Contents

List of Figures

List of Tables

1. Introduction (5 Pages)
    1.1. Who are HubSpot - 2
    1.2. The Email Sending Infrastructure Team - 2 (team structure (what a TL is), what they all do, Hubspot small team methodology, cross team work)
    1.3. What the team owns (products)
    1.4. Goals of the team or something - 1
2. HubSpot Methodology (10 Pages)
    2.1. Java MicroServices (unit, integ, accept)- 5
    2.2. React Native / Redux front end - 4
    2.3. Team Process
        2.3.1. Task Management
        2.3.2. Ops Review (System Health)
        2.3.3. Code Walkthroughs
        2.3.4. Inter Team Coordination
3. Technologies Used (26 pages, probably excessive)
    3.1. Java 8 Features (streams, CF, lambdas, Executors) - 5
    3.2. Kafka - 5
    3.3. Hadoop - 3
    3.4. DNS - 3
    3.5. SMTP - 2
    3.6. MySql (InnoDB) - 3
    3.7. HBase (Sync, Idempot, Locks) - 3
    3.8. ZooKeeper (+ Circus) - 2
    3.9. gRPC + Proto
4. Major Projects Undertaken (20)
    4.1. Rebuilt DNS management - 5
    4.2. Auto DNS for dedicated accounts - 5
    4.3. EmailMtaSending Kafka stuff - 5
    4.4. Minimize CIDR IP selection alg - go into detail about this (eg talk about SPF records in email, CIDr notation and diagram out the alg, performance etc)- 3 pages

CHECK THIS - I double we need these headings.. There are a number of chapters that you must have: an introduction; a background or literature review chapter; and

a conclusion chapter. The focus of the other chapters will depend on your specific project.

Introduction

What is HubSpot?

Inbound Marketing All hubspot products

Email at HubSpot

Teams, cusomter POV etc etc

Team Management and Team Process

As with any type of team, it is critical that the team is managed appropriately. Blah blah blah..

Issue Tracking

Issue tracking through GitHub, cross team colab, closing issues with PRs,

Task Managemenet

TODO: Link this with above issue tracking In order to maximize productivity and maintain a healthy workload, a system must be put in place which dictates how team members are assigned units of work. The method used by the Email Sending Infrastructure team, was to split tasks up into the following five stages:

Backlog
: This contains tasks which are of low priority and can be deferred until a later date

Design
: This contains tasks which require a considerable amount of planning before being undertaken. Tasks in this stage often result in discussions amongst the team about how best to tackle the task.

Ready
: Tasks which are sufficiently specified that they can be undertaken. More complex tasks that were once in the design category land here once all of the corresponding details have been decided upon. These are tasks which engineers on the team should choose as a next task upon completion of a task.

In-Progress
: This contains tasks which are currently being worked on (usually) by a single engineer.

Completed
: Tasks which have been completed this week.

All of the tasks are managed through an online platform called Waffle(4) which presents a *"Waffle Board"* which displays the tasks in each of the above stages. The team concludes each week with a meeting in which the board is inspected and tasks are moved to new stages as appropriate. This provides a great mechanism for monitoring

the team's productivity and ensuring that the team is focused on the most important tasks.

Communication

Company wide slack, channel per team, support channels, searchable.

Proactive Ops Reviews

Code Walkthroughs

Critical Situation Post Mortums

Daily Standups

1 to 1s

Might not include this

Software Development at HubSpot

Software development has been a core part of HubSpot since day one. As such, the overall development process and methodology has undergone several iterations. Critically, at each iteration, the development process is reevaluated and the experience gained over the past number of years is capitalized upon. This has lead to a streamlined and simple development process which is easy to learn, allowing both new hires and temporary interns to get up to speed quickly. HubSpot also has several *platform* teams soley dedicated to providing easy to use integrations of powerful tools which can be combersome to configure on a per project basis. These teams contribute to what's known as a the overall *platform as a service (PaaS)*. Product development teams can then use this platform to get access to these powerful tools, without needing to spend time on configuration.

As all of HubSpot is built on the same technology stack and all projects adhere to a common project structure, any developer can drop into any project owned by another team and immediately know where to search for what they are looking for. This has the direct benefits of allowing engineers to fix bugs they encounter when using other teams' projects. One of HubSpot's core beliefs when it comes to software engineering is that everyone should contribute what they can, instead of passing the blame to other teams. Most engineers are busy with new tasks and challenges and fixing a bug in code that belongs to a fellow engineer is substantially more productive for **both** teams than filing an issue and waiting for it to be resolved.

Technologies in use at HubSpot

As discussed, all of HubSpot is built using a set of technologies. This set of technologies is continuously growing as needs change and as new technologies become available. Typically any given team and/or project will use a subset of the technologies on offer. This section outlines some of the core technologies used by the Email Sending Infrastructure team.

Java 8

Java 8 is the second to newest release of Java. Java as a language has been around since the mid 1990s and has been under continuous development since then. As

such, it has become a very stable and reliable language, used by nine million developers world wide (5). At HubSpot, Java 8 is used for the development of all backend services. The architecture employed is one of micro-services. This allows for rapid development and deployment of many loosely coupled services. The services are extremely modular and are often used by a variety of teams and projects.

Java is an object oriented language which provides a variety of concepts to aid in system design. Classically, the core of object oriented programming is that entities (objects) should model one *real* entity and one only. Objects can then be used inside of other classes, abstracting away all of the complex implementation logic of the class in use from the user. Any information required by the object can be encapsulated within the object itself allowing the object to provide a simple interface to it's users, coinsisting of a number of methods, which if named correctly, provide a succinct description of what the method does which lines up exactly with what the user thinks the method should do.

Java 8 was released in March of 2014 and provides several new and extremely powerful features, many of which are used on a daily basis at HubSpot. Aside from all of the core functionality contained in Java, a subset of the most interesting and useful features it provides are outlined below.

Lambda Expressions

Often times it is useful to create classes which wrap a piece of logic or code. Similarly, systems often require the execution of a piece of code in response to a certain event (e.g. running some code in response to a mouse click). Traditionally in java this was accomplished by writing a manual *functional interface*. There functional interfaces were simple interfaces which provided a contract containing a single method. The class class in question can then be passed an instance of an object which implements the functional interface and can thus invoke the method defined in the interface when appropriate. An example of this using traditional java is outlined in code listing 3.1 which contains code for a button that can be clicked and can have on-click methods associated with it. Prior to Java 8, this code was cumbersome to write requiring a custom interface to be created, implemented by a concrete class and its implementation instantiated and passed to the class.

Listing 3.1: onClick Listener without Lambda Expression

```
1  public class Button {
2     private OnClickRunner onClickRunner;
3     private String buttonName;
4
5     public Button(OnClickRunner onClickRunner,
6                   String buttonName) {
7        this.onClickRunner = onClickRunner;
8        this.buttonName = buttonName;
9     }
10
11    public void click() {
12       onClickRunner.runOnClick(buttonName);
13    }
```

```
14
15      public static void main(String[] args) {
16          Button button = new Button(new ButtonClickLogger(), "SEARCH");
17          button.click();
18      }
19  }
20
21  interface OnClickRunner {
22      void runOnClick(String buttonName);
23  }
24
25  class ButtonClickLogger implements OnClickRunner {
26      @Override
27      public void runOnClick(String buttonName) {
28          System.out.println("Button " + buttonName + " was clicked.");
29      }
30  }
```

This code can be written in a much more concise form by using Java 8's new lambda expressions. The JDK now provides the most common functional interfaces which can be used in place of custom functional interfaces. For example the Consumer<T> functional interface, defines a single accept function which takes an argument of type T and returns nothing (it *consumes* the argument), which is exactly what our OnClickRunner defined. The Button class can be refactored to use a Consumer<String>, allowing the logic of the OnClickRunner to be directly specified through a lambda expression. An example of this is shown in code listing 3.2 and the lambda expression can be seen on line 16.

Listing 3.2: onClick Listener with Lambda Expression

```
1   public class Button {
2       private Consumer<String> onClickConsumer;
3       private String buttonName;
4       public Button(Consumer<String> onClickConsumer,
5                     String buttonName) {
6           this.onClickConsumer = onClickConsumer;
7           this.buttonName = buttonName;
8       }
9
10      public void click() {
11          onClickConsumer.accept(buttonName);
12      }
13
14      public static void main(String[] args) {
15          Button button = new Button(
16                  name -> System.out.println("Button " + name + " was clicked"),
17                  "SEARCH"
18          );
19
20          button.click();
21      }
22  }
```

Streams

Another extremely powerful feature of Java 8 is the new Stream API. This allows for the bulk processing of collections through map/reduce like operations. Performing arbitrary data manipulation on a collection of Java objects is extremely common. Typically this could be accomplished using a simple loop. However this often requires the introduction of several local variables which can add excessive noise to code. Of course, some operations are still better suited to a simple loop, particularly if the data transformation function has side effects, in which case it is impossible to use streams. However it is widely considered good practice to minimize side effects of functions in order to maintain simplicity and making heavy use of streams is a great way to remind developers not to introduce side effects and to in general, minimize the amount of state required by a class. The Java 8 Stream API is fluent, allowing for arbitrarily complex stream operations to be chained together. Streams are also evaluated lazily, minimizing the amount of work to be done and can be parallelized internally by the API, providing excellent performance. The Java 8 Stream API consists of three types of operations:

Initial stream call	The `stream()` call can be invoked on any Java collection of objects. This call returns a `Stream<T>` where T is the type of the objects in the collection, allowing subsequent intermediate and terminal operations to be invoked on the returned stream.
Intermediate Operations	These are the operations which perform the data transformation. There are a variety of intermediate operations provided such as `sort` which sorts the objects in the stream, `filter` which filters objects in the stream according to some predicate and map which maps an arbitrary function over each object in the stream. As mentioned, the Stream API is fluent, meaning multiple intermediate operations can be chained together (for example filtering the objects and then sorting them).
Terminal Operations	The is the final operation which describes how the data should be reduced to a single object (which may be a collection). Common terminal operations are max which returns the maximum of the objects in the stream, `findFirst` which returns the first object the stream encounters that matches a given predicate and `collect` which defines how the objects in the stream should be collected into a collection (for example collecting the objects into a set would remove any duplicates)

Comparing code listing 3.3 and code listing 3.4, the clarity of the code produced using the Streams API can be seen. The code shows two approaches to a piece of code which returns the length of each of the strings (in ascending order) without whitespace and which don't contain the word *owl*. Although this is a toy example, several benefits of the Streams API can be seen. The code using streams (code listing 3.4) reads like the steps of an recipie, clearly stating what is performed at each step. However the code using the traditional `for` loop (code listing 3.3), requires `if` statements and

redundant local variables, distracting the programmer from the core steps of the algorithm. Streams also provide the benefits of immutability and parallelism for free.

Listing 3.3: Batch Processing without Streams

```
List<String> words = Arrays.asList(" cats", "dogs ", "OWLS", "frogs");
  List<Integer> results = new ArrayList<>();
  for (String word : words) {
     word = word.trim();
     word = word.toLowerCase();
     if (!word.contains("owls")) {
        results.add(word.length());
     }
  }
  results.sort(Comparator.naturalOrder());
  return results;
```

Listing 3.4: Batch Processing with Streams

```
List<String> words = Arrays.asList(" cats", "dogs ", "OWLS", "frogs");
return words.stream()
  .map(String::trim)
  .map(String::toLowerCase)
  .filter(word -> !word.contains("owls"))
  .map(String::length)
  .sorted()
  .collect(Collectors.toList());
```

Completable Futures

Asynchronus programming is present in most if not all modern systems. In the early days of Java, this was accomplished by the JDK through abstractions at the thread level. This required careful tracking of the state of threads by the programmer. Concurrent programs are incredibly difficult to reason about and thus concurrency is one of the most challening aspects of modern software engineering and is the source of a huge number of bugs. However the benefits of concurrent programming are extremely obvious, essentially making it a nescesity in modern systems. Thus any abstractions that can aid in reducing the number of things a programmer must keep track of will be beneficial. In Java 8's case, this abstraction is the CompletableFuture API. This API allows for programmers to perform asynchronus tasks by specifying a `Supplier<U>`, a function which takes no arguments and returns (supplies) a value of type U, which will return a value at some point in the future. Thus what's returned from this call is not an instance of type U, but a `CompletableFuture<U>`, that is, an object that at some point in the future will contain an instance of type U (provided no errors occur).

The programmer may also specify an `Executor` (6) which is essentially an object (e.g. threadpool) capable of running tasks on a thread other than the thread in the current context. If no `Executor` is specified, the task is automatically submited to Java's work stealing `ForkJoinPool` (7).

The API provides a simple `get` method for blocking until the return value is present (or

throws an exception). More interestingly however, it also provides several methods to specify subsequent processing of the return value when it returns. A subset of these methods are the following (8):

thenApply — This method is used to supply a function that should be run upon completion of the underlying CompletableFuture. The result of the CompletableFuture will be passed as the sole argument to this function. This function is free to return any type and in doing so, sets the type associated with the underlying CompletableFuture.

thenCompose — This method is very similar to thenApply except it is used when the desired function to be run is also asynchronus (that is, it too returns a completable future). This behaviour could technically also be handled by /javathenApply (as it is free to return any value), but this would cause the return type of the parent CompletableFuture to itself be a CompletableFuture. The benefit of thenCompose is that it contains logic to unwrap this nested CompletableFuture, allowing the return type of the parent to remain a CompletableFuture<T> even if the supplied function is asynchronus.

thenAccept — This method is used when the return type of the CompletableFuture only needs to be used in the registered callback and is not used outside of the supplied function. As such, this method takes a Consumer<T> as its argument, that is, a function which takes T as a parameter and does not return anything.

Guice - Dependency Injection

Google's Guice (9) is a dependency injection framework for Java. Dependency injection is a software pattern which abstracts away the actual construction (instantiation) of objects from the user. As a system grows and abstractions are built atop one and other, simply instantiating an object can be come tedious and difficult. Traditionally, in order to instantiate an object, all of its unconditionally required dependencies must be passed to the constructor of the object. Otherwise the object could be left in an inconsistent state. Thus in order to instantiate an object O, all of its dependencies, for example X, Y and Z must be provided to O's constructor. Thus the client of O must first instantiate instances of X, Y and Z before it can use O. However X may have it's own set of dependencies and the problem simply gets worse and worse. An example of the difficulties this can cause (based on the example in Guice's documentation (10)) is shown in code listing 3.5. The code shown has to recursively create each of the dependencies for each of it's dependencies which can quickly get out of hand for large systems.

Listing 3.5: Pizza Ordering Service with no Dependency Injection

```
1 public class PizzaOrderingService {
2   private SqlTransactionLogger transactionLogger;
3   private AccountsDao AccountsDao;
4   private PayPalCustomerBiller customerBiller;
5
```

```
6   public PizzaOrderingService() {
7     SqlCredentials sqlCredentials = new SqlCredentials("dbName",
          "password");
8     SqlConnection sqlConnection = new SqlConnection(sqlCredentials);
9     PayPalCredentials payPalCredentials = new
          PayPalCredentials("accountId", "password");
10    this.transactionLogger = new SqlTransactionLogger(sqlConnection);
11    this.accountsDao = new AccountsDao(sqlConnection);
12    this.customerBiller = new PayPalCustomerBiller(payPalCredentials);
13  }
14 }
```

Traditionally, this problem was somewhat helped (but not entirely solved) by the using the factory pattern (11). Depending on the implementation, this can ease the pain of getting access to objects the client depends on by allowing the factory to contain the logic for the instantiation of thee concrete class' dependencies. However this still requires the manul implementation of the factory classes themselves, leaving much to be desired.

Dependency injection solves this problem by allowing fully formed instances of dependencies (eg a `CustomerBiller`) to be *passed* to the client, removing the need for the client to instantiate the object themselves. Client's simply ask for their dependencies to be injected into their constructor. Guice provides this functionality by simply annotating the constructor with the `@Inject` annotation. This (along with a some other boilerplate) informs Guice that this class should have it's dependencies injected into the classes constructor. Guice accomplishes this by building an arbitrarily complex dependency graph at run time. When a class needs a certain dependency, the graph can be examined in order to figure out how to instantiate that dependency.

However Guice does need a starting point - dependencies can't just be injected for every single class without providing some initial classes in which to build upon. In the example above, these base classes would be the `SqlCredentials` and `PayPalCredentials` classes. These classes should not have any injected dependencies. In this example these could simply read the credentials needed from a file. Guice allows us to add vertices to the dependency graph by *providing* objects using the `@Provides` annotation. Thus both `SqlCredentials` and `PayPalCredentials` would need to be `@Provided`. The resulting object graph from the code above is show in 3.1 - notice the classes at the bottom of the hierarchy are provided.

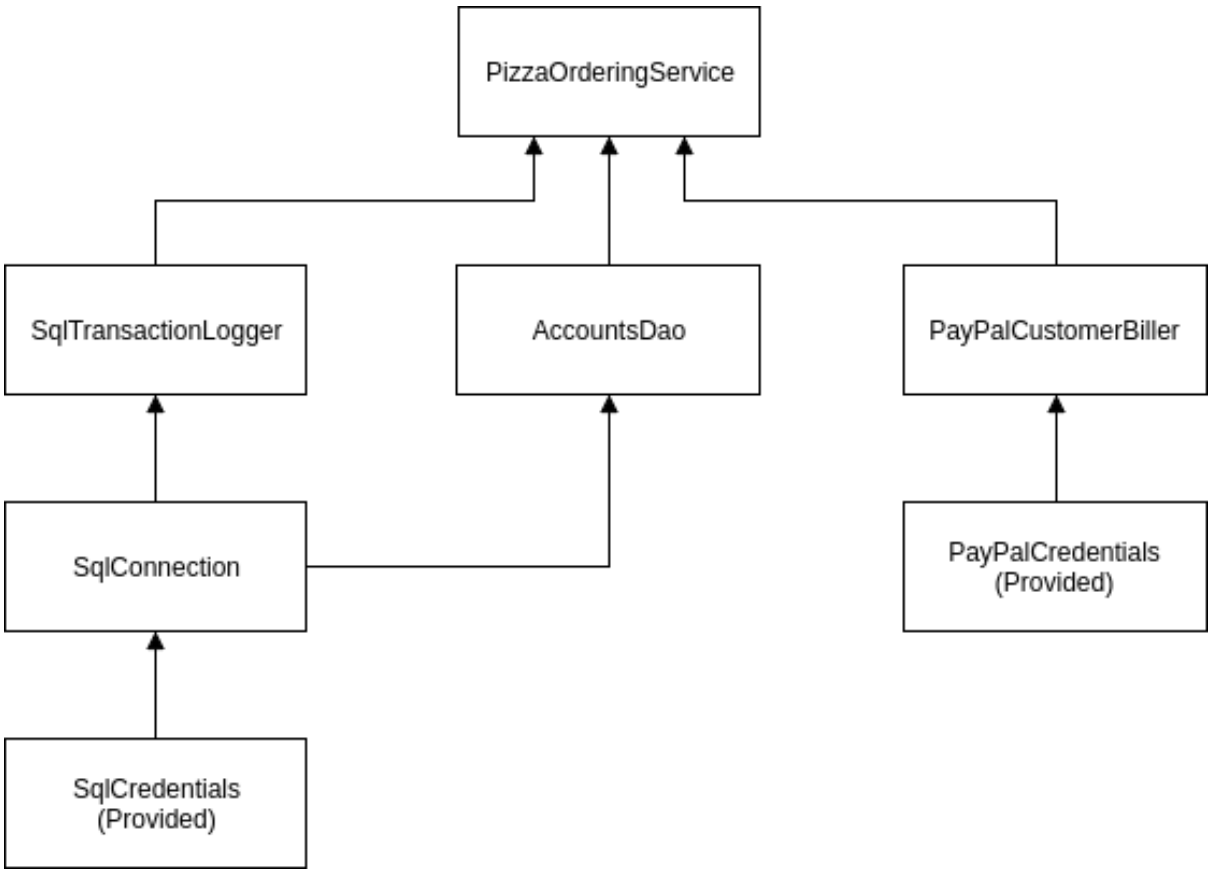

Figure 3.1: Dependency Graph built by Guice

This leads to extremely simple code for the PizzaOrderingService in which the logic is entirely separated from the dependency management.

Listing 3.6: Pizza Ordering Service using Dependency Injection

```
1  public class PizzaOrderingService {
2    private SqlTransactionLogger transactionLogger;
3    private PayPalCustomerBiller customerBiller;
4    private AccountsDao accountsDao;
5
6    @Inject
7    public PizzaOrderingService(SqlTransactionLogger transactionLogger,
8                                PayPalCustomerBiller customerBiller,
9                                AccountsDao accountsDao) {
10     this.transactionLogger = transactionLogger;
11     this.customerBiller = customerBiller;
12     this.accountsDao = accountsDao;
13   }
14 }
```

Aside from abstracting away the dependency instantiation, dependency injection also has the added benefit in that all of a classes dependencies become part of the classes signature. There are no required dependencies that are buried within the implementation logic. This makes the code much simpler to test. The PayPalCustomerBiller used by the class is essentially hardwired into the code that does not use dependency

injection (code listing 3.5), meaning there is no way to test this class without actually billing a (fake) customer. However as the PayPalCustomerBiller used by the dependency injected code (code listing 3.6), a mock of this object (which skips the actual billing, but returns results as if it actually billed a customer) can be provided to the class and used for testing purposes.

Immutables - Immutability for Java

Kafka

MySql

Indexes, liquibase, InnoDB

HBase

ZooKeeper (Circus)

Hadoop

gRPC and Protobuf

Email Specific Technologies

DNS

SMTP

System Management and Health Monitoring

Should this be in technologies used, focus this on how it helps the team manage the system While working on projects of this magnitude, bugs and issues are an inevitability. The amount of traffic seen by these systems compounds any small issues or bugs present in the system. As such, it is critcal to have systems in place which monitor the health of the system and inform the team of any potential issues with the system. Whatsmore, these issues must be continuously examined and remedial action must be taken where applicable. The Email Sending Infrastructure team made use of several tools and methods for monitoring the health of their systems, a subset of which are outlined below:

Log4j2(1)

Log4j2 is a Java framework which is provides facilities for logging to different log levels and advanced log filtering (for example with regular expressions). This provides an excellent facility for understanding why systems are behaving unexpectedly in production. A common pattern is to insert log messages to a low priority log level (eg DEBUG) which describe the state of the system or the code path taken. Typically when the system is behaving normally, a highger priortiy log level is set (eg INFO) meaning these finer grained log messages are skipped. However, should an issue arise, the log level can then be easily switched to the lower priority temporarily to get a more detailed insight into why the system is misbehaving. This pattern allows detailed log messages to be produced only when they are needed, reducing the amount of noise present in the logs. The framework also provides the ERROR debug mode which can log error messages as uncaught exceptions without killing the currently executing thread.

Sentry(2)

Sentry is an online platform which logs uncaught exceptions that arise during program execution. This greatly simplifies the task of finding out the reason for a system fault or failure without the need to trawl through pages of log files. Sentry logs the full stack trace associated with an exception, the time of occuence and other pieces of meta data such as the name of the deployable. It uses this data to monitor the occurences of particular exceptions over time, provides facilities for opening and closing GitHub issues and most importantly, to send an email to all those subscribed to the project (eg the Email Sending Infrastructure in this case) when an exception occurs. Sentry proves to be extremely useful at deploy time. Obviously when deploying new code to production servers, one must be sure that the changes did not cause the system to enter an unhealthy state. Provided the code is well written and that unexpected exceptions that occur are not silently swallowed, Sentry can be monitored at deploy time in order to help provide the engineer with confidence that the deployed changes were non breaking. Sentry also provides support for integrating into the aforementioned Log4j2. Sentry can monitor ERROR level log messages that are produced by Log4j2 and subsequently log these error messages to sentry. As an engineer this combination of tools is extremely useful for indicating that the system has found an issue, without killing the thread. This is ideal in cases where some work has been done and the system has encountered a critical error, but does not need to be restarted. This mitigates the need to repeat the work, but still enforms the team that an error has occured by logging an exception to Sentry.

SignalFX(3)

Tasks Undertaken

Throughout the course of the internship, a variety these tasks were undertaken and completed. For the sake of brevity a small subset of the most interesting of these tasks are outlined below.

Rebuilt DNS Management System

Auto DNS for dedicated IPs

EmailMtaSending Kafka

Minimize CIDR IP selector alg

Bibliography

[1] The Apache Software Foundation. Log4j2 - java logging framework. URL `logging.apache.org/log4j/2.x`.

[2] Inc. Functional Software. Sentry - error tracking software. URL `sentry.io`.

[3] SignalFx. Signalfx: Cloud monitoring for the enterprise. URL `signalfx.com`.

[4] CA Technologies. Waffle - developer-first project management for teams on github. URL `waffle.io`.

[5] Timothy Beneke and Tori Wieldt. Javaone 2013 review: Java takes on the internet of things. 2013. www.oracle.com/technetwork/articles/java/afterglow2013-2030343.html.

[6] Oracle. Java 8 executor documentation, . docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executor.html.

[7] Oracle. Java 8 forkjoinpool documentation, . docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html.

[8] Tomasz Nurkiewicz. Java 8 definitive guide to completablefuture. www.nurkiewicz.com/2013/05/java-8-definitive-guide-to.html.

[9] Google. Google's guice dependency injection framework, . github.com/google/guice.

[10] Google. Google's guice dependency injection framework docs, . github.com/google/guice/wiki/Motivation.

[11] Wikipedia. Factory method pattern wiki post. en.wikipedia.org/wiki/Factory_method_pattern.

Listings

Appendix

You may use appendices to include relevant background information, such as calibration certificates, derivations of key equations or presentation of a particular data reduction method. You should not use the appendices to dump large amounts of additional results or data which are not properly discussed. If these results are really relevant, then they should appear in the main body of the report.

Appendix numbering

Appendices are numbered sequentially, A1, A2, A3... The sections, figures and tables within appendices are numbered in the same way as in the main text. For example, the first figure in Appendix A1 would be Figure A1.1. Equations continue the numbering from the main text.