Scaling a Web Application

Version 1.0

Rasmus Larsson

Abstract

A startup is in need of advice on how to scale their web application. This report takes a look at their current architecture and proposes a strategy and several tactics for achieving scalability.

Summary

A startup company has created a web application, which is currently in its alpha phase. The application targets a business area where there is little or no competition today. The company is worried that they may have to scale the application fairly quickly once the business takes off.

The existing architecture, consisting of Python/Django, Nginx, JavaScript and PostgreSQL, is detailed through the use of different architecture views and by segmenting the architecture into layers and tiers.

Based on both formal and informal sources an investigation of architecture qualities specifically scalability, performance, and availability is performed. Each quality is defined and tactics for achieving it are detailed. In some cases relationships between different qualities and how they converge or conflict are also mentioned.

Using the architecture qualities as a backdrop the current architecture is analyzed, above all from a scalability perspective, and it is found that the current architecture is a good foundation for achieving scalability.

The report proposes an overall strategy for moving forward: start measuring before making any changes. The report includes some basic details of how to measure the software. Based on the tactics for achieving scalability, examples of changes that the company may have to make in the future are also given.

Table of Contents

1 Introduction

You have an idea. You have the know-how. And you have the guts. Before you lies an open path to world domination, or at least success. How do you build it? And what happens if you do succeed, how will you scale?

These are some of the questions facing IT startups around the world today. This report takes a look at one of those startups and strives to help them along a bit so that they ultimately achieve world domination, or at least success.

1.1 Background

The company Acme AB is a newly formed Swedish startup company that has a specific business idea. They want to create an IT tool, a project tool, tailored to support a niche business area where there are no such tools today. The tool should be available through various devices such as smartphones, computers and tablets. It should simplify collaboration between different team members in a project.

The company currently has an alpha version of the tool in the form of a web application deployed to the cloud with some experimental users. The company is planning on moving into a beta stage in the first quarter of 2014.

1.2 Problem

The company founders have several years experience with web development and also experience from the business area they are targeting. They however lack experience from large-scale applications in the cloud. They perceive this as being their greatest technical weakness at the moment since a web application, due to its nature, may need to scale to hundreds of thousands of users, maybe even millions, in a very short time.

The problem therefore is simply put, how does one scale a web application? More specifically, based on their current technical platform, how do you scale their web application to handle possibly millions of users?

1.3 Purpose

The purpose of this report is to propose an architecture strategy for the company to use in their web application so that it can grow in accordance with future scalability requirements that may be placed on it.

1.4 Scope

This report will not take into consideration cloud providers and their various services. This includes hardware, network, and the underlying operating system. The company already has a cloud provider that provides them with excellent, personalized, service. The company is aware that they in the future may have to switch providers as they grow but this is currently not a concern to them.

The topic of Content Delivery/Distribution Networks (CDN) is also outside the scope of this report. The company has mentioned the possibility of using a CDN in the future but as with the cloud providers it is currently not a concern.

1.5 Target Audience

There are several target audiences for this report. The first is the company that wishes support and advice on how they should move forward with their architecture. The second is anyone interested in scaling web applications in general, or for the specific technical stack that the company uses. The third target audience is the teachers and students of the course Certified IT Architect, held by DF Kompetens AB, a subsidiary of the non-profit organization Dataföreningen i Sverige (the Computer Association of Sweden).

1.6 Method

A qualitative approach has been chosen. This will be based on interviews with the company to understand the problem domain and on studies of both formal (product documentation, books etc.) and informal literature (blogs, forums etc.). This may be further enriched with small code experiments.

The reason for disregarding a quantitative approach, which tends to be empirical by nature, is that the author does not feel that it would be practically feasible given the time frame and scope of the report. It is also the assumption of the author that existing architectures are tailor-made in various ways, which make them quantitatively incomparable.

1.7 Abbreviations and Terminology

The following table contains a list of abbreviations and terms and their definitions as used in this report.

Term/Abbreviation	Definition

Apache	Apache HTTP Server, an open-source web server.
CAP Theorem	A theorem that states that you can only guarantee two of the three qualities Availability, Consistency (as applied to data or state) and Partition tolerance in a distributed system.[1]
Cloud	Distributed computers providing services on-demand over a network. There are many different types of clouds and cloud providers.
Component	A discrete set of computing functionality. A component can be individually version controlled, deployed and executed. A component consists of one or more modules (see module).
High Cohesion	High cohesion is when responsibility is concentrated in a single or as few software modules as possible.
Latency	Latency is the amount of time that passes between the occurrence of an event and the point in time when the event actually becomes visible or has an impact. For example the time that passes between the broadcasting of a signal (event) and the reception of the signal (impact) is the latency. Latency is measured between two points.[2]
Layer	A layer is a collection of computing functionality (hardware or software) that has something in common, usually behavior or responsibility. Layers are ordered with respect to other layers based on level of abstraction. Examples of this are the OSI Model (ISO/IEC 7498-1:1994)[3] and the Internet Protocol Suite (TCP/IP, HTTP etc.)[4].
Loose Coupling	Loose coupling is when one software module has or makes as little use as possible of knowledge (or assumptions) about the definition of another module.
Module	A generic term for any discrete set of computing functionality regardless of size. Components, sub-components and sub-sub-

[1] Distributed Systems for Fun and Profit, chapter 2, Mikito Takada - http://book.mixu.net/distsys/single-page.html

[2] The Architecture of Open Source Applications, ZeroMQ, chapter 24.3, Martin Sústrik - http://aosabook.org/en/zeromq.html

[3] Wikipedia, The OSI Model - http://en.wikipedia.org/wiki/Osi_model

[4] Wikipedia, Internet Protocol Suite - http://en.wikipedia.org/wiki/Internet_protocol_suite

	components are all modules.
MySQL	An open-source RDBMS.
NAS	Network-attached storage, a file-level computer data storage available over a “normal” network. It provides both storage and a file system.
Nginx	An open-source web server.
PHP	A server-side scripting language targeted at web development and a general purpose programming language.
PostgreSQL	An open-source RDBMS.
Python	A high level multi-paradigm programming language.
RDBMS	Relational DataBase Management System
SaaS	Software as a Service, software provided on-demand, centrally hosted in a cloud.
SAN	Storage Area Network, a dedicated network that provides block-level storage. Unlike a NAS it does not provide a file system.
Shared-nothing architecture	A type of architecture based on a modular software stack where each module is distinct and independent from each other module and where each module is replaceable with another of the same type. An example of this is the LAMP stack (Linux, Apache, MySQL, PHP)[5] where each part of the stack is theoretically replaceable with another of the same type, e.g. Nginx instead of Apache, PostgreSQL instead of MySQL and Python instead of PHP. The shared nothing architecture was designed to run on commodity hardware and allow for easy expansion with more hardware (horizontal scaling)[6]. Another aspect of a shared-nothing architecture[7] is that each processing node is independent; there is no central state management that needs to be coordinated between nodes.
Sub-component	A discrete set of computing functionality. A sub-component

[5] Wikipedia, LAMP software bundle - http://en.wikipedia.org/wiki/LAMP_(software_bundle)
[6] The Django Book, Chapter 20: Deploying Django - http://py3k.cn/chapter20/en/
[7] The Architecture of Open Source Software, Scalabale Web Architecture and Distributed Systems, Kate Matsudaira - http://www.aosabook.org/en/distsys.html

	can usually be individually version controlled, but neither individually deployed nor executed. A typical example is a shared library.
Throughput	Throughput is the number of work items that can be processed during a single unit of time. Throughput is measured in a single point.[2]
Tier	A tier is a collection of computing functionality (hardware and/or software) that has something in common, usually behavior or responsibility. Tiers, unlike layers, are ordered with respect to other tiers based on responsibility and/or behavior, not abstraction. An example of tiers is the client and server tiers in a multi-tiered architecture[8]. A tier's relation to a layer is orthogonal, where a layer may be considered a vertical ordering (or a row) and a tier is a horizontal ordering (or a column).
Two-phase commit	Two-phase commit (2PC) is a protocol for distributed atomic transactions consisting of two phases, commit-request and commit. It is implemented using a transaction manager, e.g. an XA compliant manager (XA is a standard specified by the Open Group).
UML	Unified Modeling Language, a graphical modeling language.
WSGI	Web Server Gateway Interface, the interface between a Python application and web servers.

[8] Wikipedia, Multitier Architecture - http://en.wikipedia.org/wiki/Multitier_architecture, Linux Journal, Three-tier Architecture, Ariel Ortiz Ramirez - http://www.linuxjournal.com/article/3508

2 Current Architecture

This chapter will describe the current architecture as seen from various architecture views. It will lay the foundation for coming chapters that will discuss the current architecture and possible changes that may need to be performed in the future. Views chosen are those stipulated by Kruchten, also known as the 4+1 view model[9].

UML or simplified UML-esque notation will be used in the diagrams, the goal being communication, not necessarily UML completeness/correctness.

2.1 Use Case View

This sub chapter describes the business goals, processes and use cases that the system implements. Due to the scope and target audience of the report this chapter is brief.

The goal of the company is to provide a tool in a market segment where there currently are none. The aim is to be market lead within five years. The goal of the tool is to provide support in organization and collaboration within a specific type of projects from project inception to project delivery/end of life, with a concentration on support during the main execution step of the project type. Increasing ease of organization and collaboration is the main objective of the tool.

The market segment itself is worldwide with projects taking place on basically every continent. Being able to provide the tool to all of these locations is important to the company.

The project type is very dependent on geographic locations, one or more. Some tasks may be performed remotely but most tasks require physical co-location of the people performing them.

The project life cycle typically involves a few people during the inception phase, which then grows to a proportionally large number of people (usually specialists in one field or another) during the main execution step to then dwindle down again to a few people towards the end of a project. Project length (in calendar time) will vary depending on the size of the project. Everything from a few weeks to several years is possible.

[9] Architectural Blueprints—The “4+1” View Model of Software Architecture, Philippe Kruchten - http://www.cs.ubc.ca/~gregor/teaching/papers/4+1view-architecture.pdf

There may be related project types that follow the same basic life cycle but are repetitive, in other words project delivery does not lead to end of life but instead to project inception of the next repetition. The company is aware of this and may include support for this type of project in the future but it is not within the current scope of the tool today.

The project type is usually distributed geographically. It is therefore important that collaboration be possible on different devices ranging from laptops to tablets to smart phones. Connectivity may also be an issue in some cases so the tool should provide certain “offline” capabilities.

2.2 Logical View

This sub-chapter describes the information that exists and is used in the system, including requirements and constraints on that information and the relationships between different information entities, and the functionality that the system provides.

Currently there are no dependencies to other systems. There has been talk about integrating with an external payment provider but that is something that’s still on the drawing board.

2.2.1 Domain Entities

The below diagram (Figure 1: Domain Entity Model) illustrates a basic conceptual model of the various domain entities. The subsequent table (Table 1: Table of Domain Entities) provides further information concerning the individual entities and their relationships.

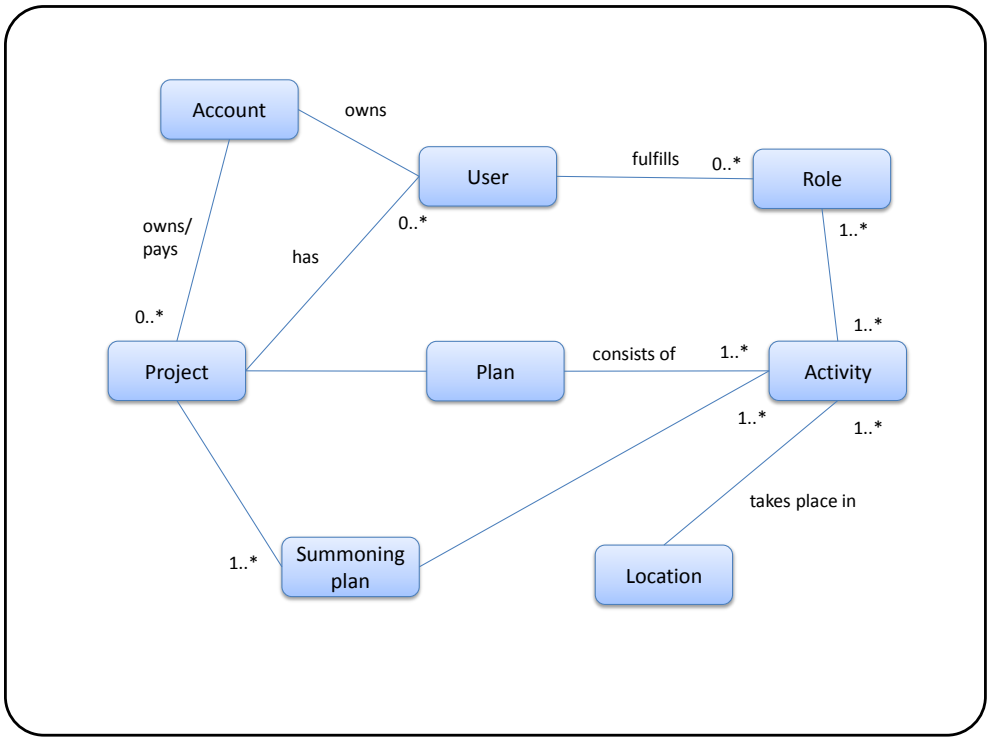

Figure 1: Domain Entity Model

Table 1: Table of Domain Entities

Entity	Description
Account	An account is the fiscal entity that owns and pays for one or more projects. An account is associated to a specific user that owns the account.
Project	A project with a goal to achieve something. A project has a plan and also has users associated with it. A project also has a summoning plan (see call-in plan).
User	A user is a user of the tool. A user can either be an account owner (and therefore owns and pays for projects) or someone associated with a project to fulfill a certain role (see role).
Plan	A plan consists of the activities that need to be performed to achieve the goal of the project. A plan details what it is that needs to be done as opposed to a summoning plan which details who it is that needs to

	do it.
Summoning plan	A summoning plan details which roles that are needed to perform a certain activity. The summoning plan includes information about when the roles needed should be summoned.
Role	A role is a specialist function fulfilled by a user needed to perform an activity. A user may fulfill several roles.
Activity	An activity is a set of tasks that need to be performed by specialists (roles) to complete the activity. An activity is performed in a location at a certain time for a period of time.
Location	A location is a place in the physical universe where activities are performed. A location may be used for more than one activity.

2.3 Development View

This sub-chapter describes the system as a set of software components and packages that relate to each other. It also describes the technologies used to implement the system.

The current system architecture is based on the standard out-of-the-box Django stack, in other words a database persistence layer with a Python/Django capable application server and a web browser client. The architecture is a client-server three-tier architecture. One could argue that the architecture is only a two-tier architecture, that the application tier and database tier are in fact the same. Such a discussion is however outside the scope of this report.

The current database server used is PostgreSQL. On the client side custom JavaScript is used to enrich the user experience.

The below diagram (Figure 2: Model of Tiers and Layers) illustrates the current architecture in the form of tiers and layers.

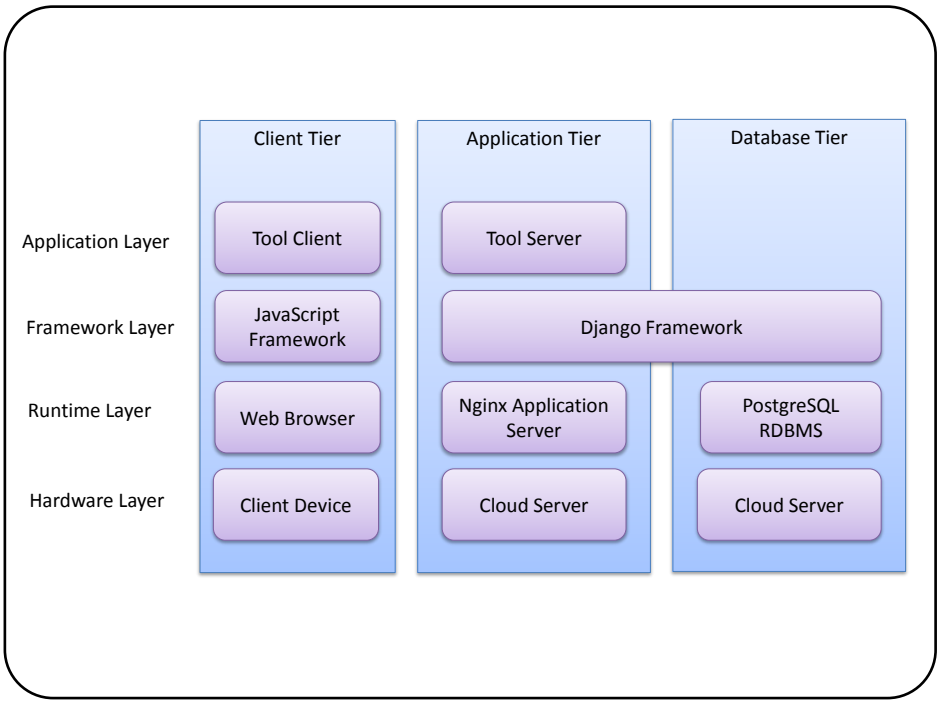

Figure 2: Model of Tiers and Layers

The tiers contain separate areas of responsibility. The client tier contains all the components necessary to run the client side of the application. The application tier contains the server side components, the bulk of the implementation with business logic and similar. The database tier contains components necessary for persisting data over time.

Each tier is composed of several layers of abstraction with the hardware at the bottom and increasing levels of software abstraction above that.

The below table (Table 2: Table of Architecture Artifacts) gives a short description of the various artifacts found in the architecture.

Table 2: Table of Architecture Artifacts

Artifact	Description
Tool Client	Application layer, client tier. The graphical client that the user uses to access the application.
JavaScript Framework	Framework layer, client tier. The underlying JavaScript framework of the client.
Web Browser	Runtime layer, client tier. The underlying execution environment of the client. This layer executes the JavaScript of the layers above it

	and mediates communication with the application tier through the layer below it.
Client Device	Hardware layer, client tier. The actual hardware device the user is using, such as a cell phone, laptop or desktop computer.
Tool Server	Application layer, application tier. The main implementation containing business logic and similar.
Django Framework	Framework layer, application and database tier. The underlying web framework of the application.
Nginx Application Server	Runtime layer, application tier. The execution environment of the application.
PostgreSQL RDBMS Server	Runtime layer, database tier. The execution environment of the database engine.
Cloud Server	Hardware layer, application and database tier. One or more servers in a cloud. These servers can be either dedicated physical servers or virtual servers running on top of physical servers.

2.3.1 About Django

Django is a "high level" web application framework built in Python. The basic architectural pattern of the framework is model-view-controller[10]. The basic artifacts of the framework are:

- Model - Design and implementation of an application's object model and logic. The model is relational and corresponds to the database tier artifacts, which are automatically created by tools included in the framework using the model as a specification. The model can also be enriched with rules governing the model's objects.
- View - Server side presentation module providing the server side interfaces to the model.

[10] MVC XEROX PARC 1978-1979,Trygve M.H. Reenskaug - http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html

- Template - Client side presentation module providing the client side user interface to the application.

The framework also includes ready to use functionality such as user management and authentication, an administrative interface and various tooling for automation of certain development tasks. Django aims for simplicity and rapid development but can be highly customized should one wish to do so. The framework uses a shared-nothing architecture[11] with distinct tiers between the application tier and the database tier.

2.3.2 About PostgreSQL

PostgreSQL is a multi-platform open source object-relational database system. It is fully ACID compliant, implements a large part of the SQL:2011 standard and fairly extensible. It features commonly found RDBMS functionality such as triggers, stored procedures, indexes and schemas among other things.

2.3.3 About Nginx

Nginx[12] (Engine X) is an open source web server, reverse proxy, load balancer and HTTP cache. It is lightweight and able to deliver high performance, especially for static content[13].

2.3.4 Future Additions

The company has considered adding a caching solution such as memcached[14] but this is still in the planning stages.

2.4 Process View

This sub-chapter describes the runtime aspects of the system such as threading, concurrency and the communication between them.

[11] Django Documentation, FAQ, Does Django scale? - https://docs.djangoproject.com/en/dev/faq/general/#does-django-scale
[12] Nginx.org - http://nginx.org/en/
[13] Linode library, Basic Nginx Configuration - https://library.linode.com/web-servers/nginx/configuration/basic
[14] Memcached.org - http://memcached.org/

2.4.1 Django

Django is multi-threaded and has a thread safe core[15] that can be deployed in a threaded container (a container is the interface between the application and the web server it's deployed on, see WSGI). That being said this does not guarantee that applications written on top of Django are thread safe. It is however outside the scope of this report to look into the details of threading in the company's application.

Due to its shared-nothing architecture, Django is stateless; all state is kept in the database.

2.4.2 Nginx

Nginx uses a non-blocking asynchronous event driven architecture internally. Connections are handled by a number of single-threaded "workers". The workers share a common inbound socket where they take turns in accepting new requests. The worker process is highly optimized and each worker is able to handle thousands of connections.

The workers, and Nginx, do not perform forking of processes or threads. Instead workers are initialized during startup. This leads to efficient memory and CPU usage, and scales easily to multiple cores.[16]

2.4.3 PostgreSQL

PostgreSQL uses a client-server model where a master process, postgres, spawns a new server process for each client that connects. Server tasks communicate internally using shared memory and semaphores to coordinate the work.[17]

2.5 Physical View

The physical view is outside the scope of this report.

---

[15] Quora.com, Does Django have any thread safety issues? - http://www.quora.com/Django-web-framework/Does-Django-have-any-thread-safety-issues

[16] The Architecture of Open Source Applications, chapter 14, Nginx, Andrew Alexeev - http://www.aosabook.org/en/nginx.html

[17] PostgreSQL 9.3.1 Documentation, Chapter 46.2 How Connections are Established - http://www.postgresql.org/docs/9.3/static/connect-estab.html

3 Architecture Qualities and Optimization

This chapter will look at a few application architecture qualities and their tactics. The described qualities have been chosen based on interviews with the company and the concerns they have expressed, with a concentration on scalability. This chapter will also look at optimization. Optimization is not an architecture quality but rather work to achieve or improve one or more qualities.

3.1 Scalability

Scalability is the ability of a system to continue providing its function whilst growing the amount or size of work, in reverse proportion to the amount or cost of change needed to grow. There are several aspects of scalability:

- Size scalability – The ability of the system to grow the amount/size of work through the use of more hardware, and the ability of the system to grow its dataset without affecting latency.
- Geographic scalability – The ability of the system to expand to geographically distributed datacenters.
- Administrative scalability – The ability of the system to scale without incurring additional administration.

Size scalability is usually split into two dimensions, horizontal and vertical scalability. Vertical scalability is increasing the capacity of a single node e.g. by adding more memory. Horizontal scalability is increasing capacity by adding more nodes.

Scalability is not to be equated with performance or availability. They are related but can be both synergistic and contradictory.

eBay has a pragmatic list of seven best practice tactics for scalability[18]. These will be explained in the below table (Table 3: Scalability Tactics, part 1).

[18] InfoQ, Ebay Scalability Best Practices, Randy Shoup - http://www.infoq.com/articles/ebay-scalability-best-practices

Table 3: Scalability Tactics, part 1

Tactic	Description
Partition by function	The more modularity-by-function you have (essentially loose coupling and high cohesion), the easier it is to scale one module independently from other modules. This type of modularization is possible in several different places, both layer and tier. One could e.g. create a new tier for a subset of services.
Split horizontally	In stateless services it should be trivial to add additional instances side by side with the existing instances since specific processing on any given instance will return the same result regardless of instance. This allows for scaling capacity up or down by simply adding or removing instances. For stateful services it becomes more difficult and one solution for this is to split state (shard) across several instances and address a specific instance according to the state one is interested in.
Avoid distributed transactions	Distributed transactions are a way to coordinate distributed resources. Coordination of resources affects scaling, availability and latency negatively and increases as the number of resources involved increases. By strategically relaxing, on a use case basis, the requirements for Consistency to the advantage of Availability and Partition tolerance (see CAP theorem) through disallowing distributed transactions one can avoid the cost of coordination.
Decouple functions asynchronously	Synchronous calls between components increases the coupling between the components. If a component is unavailable all other components dependent on that component are essentially also unavailable. Not so in asynchronous events that are propagated within and between components.
Move processing to asynchronous flows	By making not only component interaction asynchronous but also use case processing asynchronous allows the system to dynamically swallow load in queuing mechanisms. This is essentially distributing work over time, delaying processing until resources become available. This further increases the possibility of individually scaling services and scaling them to average load, not peak load.
Virtualize at all levels	The addition of an abstraction layer allows for controlling, changing and scaling the layer below it without affecting

	components in layers above it. This effectively adds manageability to scaling. Examples of this are virtual IPs that may reference several different nodes or composite services that wrap other services for the clients.
Cache appropriately	Caching can lead to increased scaling at very little cost and slow changing or static data are typically likely candidates. Taken to its extreme caching can however become a dangerous path were the cache steals resources from your services or even becomes a dependency to the services to the degree that it affects availability; the system cannot run if the cache isn't available. Each situation must be analyzed, and re-analyzed after a time, to see where caching will help in an appropriate way.

Rozanski and Woods specify a list of thirteen different tactics for scalability[19]. There is overlap between these tactics and those of eBay. Only the tactics that aren't completely covered by eBay's will be listed (Table 4: Scalability Tactics, part 2) and commonalities will be annotated with EBBP (eBay's Best Practices).

Table 4: Scalability Tactics, part 2

Tactic	Description
Optimize Repeated Processing	One way of doing this could be through caching (EBBP). Another could be algorithmic optimization for speed or some other characteristic.
Reduce Contention via Replication	Contention could be for either a stateless or stateful service. In the case of a stateless service splitting horizontally (EBBP) will most likely reduce contention. For a stateful service replication from a write master to read slaves could be one way of doing this.
Prioritize Processing	Some processing is more important than others. For example, being able to book a ticket is more important than seeing the seat layout. Having the ability to prioritize allows you to scale up one service at the expense of another service. This also correlates to availability of one service at the expense of another.
Partition and Parallelize	Partitioning functionality (EBBP) may be combined with parallelizing work either across several different services or

[19] Software Systems Architecture, Nick Rozanski and Eoin Woods - http://www.viewpoints-and-perspectives.info/home/perspectives/performance-and-scalability/

	across several different instances of the same service.
Scale Up or Scale Out	One can increase resources in one of two ways, horizontally or vertically. Vertical scaling is increasing the amount/size of work a single node can do whereas horizontal scaling is an increase in the amount of nodes doing work.
Degrade Gracefully	If a service becomes unavailable it's important that this doesn't affect other services (loose coupling) and that a backup instance takes its place (abstraction layer). This is essentially an availability tactic.
Make Design Compromises	Just as the three qualities of the CAP theorem contend with each other, other qualities will be in contention. In some cases compromises will have to be made for the greater good.

3.2 Performance

Performance is the amount of time spent on doing useful work compared to the total time spent and the resources used. Performance can be measured in several different ways, for example:

- Response time (latency)
- Throughput
- Amount of resources used in relation to the work being performed (e.g. CPU efficiency)

There are two main tactics to achieve performance, which Martin Fowler very elegantly summarized as "speed up the slow things or do the slow things less often."[20]

The below table (Table 5: Performance Tactics) describes these two tactics a bit more in detail. Each tactic has a number of sub-tactics, which may be mentioned but will not be expanded on.[21]

[20] Yet Another Optimization Article, Martin Fowler - http://martinfowler.com/ieeeSoftware/yetOptimization.pdf

[21] Software Architecture in Practice (ISBN-10: 0-321-81573-4), Len Bass, Paul Clementz and Rick Kazman

Table 5: Performance Tactics

Tactic	Description
Control Resource Demand	This tactic is aimed at reducing demand on services and other resources. It includes amongst its sub-tactics prioritization of processing which goes hand in hand with scalability.
Manage Resources	This tactic is aimed at controlling the services and resources themselves. Amongst its sub-tactics are increase resources, concurrency, multiple copies of data and computation. The last two also go hand in hand with scalability.

It is worth noting that performance is in contention with other qualities such as modifiability; one modifiability tactic is to use abstraction layers. Using abstraction layers is also a scalability tactic; as such it may be in direct conflict with performance, depending on the situation.

3.3 Availability

Availability is the amount of time a system is functioning and accessible compared to total calendar time. Availability is usually measured as a percentage of calendar time. Availability often has a direct relationship with fault tolerance.

There are three main tactics (Table 6: Availability Tactics) one can use to achieve availability.[21]

Table 6: Availability Tactics

Tactic	Description
Detect faults	This includes amongst others monitoring for certain conditions, using heartbeats to ensure processes are up and running, and self-tests of various sorts.
Recover from faults	This tactic includes both proactive and reactionary sub-tactics. Proactive sub-tactics are e.g. redundancy, exception handling and rollbacks. Reactionary sub-tactics include amongst others resynchronization after failure/restart.
Prevent faults	This tactic includes sub-tactics such as predicting faults (e.g. monitoring of disk usage), increasing the number of faults a component can handle, using transactions (two-phase commit) to avoid race conditions and removing/restarting services before they fail.

Availability tactics have a convergence with scalability tactics in for example redundancy. However, there may also be a conflict when it comes to transactions, depending on the situation.

3.4 Optimization

Optimization (Latin: optimus, meaning “best”) is usually, within the field of computer science, any work aimed at improving performance. Optimization may however affect other architecture qualities as well since performance has a relationship with other qualities such as scalability and is therefore of interest in this report. Also, using the original Latin definition, optimize need not necessarily mean “fastest”. One could for example optimize for scalability. The best scalability need not necessarily be the fastest but it might be fast enough.

With today's hardware performance it is usually not the most prioritized quality in an application architecture (there are of course exceptions). Rather other qualities, which may be in direct conflict with performance, will take precedence and functionality will trump them all. If one tries to optimize from the beginning one is prioritizing performance over other architecture qualities and possibly even functionality, thus degrading the application that's being built.

Therefore the first rule of optimization could well be: Optimize Later[22]

In software (and hardware) there are so many “moving” parts, systems and their relationships are so complex, that it becomes extremely difficult to predict in an accurate fashion where bottlenecks will crop up. It is only when you actually measure that you can see where they truly are.

Therefore the second rule of optimization could well be: Measure First[20]

[22] Portland Pattern Repository Wiki - http://c2.com/cgi/wiki?OptimizeLater

4 Analysis of the Current Architecture

4.1 Client Tier

Each new user will be using his or her own device, which means that the client to user ratio will be constant at 1:1. Scalability on the client side should therefore not be an issue regardless of the total number of users. Performance of the client should not be affected either, provided the initial implementation achieves the minimum performance requirements.

The client already has certain offline capabilities, where the client downloads all necessary information in an initial load and then only downloads/uploads changes. This helps fulfill the availability requirements. It is important that the architecture as a whole take this offline capability into consideration however; if clients have write access to local data when offline there is a risk of two different clients making changes to the same data while offline and then one overwriting the other's changes when coming online.

4.2 Application Tier

With its stateless architecture Django is quite capable of scaling a fair bit. This is evidenced by companies such as Instagram[23] and Disqus[24] using Django in their stack. Disqus for example was handling 500 million unique visitors per month in 2011, with a peak of 25k requests per second. It should be noted however that Disqus makes modifications to Django in order to make it scale to their needs. This was in 2011, since then Django has gone from v1.2 to v1.6 (just released) so it's hard to say how applicable the need for these modifications are to the current version of Django. Regardless, Django evidently does scale.

[23] Instagram Engineering Blog, What Powers Instagram: Hundreds of Instances, Dozens of Technologies - http://instagram-engineering.tumblr.com/post/13649370142/what-powers-instagram-hundreds-of-instances-dozens-of

[24] PyCon 2011, Disqus Serving 400 Million People with Python - http://blip.tv/pycon-us-videos-2009-2010-2011/pycon-2011-disqus-serving-400-million-people-with-python-4898303

NGINX is a really good fit for the future. Its processing model allows it to scale easily, horizontally, with increased demand. Several large sites such as FaceBook, Netflix and WordPress.com use NGINX[25] today, handling up to 70000 requests per second[26].

4.3 Database Tier

Django supports using multiple databases[27] at the same time with the ability to specify not only different database types (e.g. PostgreSQL and MySQL) but also route, on a model level, which database node should be used for read operations, or write operations, for a specific Django model. This allows a great deal of freedom in configuring Django, e.g. writing to a master node but reading from any of a set of slave nodes. There is however a limitation, Django does not support foreign key or many-to-many relationships to span across multiple databases.

PostgreSQL supports a number of different configurations for high availability, load balancing and replication. These include solutions on the file system level such as a shared disk (NAS/SAN solutions) and file system replication, and solutions on the database server level such as hot standby and master-standby replication in either synchronous or asynchronous mode.[28]

PostgreSQL has a number of big companies among its user base such as Disqus, Skype[29] and Braintree[30]. PostgreSQL therefore seems to be a good fit for applications that need to scale.

[25] Who Uses NGINX? - http://nginx.com/company/
[26] WordPress.Com Serves 70,000 Req/Sec And Over 15 Gbit/Sec Of Traffic Using NGINX - http://highscalability.com/blog/2012/9/26/wordpresscom-serves-70000-reqsec-and-over-15-gbitsec-of-traf.html
[27] Django Documentation, Multiple Databases - https://docs.djangoproject.com/en/1.5/topics/db/multi-db/
[28] PostgreSQL 9.3.1 documentation, Chapter 25 - http://www.postgresql.org/docs/9.3/static/high-availability.html
[29] PostgreSQL Featured Users - http://www.postgresql.org/about/users/
[30] Scaling PostgreSQL at Braintree: Four Years of Evolution - https://www.braintreepayments.com/braintrust/scaling-postgresql-at-braintree-four-years-of-evolution

5 Strategy for the Future

Thus far it has been shown that the current architecture and software stack have the potential for high scalability. There are several large solutions deployed today based on more or less the same stack that handle thousands of requests per second.

The strategy for the future will be based on the various tactics described in the chapter concerning Scalability, primarily on eBay's tactics for scalability since eBay is more or less within the same domain: web applications. The other tactics will be kept for future reference and the interest of the reader but not further detailed here unless otherwise noted.

5.1 The First Thing to Do is Nothing?

I think the biggest mistake one could make concerning any architecture faced with a need to scale would be to assume that it currently cannot. A common sense proverb comes to mind: don't fix what isn't broken. Therefore the logical first step in making the architecture more scalable is to do nothing at all, at least not to the scalability of it. This goes hand in hand with the first rule of optimization: optimize later. If one were to concentrate on making the architecture more scalable other qualities and possibly even functionality would take a back-seat row, which in the long term could become fatal for the architecture's viability.

There is however one thing that can and should be done, introduce measurability. Based on the second rule of optimization one should first measure whether in fact there is a need to scale. Introducing measurability at an early stage could indeed save a lot of work further down the line, maybe throughout the entire life of the architecture.

One word of caution, introducing any new quality into an existing architecture may to some degree affect the other pre-existing qualities, e.g. performance. However, it is my opinion that the risks of not measuring at all outweigh the risks of introducing measurability.

Delving into the various types of performance testing and how to perform them is not the objective of this report. Microsoft has published a whole book on the matter, "Performance Testing Guidance for Web Applications", available freely from Codeplex (http://perftestingguide.codeplex.com/) that could be a good place to start.

The rest of this chapter will instead take a look at some of the practical sides of how to measure in various parts of the architecture. Hopefully these will work as starting points to more in depth measuring of the architecture without going too deep into the technical or configuration side of the various products used.

5.1.1 Measuring Nginx

Nginx has an HTTP logging module that allows logging, amongst other things, the response time[31] and the requested resource[32]. Modifying the current logging to include that information is a relatively fast and cheap way to quickly gain information about the current responsiveness of the system. Since it can include information about the requested resource it should give insight into not only Nginx but also the whole application tier and also the database tier. It cannot however show what in the application tier or database tier it is that's taking time but it should suffice to indicate when latency is increasing and throughput is decreasing, and the resources affected.

Exporting the log statistics on a regular basis, visualizing them in some sort of graph and keeping a historical archive of averages and peaks should be a good start to keeping track how load evolves over time.

5.1.2 Measuring Django

There are a number of tools available for Django to measure performance. The developers at Disqus have open-sourced a performance tool (https://github.com/disqus/django-perftools) that will log request execution times. Logging can be configured to log only when execution times exceed a certain threshold.

At Github there is also a Django Debug Toolbar (https://github.com/django-debug-toolbar/django-debug-toolbar/) that will give insight into execution times and how long each part of the execution is taking, including insight into the SQL queries themselves.

[31] Tracking Application Response Time with Nginx, Vlád'a Macek - http://lincolnloop.com/blog/tracking-application-response-time-nginx/
[32] Nginx HTTP log module - http://nginx.org/en/docs/http/ngx_http_log_module.html

5.1.3 Measuring PostgreSQL

PostgreSQL is extensively documented and not surprisingly this includes information about performance tuning and logging. The documentation points to at least two different logging tools useful for visualizing log data, pgFouine and pgbadger. The logging parameters in PostgreSQL include "log_min_duration" which can be used to find slow statements[33].

5.1.4 Measuring Everything

Another option when it comes to measuring is using a third party provider that supports the current hardware and software stack, e.g. a SaaS provider such as New Relic (http://www.newrelic.com/).

5.2 Partition by Function

If it turns out that the architecture can no longer scale with the requirements the architecture must be changed. The first tactic of scalability is therefore to start partitioning the architecture by function. In the end, the resulting measurements from previous work should dictate which actions to take. This and the following sub-chapters are therefore purely theoretical and should be seen as recommendations only.

The application tier contains a number of different entities, all part of the conceptual model, that interact in different ways. In a standard Django application these are tightly coupled to each other through the Django models (more on this in the next sub-chapter). As load increases it's not certain that load will increase linearly across all entities. Looking at the domain entities from chapter 2, the User entity will most likely see a much higher load as users log in and out than the Account entity, which may be accessed only occasionally to initiate new projects, pay bills etc. Decoupling these two entities from each other, at the technical level (the logical level must of course remain intact), would allow more freedom in modifying the User entity to handle load.

The end result of decoupling entities to such a degree that they can evolve independently are services that communicate using predefined interfaces. It is possible that each major domain entity in the end must be encapsulated in one or several services for the architecture to be able to scale.

[33] PostgreSQL Wiki - http://wiki.postgresql.org/wiki/Tuning_Your_PostgreSQL_Server

The above paragraphs concentrate on domain entities as logical boundaries for functionality. This is not necessarily the case. In some cases other types of boundaries such as behavior, e.g. reading as opposed to writing, may be the more efficient boundary. An example of this is with streaming services such as Youtube where uploading and encoding video (computation intensive) is probably handled by a completely different service than watching video (performance sensitive).

Looking at the database tier, Django forms joins between different models at the database level. This creates a tight coupling between different tables that may put restrictions on the architecture. To achieve functional partitioning these joins may need to be broken. One way of doing this is to move the joins from the database layer up through the abstraction layers within the database tier into the Django framework, or even horizontally into the application tier. This goes hand in hand with creating services for the domain entities and would allow different services to use different databases removing resource contention as an issue. Note that this type of change will affect performance negatively but is most likely worth it to gain scalability.[34]

Another type of partitioning available is what's commonly called vertical partitioning within the DB community. It is the voluntary splitting of a table into several tables with e.g. less frequently used columns moved out from the primary table to the secondary table thus decreasing the amount of information needed to be stored in memory, or index, for the primary table. This is something that PostgreSQL supports, but should only be used once a table grows to a size where it no longer fits in memory[35].

5.3 Split Horizontally

Splitting horizontally is basically the distribution of processing to several nodes. This implies concurrency by letting several different nodes do work at the same time. The work can be either the same type of work or different types of work.

The application tier of the architecture is stateless. Scaling the application tier is therefore simply a question of deploying more nodes and putting a load balancer in front of them. Since the architecture already includes Nginx, which implies a certain familiarity with the product, and Nginx can also act as a load balancer, reconfiguring Nginx (possibly a separate instance) should be a simple first step.

[34] Disqus: Scaling the World's Largest Django Application - http://ontwik.com/python/disqus-scaling-the-world%E2%80%99s-largest-django-application/
[35] PostgreSQL 9.3.1 Documentation, chapter 5.9 Partitioning - http://www.postgresql.org/docs/current/interactive/ddl-partitioning.html

Unlike the application tier the database tier does contain state. This means that one cannot simply add database nodes without making other changes.

As mentioned previously, partitioning functionality will allow one to run different services on different database nodes, but what happens when the data of a service grows too large for a single database node? The answer is horizontal partitioning also known as sharding. Table rows are grouped according to some logical key e.g. creation date, a segment of the alphabet or similar, and each group is placed on a different node, or shard. This distributes load across several different shards as requests target different groups.

Sharding solves the problem of distributing data across several nodes, but how do you scale a read intensive service, or a service where read and writes are creating contention? One possible solution is to set up a write master and one or more read slaves. All writes are directed to the master and then the data is replicated to the slaves. This has to be done with care since this type of setup will break consistency (CAP theorem) between the master and slaves. Not all data can afford to be inconsistent (economic transactions) whereas it is acceptable for other types of data (social network updates). As mentioned previously, PostegreSQL supports streaming replication, both synchronously (consistent) and asynchronously (inconsistent). One possible set up is using synchronous replication between collocated nodes and asynchronous replication between different data centers. This provides a high level of load balancing (and fault tolerance) providing consistency as long as a client continues addressing the same datacenter, without incurring latency costs on the request for replicating data to other datacenters.

5.4 Avoid Distributed Transactions

Going forward the architecture must try to avoid distributed transactions (two-phase commit) wherever and whenever possible. One must look at the requirements for each use case carefully and decide whether a distributed transaction really is necessary. In most cases it probably isn't.

A distributed transaction, should it be deemed necessary, can possibly be replaced by something else. For example a transaction that needs to cross two different services could be handled by a third service that wraps the other two (a so called composite service) that includes compensation logic in case a call to one of the two fails.

5.5 Decouple Functions Asynchronously

Any communication where the response is not needed, or not needed right away, makes a good candidate for converting to an asynchronous mode. This could be between different modules in the same layer, between different layers or between different tiers. Prime candidates are for example services that depend on other services and the communication between them is adding unnecessary latency.

Django has a built-in event dispatching system called signals, which seems to fit perfectly, if it weren't synchronous. One alternative is to use the python threading API to kick off a separate thread that will execute the action on its own. Introducing threading may include thread-safety issues, which currently don't exist. Another alternative is to implement a queuing mechanism of some sort, e.g. Celery. Celery is "an asynchronous task queue/job queue based on distributed message passing."[36] It is built in Python and supports Django out of the box[37]. The drawback of Celery is however that it requires the implementation of a message broker as well.

In the database tier, PostgreSQL can be configured to perform certain operations asynchronously. As already mentioned PostgreSQL supports asynchronous replication, which can be used to achieve geographic scalability without increased latency. PostgreSQL also supports asynchronous commits[38]. Asynchronous commits can however also cause data loss if the database node fails before the commit has been written to disk. Asynchronous commits should therefore only be used for non-vital data.

5.6 Move Processing to Asynchronous Flows

Having decoupled functions allows for the next step, the conversion of processing to asynchronous flows. Many of the use cases in the architecture may be possible candidates for asynchronous processing. Since the use cases of the architecture have not been detailed in this report two fictive use cases (Table 7: Example Use Cases) will be used as examples.

[36] Celery Project - http://www.celeryproject.org/
[37] Using Celery With Django - http://docs.celeryproject.org/en/latest/django/first-steps-with-django.html#using-celery-with-django
[38] PostgreSQL Documentation, chapter 29.3 - http://www.postgresql.org/docs/9.3/static/wal-async-commit.html

Table 7: Example Use Cases

Use Case	Description
Account creation with verification e-mail	The user creates an account in a two-step process where the user must first register on a web page and then click a link in an e-mail sent to her to verify that the e-mail address of the account is correct. When registering on the web page there is no reason for the user to wait for the system to actually send the e-mail before returning a result, the sending of the e-mail and generation of a one-time verification URL can be handled asynchronously with respect to the registration form.
Upload of media	The user uploads media of some type that needs to be processed before being made available in the system. The upload view returns a successful result as soon as the media is uploaded, whereas processing is handled asynchronously in the background. Should the user want to view the media before processing is complete a temporary status such as "Waiting to be processed" could be returned.

5.7 Virtualize At All Levels

By abstraction (through virtualization) the architecture gains improved availability and administrative manageability (which translates to administrative scalability). An example from the network level is a virtual IP that may be redirected to a new host should the old one fail or need to be replaced.

The Django framework already has a data access layer that abstracts the database to a certain degree. Moving forward this layer may have to be customized to hide complexities in the database layer such as master/slave set ups, shards etc.

Read or search intensive services may have to be abstracted using a composite service to parallelize searches across several instances and then aggregate the results before sending them back to the user.

Load balancers typically disregard non-responding nodes. This is also a form of virtualization.

All of the above allow increased availability since failing nodes will be hidden from view, but it also increases manageability since it allows administrators to add new nodes, move nodes and remove old nodes with greater freedom.

5.8 Cache Appropriately

Typical candidates for caching are static data sets or static media such as images, videos and static web pages. Dynamic content is more difficult to cache and questions such as when to refresh the cache become important.

For the specific architecture two possible candidates for caching may be:

- Activities – Once they have started it is unlikely that they will change, but will need to be accessed by many people while they are ongoing.
- Plans – After the first planning stage, plans will most likely not change too much and could therefore be cached. Should they change, there must however exist functionality to invalidate the cache when that happens.

5.9 Optimize Repeated Processing

Due to recent improvements in the Django framework this tactic, all though not a part of eBay's best practices, is being included here.

Prior to version 1.6 of the Django framework database connections were not pooled. Each new request resulted in the negotiation of a new connection instantly incurring an overhead cost. In version 1.6 (released 2013-11-06) this has been changed through the implementation of persistent connections[39]. A simple improvement is therefore to upgrade to version 1.6 and activate the use of persistent connections.

5.10 A Graphical Roadmap for the Future

The Django Book (http://www.djangobook.com/en/2.0/chapter12.html) has a nice roadmap, including illustrations, for how to move from a single server application to a setup with 5 independent clusters: load balancing, caching, media, application and database. Even though it uses a slightly different software stack it is recommended as a visual guideline on the first steps to scaling a Django application.

[39] Django Documentation v1.6, Databases - https://docs.djangoproject.com/en/dev/ref/databases/#persistent-database-connections

6 Conclusions

The purpose of this report was to propose a strategy for developing the architecture.

Referenced work describing existing architectures have shown that scaling web applications with the same or similar software stack is possible. As such the application has the potential for the same high scalability and in fact the company has made a good choice of platform from the beginning. The path ahead lies in fine-tuning it to their specific needs.

A strategy has been proposed consisting of eight tactics for scalability, each described with one or a few mostly technology agnostic examples of how the tactics could be implemented.

The strategy also proposes the introduction of a new capability into the architecture, measurability, and that this should be the first step when moving forward on the path to scalability. Measurability will provide hard metrics that will help in discerning when and where to introduce changes that will improve scalability.

7 Discussion

Approaching the area of web applications that I previously had little knowledge of has been an interesting and exciting endeavor. The openness of open source software aided me a lot in finding out details about the internal workings of software such as PostgreSQL and Nginx. Likewise the willing-to-share attitude in general found in blogs, webcasts etc. was of invaluable help to me. The Internet has my thanks!

At the same time I feel that sometimes the software documentation of open source is more geared at how to use the software and less on what the architecture of the software is. I felt this to be especially true of Django (perhaps my searching skills just aren't up to par). This tendency makes it more difficult to objectively analyze and choose software that will fit specific architecture requirements.

There is no static; the state of software is in continuous change. This was especially noticeable in both Django and PostgreSQL announcing new releases during the writing of this report. The documentation of software lives its own life and need not necessarily be updated at the same pace as the software. Blogs, webcasts, articles and even this report are snapshots in time. Finding up to date information can therefore be difficult and some of the references used in this report may very well already be obsolete. My hope is that this will be due to software improvement, not decline.

8 Appendix: List of Tables and Figures

Hittar inga figurförteckningsposter.