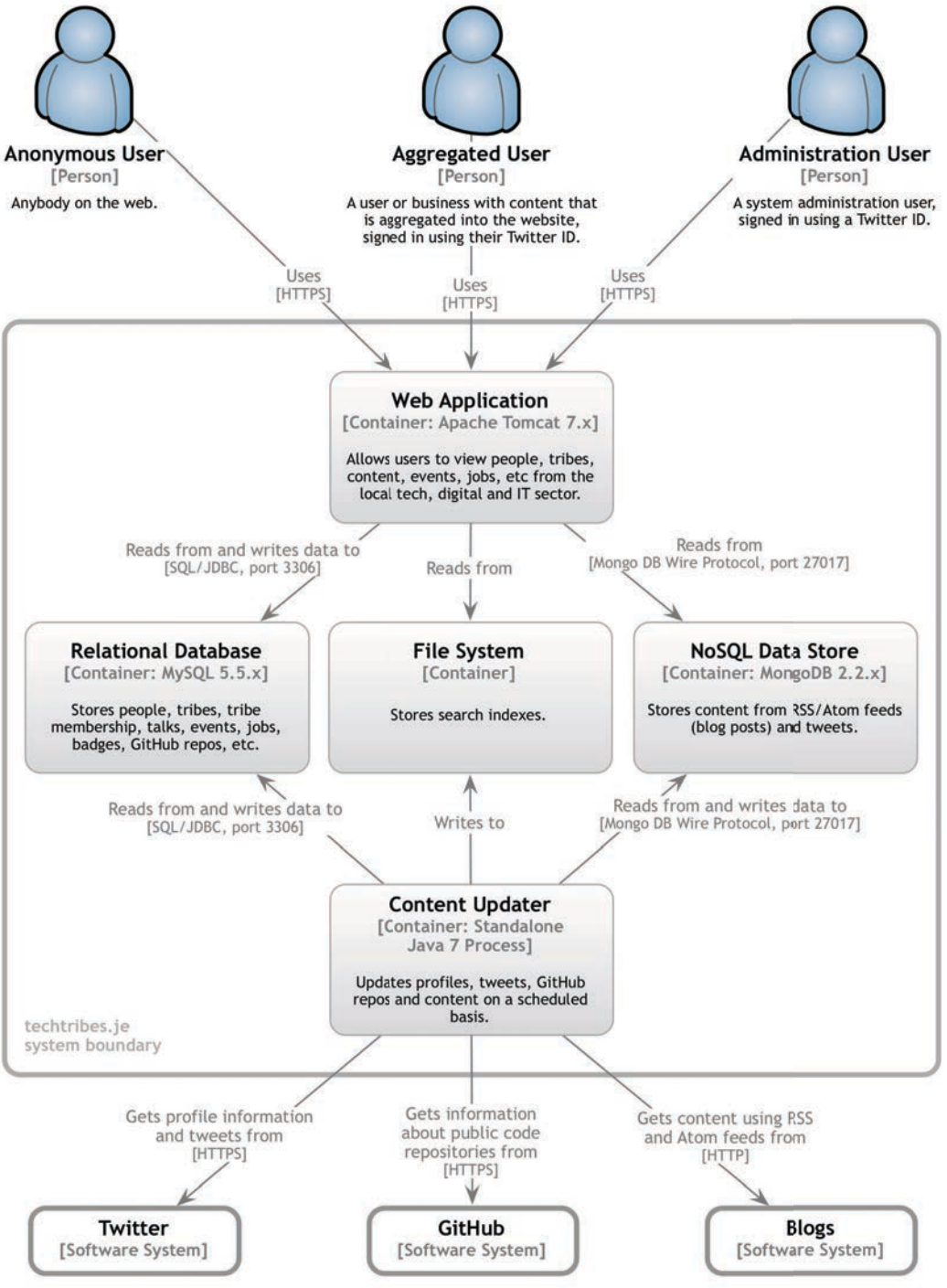
As with a context diagram, this should only take a couple of minutes to draw, so there really is no excuse not to do it either.

Audience

- Technical people inside and outside of the immediate software development team; including everybody from software developers through to operational and support staff.

Example

The following diagram shows the logical containers that make up the techtribes.je website.



techtribes.je - Containers

Put simply, techtribes.je is made up of an Apache Tomcat web server that provides users with information, and that information is kept up to date by a standalone content updater process. All data is stored either in a MySQL database, a MongoDB database or the file system. It's worth pointing out that this diagram says nothing about the number of physical instances of each container. For example, there could be a farm of web servers running against a MongoDB cluster, but this diagram doesn't show that level of information. Instead, I show physical instances, failover, clustering, etc on a separate deployment diagram. The containers diagram shows the high-level shape of the software architecture and how responsibilities are distributed across it. It also shows the major technology choices and how the containers communicate with one another. It's a simple, high-level technology focussed diagram that is useful for software developers and support/operations staff alike.

36. Component diagram

Following on from a container diagram showing the high-level technology decisions, I'll then start to zoom in and decompose each container further. How you decompose your system is up to you, but I tend to identify the major logical components and their interactions. This is about partitioning the functionality implemented by a software system into a number of distinct components, services, subsystems, layers, workflows, etc. If you're following a "pure Object Oriented" or Domain-Driven Design approach, then this may or may not work for you.

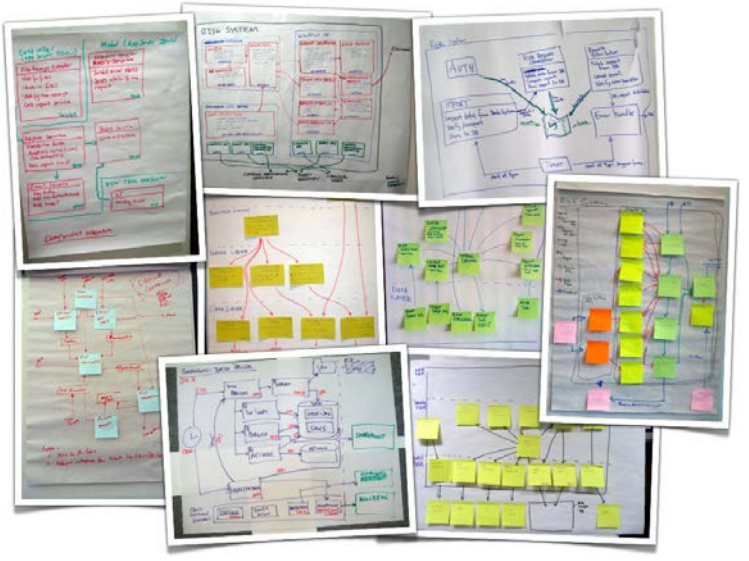
Intent

A component diagram helps you answer the following questions.

1. What components/services is the system made up of?
2. It is clear how the system works at a high-level?
3. Do all components/services have a home (i.e. reside in a container)?

Structure

Whenever people are asked to draw "architecture diagrams", they usually end up drawing diagrams that show the logical components that make up their software system. That is basically what this diagram is about, except we only want to see the components that reside within a *single* container at a time. Here are some examples of component diagrams if you were designing a solution to the financial risk system.



Example component diagrams for the financial risk system (see appendix)

Whenever I draw a component diagram, it typically only shows the components that reside within a single container. This is by no means a rule though and, for small software systems, often you can show all of the components across all of the containers on a single diagram. If that diagram starts to become too cluttered, maybe it's time to break it apart.

Components

If you were designing a solution to the financial risk system, you might include components like:

- Trade data system importer
- Reference data system importer
- Risk calculator
- Authentication service
- System driver/orchestrator
- Audit component
- Notification component (e.g. e-mail)
- Monitoring service
- etc

These components are the coarse-grained building blocks of your system and you should be able to understand how a use case/user story/feature can be implemented across one or more of these components. If you can do this, then you've most likely captured everything. If, for example, you have a requirement to audit system access but you don't have an audit component or responsibilities, then perhaps you've missed something.

For each of the components drawn on the diagram, you could specify:

- **Name**: The name of the component (e.g. "Risk calculator", "Audit component", etc).
- **Technology**: The technology choice for the component (e.g. Plain Old [Java|C#|Ruby|etc] Object, Enterprise JavaBean, Windows Communication Foundation service, etc).
- **Responsibilities**: A very high-level statement of the component's responsibilities (e.g. either important operation names or a brief sentence describing the responsibilities).

Interactions

To reiterate the same advice given for other types of diagram, it's useful to annotate the interactions between components rather than simply having a diagram with a collection of boxes and ambiguous lines connecting them all together. Useful information to add the diagram includes:

- The purpose of the interaction (e.g. "uses", "persists trade data through", etc)
- Communication style (e.g. synchronous, asynchronous, batched, two-phase commit, etc)

Motivation

Decomposing your software system into a number of components is software design at a slightly higher level of abstraction than classes and the code itself. An audit component might be implemented using a single class backing onto a logging framework (e.g. log4j, log4net, etc) but treating it as a distinct component lets you also see it for what it is, which is a key building block of your architecture. Working at this level is an excellent way to understand how your system will be internally structured, where reuse opportunities can be realised, where you have dependencies between components, where you have dependencies between components and containers, and so on. Breaking down the overall problem into a number of separate parts also provides you with a basis to get started with some high-level estimation, which is great if you've ever been asked for ballpark estimates for a new project.

A component diagram shows the logical components that reside inside each of the containers. This is useful because:

- It shows the high-level decomposition of your software system into components with distinct responsibilities.
- It shows where there are relationships and dependencies between components.
- It provides a framework for high-level software development estimates and how the delivery can be broken down.

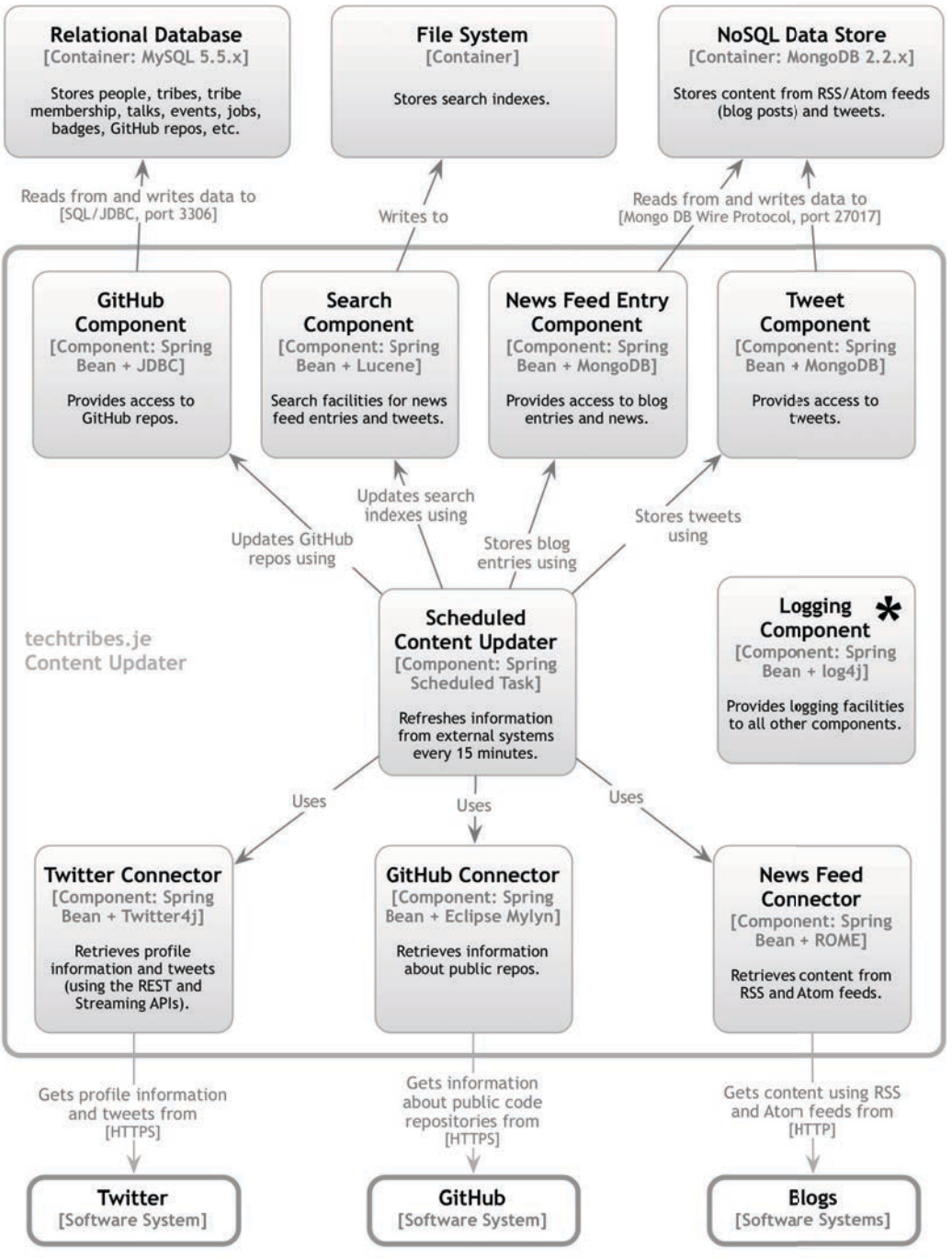
Designing a software system at this level of abstraction is something that can be done in a number of hours or days rather than weeks or months. It also sets you up for designing/coding at the class and interface level without worrying about the overall high-level structure.

Audience

- Technical people within the software development team.

Example

As illustrated by the container diagram, techtribes.je includes a standalone process that pulls in content from Twitter, GitHub and blogs. The following diagram shows the high-level internal structure of the content updater in terms of components.



techtribes.je - Components - Content Updater

* Used by all components