BAB II
LANDASAN TEORI

II.1 Sekilas Crimezone

Crimezone merupakan aplikasi *citizen journalism* pada *mobile* untuk melaporkan bentuk kriminalitas. Bentuk kriminalitas yang bisa dilaporkan berupa perampokan, penganiayaan, penculikan, pemerkosaan, pembunuhan, dan pembakaran. Aplikasi ini tersedia dalam platform *mobile android* dan *windows phone*. Secara umum, aplikasi ini memudahkan pengguna dalam memperoleh informasi mengenai area lokasi yang rawan kriminalitas, sehingga dengan informasi tersebut masyarakat dapat menghindarinya. Selain itu, pengguna juga dapat mem-*posting* tindakan kriminal dengan cara mengambil foto suatu kejadian beserta informasi tambahan seperti nama lokasi, jenis kejahatan, tanggal kejadian, dan lain sebagainya.

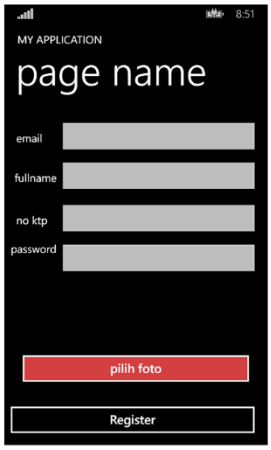
Crimezone terdiri dari dua subsistem yaitu subsistem *mobile* yang digunakan oleh masyarakat sebagai media untuk melaporkan kejahatan dan mengetahui informasi tindak kejahatan yang berasal dari pihak kepolisian atau masyarakat itu sendiri, serta subsistem *web* yang digunakan oleh pihak kepolisian untuk mengolah data kejahatan. Berikut gambaran implementasi sistem Crimezone:

1. Pengguna akan memberikan informasi posisi keberadaannya melalui GPS, kemudian GPS akan memberikan titik koordinat keberadaannya. Koordinat yang didapat akan berpengaruh terhadap map yang akan ditampilkan di *smartphone* pengguna.
2. Pengguna yang menggunakan sistem berbasis *mobile* dapat melakukan *posting* data dengan mengirimkan *post* data ke *web service*, kemudian *web service* tersebut meneruskan pengiriman data ke *database*. Selain itu, pengguna juga bisa melakukan *binding* data yang dilakukan melalui *web service*, kemudian *web service* akan melakukan *request* ke dalam *database*.
3. Untuk sistem yang berbasis *web* dapat melakukan *posting* data dengan mengirimkan *post* data ke *web service*, kemudian dilakukan pengiriman data ke basis data yang mana akan berlaku sama dengan pengiriman data yang dilakukan pada sistem berbasis *mobile*.

Berikut adalah *screen-shoot* dari aplikasi Crimezone.



Get Stared



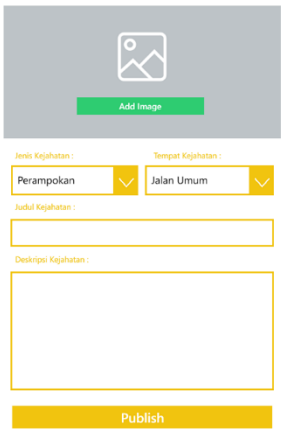
Register



Login



Home



Kirim Berita



Logout

II.2 Software Maintenance

Di dunia IT, sangat sedikit proyek pengembangan perangkat lunak yang dirancang dari awal. Biasanya, *customer* sudah memiliki sistem IT yang sudah di-*deploy* dan berjalan. Sistem tersebut dinamakan *legacy system*. Sistem tersebut memiliki nilai yang besar untuk pelanggan karena terbukti sudah berjalan dalam waktu yang lama. Di lain sisi, untuk menjaga sistem tersebut sangat mahal, sehingga dibutuhkan metode untuk mengurangi biaya pemeliharaan yang diperlukan [7].

Berikut ada beberapa pendekatan tentang bagaimana menangani *legacy system* yang termasuk ke dalam satu genus yaitu *software maintenance*. *Software maintenance* merupakan aktivitas yang dibutuhkan untuk menjaga sistem perangkat lunak operasional dan responsif setelah diterima dan diproduksi. Ini adalah serangkaian kegiatan yang mengakibatkan perubahan pada *accepted – baseline – product* yang aslinya. Perubahan ini terdiri dari modifikasi yang dibuat dengan mengoreksi, memasukan, menghapus, memperluas, dan meningkatkan sistem dasar [8]. Berikut ini yang termasuk dalam metode *software maintenance* [7]:

a. *Software Re-Development* yaitu mengembangkan sistem baru dengan fungsionalitas baru dengan menggunakan ide-ide dari *legacy system*.
b. *Software Evolution* yaitu meningkatkan *legacy system* tetapi fungsionalitasnya tidak terlalu banyak yang berubah.
c. *Software Re-Engineering* yaitu merenovasi sistem dan meningkatkan kemampuan *maintenance* tanpa mengganti teknologi sistem.
d. *Software Migration* yaitu memindahkan sistem ke dalam lingkungan baru atau teknologi baru tanpa mengganti fungsionalitas.

II.3 Software Migration

Seperti yang telah dijelaskan bahwa *software migration* merupakan metode dari *software maintenance* dengan definisi memindahkan sistem ke dalam lingkungan baru atau teknologi baru tanpa mengganti fungsionalitas yang ada pada sistem. Metode ini merupakan salah satu yang paling cocok untuk mengurangi biaya pemeliharaan karena meningkatkan *maintainability* dengan *memigrasikan* perangkat lunak ke dalam teknologi baru. Berikut ini yang termasuk dalam proyek *software migration* meliputi [9]:

a. Migration of data
b. Migration of *user interface*
c. Migration of *code*

Memigrasikan *legacy system* membutuhkan pemahaman dari *legacy code* dan mengubahnya menjadi teknologi baru. Dengan melihat aplikasi perusahaan besar yang terus berkembang biak dalam waktu yang lama, tugas ini menjadi tugas yang kompleks. Oleh karena itu, diperlukannya metode struktur dalam merencanakan dan melaksanakan migrasi.

Ackerman telah menganalisis beberapa metode *software migration* dan mengidentifikasi sekumpulan disiplin inti dari *software migration*. Hasilnya yaitu berupa pendekatan umum tentang cara *memigrasikan* perangkat lunak yang dinamai referensi migrasi proses (ReMiP). ReMiP dibagi menjadi yaitu *supporting disciplines* dan *migration disciplines*. *Supporting disciplines* menangani kegiatan pendukung seperti manajemen proyek, manajemen konfigurasi atau lingkungan pengembangan. Sedangkan *migration disciplines* menangani perencanaan, pelaksanaan, dan memvalidasi migrasi itu sendiri. *Migration disciplines* terdiri dari [5]:

1. *Requirement Analysis*

Tahapan pertama dalam ReMiP yaitu *requirement analysis*. Dalam *memigrasikan* perangkat lunak, fungsionalitas tidak di ubah sehingga analisis kebutuhan fungsional tidak diperlukan. Melainkan, berfokus pada pengumpulan kebutuhan *non-fungsional* karena memiliki dampak langsung pada pemilihan target teknologi. Selain itu, kebutuhan peralatan migrasi diperiksa dan semua kebutuhan dikelola untuk mengatasi perubahan.

2. *Legacy Analysis*

Legacy analysis dimulai dengan menganalisa *legacy system* kasar. *Legacy system* dinilai kualitas teknis dan nilai bisnis. Sistem kemudian dianalisis lebih rinci menggunakan metode *reverse-engineering*. Dengan begitu, *legacy system* sekarang bisa kembali direkayasa untuk meningkatkan kualitas sistem demi mendapatkan hasil yang lebih baik setelah transformasi.

3. *Target Design*

Selama *target design*, arsitektur, struktur data, dan antarmuka pengguna dari sistem baru didefinisikan. Di sini juga disertakan *artifact* sementara yang mungkin

digunakan selama perubahan. Selain itu, *legacy artifact* dipetakan ke target *artifact* yang sesuai.

4. *Strategi Selection*

Strategy selection merupakan tahapan memilih strategi migrasi. Strategi migrasi menjelaskan bagaimana *legacy system* akan dikonversi ke dalam sistem yang *dimigrasikan*. Terdapat tiga strategi migrasi yaitu:

a. *Re-Implementation* merupakan *re-development* sistem baru berdasarkan ide-ide yang diberikan oleh *legacy software*. Jika menggunakan strategi *re-implementation, legacy system* dianalisis dan kebutuhan fungsional dan *non-fungsional* diambil.
b. *Transformation* yaitu *mengkonversi* sistem secara otomatis menjadi target sistem. *Strategy transformation* mencoba untuk secara otomatis *mengkonversi legacy application* atau *legacy* data ke dalam teknologi yang baru. Strategi ini hanya bisa dilaksanakan dengan menggunakan alat transformasi. Biasanya, alat ini harus dirancang dan diimplementasikan sebelum migrasi. Dengan asumsi bahwa alat migrasi tidak punya kegagalan, transformasi otomatis *legacy software* akan menghasilkan salinan *one-to-one* dari sistem.
c. *Wraping* tidak mengubah *legacy system* tetapi menyediakan antarmuka baru untuk memberikan akses pada *legacy function* ke sistem yang baru. Strategi *wrapping* tidak mengubah *legacy system*. *Legacy function* tetap dalam *legacy system*. *Wrapper* yang ditulis mengakses *legacy function* dan menyediakan antarmuka untuk sistem yang baru sehingga sistem yang baru bisa mengakses *legacy function* melalui *wrapper*.

5. *Implementation*

Selama tahap implementasi, *legacy system* dimigrasikan sesuai dengan strategi migrasi yang dipilih.

6. *Test*

Salah satu tujuan utama dari migrasi perangkat lunak yaitu untuk membuat salinan *one-to-one functional* dari *legacy system*, Selam dalam tahapan *test*, sistem yang baru akan diuji, apakah mendukung fungsi yang sama dengan sistem yang lama dengan menggunakan *integration test* dan *unit test*.

7. *Deployment*

Di akhir proses migrasi yaitu bagaimana cara memperkenalkan sistem yang baru muncul. Seringkali, *legacy system* masih produktif dan tidak bisa ditutup dengan sederhana. Sehingga diperlukan strategi bagaimana cara menggunakan sistem yang baru dan menutup sistem yang lama. Strategi yang bisa digunakan antara lain:

a. Big-Bang Startegy

   Deployment menggunakan strategi *big-bang* dengan cara sistem yang baru di-*deploy* secara lengkap dan sekaligus menutup *legacy system* pada waktu yang bersamaan. Hal ini sangat beresiko karena seluruh pekerjaan yang dilakukan sistem baru belum terbukti. Sebaliknya, dengan strategi ini bisa menurunkan biaya migrasi.

b. Continuous Strategy

   Strategi ini hanya *memigrasikan* bagian-bagian tertentu dari *legacy system* yang sudah direkayasa ulang. Komponen lainnya tidak berubah sampai direkayasa ulang, Kelemahan dari strategi ini yaitu dapat menyebabkan waktu proyek menjadi sangat panjang.

II.4 Scala

Scala adalah singkatan dari *scalable language*. Dinamakan *scalable* karena Scala dirancang untuk tumbuh dengan penggunanya. Scala bisa digunakan untuk menulis kode program yang sederhana sampai dengan membangun sistem yang besar dan kompleks. Selain itu, Scala adalah perpaduan antara paradigma *Object Oriented Programming* (OOP) dan paradigma *Functional Programming* (FP). Kedua paradigma inilah yang menjadi alasan kenapa Scala menjadi bahasa yang *scalable*.

Paradigma OOP menjadikan mudah dalam menyusun struktur sistem yang besar dan mengadaptasinya terhadap permintaan yang baru. Paradigma FP menjadikan mudah dalam membangun hal yang menarik secara cepat dari bagian yang sederhana. Kelebihan dari Scala yaitu pertama, Scala kompatibel. Itu dikarenakan Scala berjalan di atas *Java Virtual Machine* (JVM) dan berbagi dengan

kode java serta dapat ditambahkan ke dalam kode yang sudah ada. Kedua, Scala sederhana, program yang ditulis dalam bahasa Scala selalu singkat dan ringkas. Ketiga, Scala memiliki *abstraction* tingkat tinggi yang berguna dalam mengatur dan mengelola kompleksitas program. Inilah yang menjadikan Scala menjadi salah satu Bahasa pemrograman yang mulai banyak penggunanya saat ini [1].

II.5 Functional Programming

Paradigma pemrograman yang memperlakukan proses komputasi sebagai evaluasi fungsi-fungsi matematika. Pemrograman fungsional didasarkan pada premis sederhana bahwa program-program yang dibangun menggunakan fungsi murni. Dalam pengertian yang terbatas, pemrograman fungsional berarti pemrograman tanpa menggunakan variabel *mutable*, *assignment*, *loops*, struktur kontrol *imperative* lainnya. Sedangkan dalam arti yang luas, pemrograman fungsional berarti berfokus pada fungsi. Pemrograman fungsional sangat berbeda dengan pemrograman konvensional. Dengan mempelajari pemrograman fungsional akan meningkatkan kualitas kode, baik dari sisi penulisan maupun perancangan [10]. Berikut adalah fitur-fitur dari pemrograman fungsional antara lain yaitu:

1. *First-Class Functions* artinya fungsi bisa disimpan ke dalam suatu variabel.
2. *High-Order Function* artinya fungsi bisa mengembalikan fungsi atau menerima fungsi lainnya sebagai parameter.
3. *Pure Function* artinya fungsi tidak mengubah nilai apapun, fungsi hanya menerima dan mengeluarkan data.
4. *Closures* artinya data bisa disimpan di dalam fungsi yang hanya bisa diakses untuk mengembalikan fungsi tertentu.
5. *Immutable State* artinya *state* yang telah diinisialisasikan tidak bisa di ubah.

II.6 Padanan PHP, Scala dan Java

Untuk memberikan gambaran tentang PHP, Scala, dan Java, berikut ini akan ditampilkan tabel yang menggambarkan *keyword* untuk mendefinisikan *variable*, fungsi, dan seterusnya yang dapat dilihat pada Tabel II.1 [1] [11] [12].

Tabel II.1 Padanan PHP, Scala, dan Java

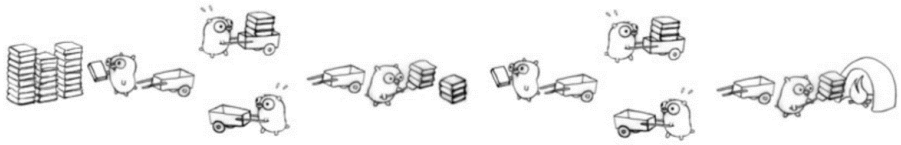
Pendefinisian	Bahasa	Keyword	Keterangan
Variabel	PHP 5	$namaVar	*keyword* untuk mendefinisikan variabel
		define	*keyword* untuk mendefinisikan *constanta*
	Scala	var	*keyword* untuk mendefinisikan variabel yang nilainya *mutable*
		val	*keyword* untuk mendefinisikan *value* yang nilainya *immutable*
	Java	type namaVar	*keyword* untuk mendefinisikan variabel
Output Data	PHP 5	echo	*keyword* untuk *output* data ke layar
		print	
	Scala	println	
	Java	println	
Fungsi	PHP 5	function	*keyword* untuk mendefinisikan fungsi
	Scala	def	
	Java	def	
Array	PHP 5	array	*keyword* untuk mendefinisikan *array*
	Scala	Array	
	Java	type[] var-name	
		type var-name[]	
Abstract	Scala	abstract	*keyword* untuk mendefinisikan kelas abstrak
	Java	abstract	
Class	PHP 5	class	*keyword* untuk mendefinisikan kelas
	Scala	class	
	Java	class	
Object	Scala	object	*keyword* untuk mendefinisikan *object*.
Namespace	PHP 5	namespace	*keyword* untuk mendefinisikan *namespace*
Package	Scala	package	*keyword* untuk mendefinisikan *package*
	Java	package	
Interface	Scala	interface	*keyword* untuk mendefinisikan *interface*. Digunakan untuk mengenkapsulasi *method* dan pendefinisian *field*
Trait	Java	trait	*keyword* untuk mendefinisikan *trait*. Digunakan untuk mengenkapsulasi *method* dan pendefinisian *field*
Collection	Scala	List	mendefinisikan array dengan tipe *immutable* singly linked list
		Set	mendefinisikan *array* dengan tipe *immutable* dan *unordered collection*, yang kerjanya mirip dengan *List*
		Map	mendefinisikan *array* dengan format *immutable key-value store*, atau istilah lainnya yaitu *associative array*

II.7 Play Framework

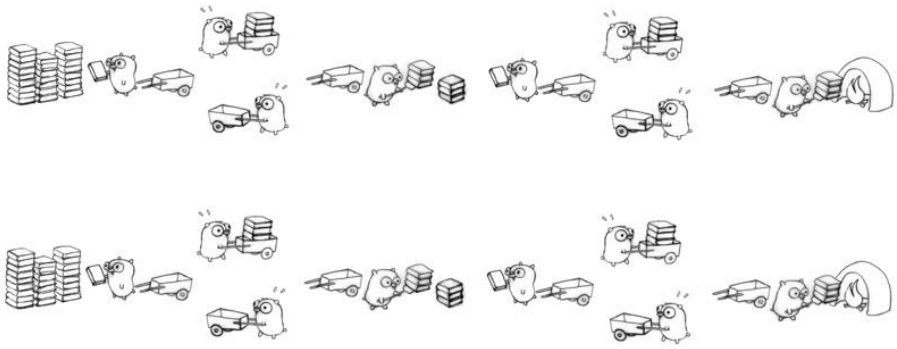
Play adalah *full stack web framework* berkecepatan tinggi untuk Java dan Scala yang memungkinkan alur kerja yang sangat produktif tanpa mengorbankan skalabilitas. Play menerapkan arsitektur MVC dengan rute *file* yang memetakan HTTP *request* ke *controller*, kemudian mengambil permintaan informasi dan menghasilkan sebuah hasil representasi. *Play* dibangun untuk *web* modern dengan konsep *non-blocking*, RESTful secara *default*, dan memiliki *built-in asset compiler* untuk teknologi *client-side* seperti CoffeeScript dan LESS [13].

II.8 Concurrency dan Parallelism

Concurrency dan *paralelisme* adalah konsep yang saling berkaitan, tetapi ada perbedaan yang kecil. *Concurrency* artinya dua atau lebih tugas yang membuat *progress* (kemajuan) tetapi tidak di eksekusi secara bersamaan. Hal ini dapat dicapai dengan mengiris waktu (*time slicing*) ketika bagian dari tugas di eksekusi secara berurutan dan mencampur dengan tugas-tugas yang lain. Paralelisme, dari sisi lain muncul ketika eksekusinya dilakukan benar-benar simultan. Untuk lebih memahami kedua konsep ini dapat dilihat pada Gambar II.1 dan Gambar II.2 [14].



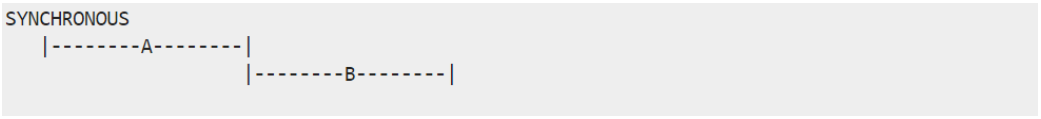
Gambar II.1 *Concurrency*



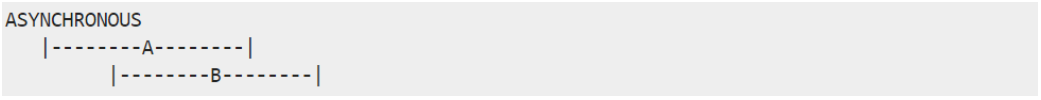
Gambar II.2 *Parallelism*

II.9 Asynchronous dan Synchronous

Suatu pemanggilan *method* dianggap (*considered*) *synchronous* jika pemanggil (*caller*) tidak bisa membuat kemajuan (*progress*) sampai *method* mengembalikan nilai atau *throw exception*. Di lain sisi, panggilan secara *asynchronous*, memungkinkan pemanggil (*caller*) membuat kemajuan setelah beberapa langkah, dan penyelesaian metode dapat ditandai dengan beberapa mekanisme tambahan seperti *callback*, *future*, atau *message*. Untuk lebih memahami kedua konsep ini dapat dilihat pada Gambar II.3 dan Gambar II.4 [14].



Gambar II.3 *Synchronous*



Gambar II.4 *Asynchronous*

Di Scala, untuk menerapkan konsep *concurrent* dan *asynchronous* dilakukan dengan cara menambahkan *library* berikut:

```
import scala.concurrent.ExecutionContext.Implicits._
import scala.concurrent.Future
```

Setelah itu, menambahkan *keyword Future* atau *Action.async* di setiap *method* kelas Scala yang menerapkan konsep ini yang akan dicontohkan di bawah ini:

```
import scala.concurrent.ExecutionContext.Implicits._
import scala.concurrent.Future

// Method bagian services
def method1(param: tipeParam): Future[Option[kelasA]] = {
  TODO }
// Method bagian controllers
def method2(param: tipeParam) = Action.async { implicit request
=>
  TODO }
```

Sintaks Future pada kode di atas mengartikan bahwa method1 menghasilkan kelasA yang dieksekusi secara *concurrent* dan *asynchronous*. Hal ini juga berlaku untuk *sintaks Action.async* pada method2 [15].

II.10 Slick (Scala Language-Integrated Connection Kit)

Slick merupakan *library* Scala untuk mengakses *database* relasional dengan menggunakan antarmuka yang mirip *library* Scala *collection*. Slick memperlakukan *query* seperti *collections*, mentransformasi dan menggabungkan mereka dengan *method* map, flatMap, dan filter sebelum dikirim ke *database* untuk me-*fetch result*. Berikut ini adalah fitur-fitur yang dimiliki Slick [16]:

1. *Functional Relational Mapping*

   Slick merupakan paradigma *Functional Relational Mapping* (FRM) baru yang memungkinkan pemetaan (*mapping*) selesai di dalam Scala dengan sejumlah keuntungan kompleksitas abstraksi ketika dihubungkan dengan *database* relasional. Beberapa keuntungan dari FRM Slick untuk pemrograman fungsional yaitu:

   a. Efesiensi Dengan *Pre-Optimization*

      FRM merupakan cara yang lebih efesien untuk menghubungkan (*connect*), ia memiliki kemampuan *pre-optimize* untuk komunikasi dengan *database* dan membuat aplikasi lebih cepat.

   b. Tidak Ada Lagi *Troubleshooting* Membosankan Dengan *Type Safety*

      FRM membawa *type safety* untuk membangun *query database* sehingga developer lebih produktif karena *compiler* menemukan kesalahan secara otomatis.

c. *Composable* Model Untuk Membangun *Queries*

FRM mendukung *composable* model untuk membangun *query*. Ini adalah model yang natural untuk menyusun potongan bersamaan (*pieces together*) dalam membangun *query*, kemudian menggunakan kembali potongan tersebut di basis kode. Berikut adalah contoh dari *compose* model dari Slick:

```
// Create a query for coffee names with a price less than 10, sorted by name
coffees.filter(_.price < 10.0).sortBy(_.name).map(_.name)
// The generated SQL is equivalent to:
// select name from COFFEES where PRICE < 10.0 order by NAME
```

2. *Reactive Applications*

Slick mudah digunakan dalam perancangan aplikasi *asynchronous*, *non-blocking*, dan mendukung pembangunan aplikasi sesuai dengan *Reactive Manifesto*.

II.11 Cloud Computing

Cloud computing merupakan gaya komputasi di mana sumber daya komputasi seperti aplikasi dan media penyimpanan yang tersedia melalui internet, biasanya melalui *web browser*. Dengan *cloud computing* banyak *web browser* mampu untuk melakukan sinkronisasi dengan aplikasi-aplikasi lainnya. *Cloud computing* adalah suatu metode komputasi di mana kapabilitas teknologi informasi disajikan sebagai suatu layanan, sehingga pengguna dapat mengakses dengan mudah melalui internet tanpa mengetahui apa yang ada di dalam, ahli menggunakan atau memiliki kendali terhadap infrastruktur teknologi yang digunakan. Menyimpan data di *cloud* dapat memberikan keuntungan dari segi peningkatan ruang penyimpanan, sebagai *backup online*, dan akses data dapat dilakukan dari mana saja selama mempunyai koneksi internet [17].

II.12 Heroku Cloud

Heroku merupakan salah satu penyedia *platform-as-a-service* (PaaS) terkemuka di bisnis *cloud software*, yang membuktikan dirinya sebagai solusi PaaS terkemuka untuk perusahaan kecil dan besar. Dengan peningkatan yang konsisten dan filosofinya kenyamanan melebihi konfigurasi, menjadikan developer untuk fokus dalam menulis aplikasi *web*. Hal yang bersangkutan dengan *server*, *building*, *deploying*, *running*, dan *scaling* aplikasi akan diatasi oleh Heroku.

Heroku menyediakan dukungan platform untuk Ruby, Ruby on Rails, Java, Node.js, Clojure, Scala, Python, dan PHP. Arsitektur Heroku add-on memungkinkan developer menyesuaikan penggunaan berbagai paket pihak ketiga berdasarkan kebutuhan. Developer memiliki fleksibilitas untuk memilih *plan basic* atau *premium* berdasarkan kebutuhan website. Developer juga bisa mempercepat aplikasi dengan menambahkan *resources* seperti memcached untuk *caching* data, sehingga menciptakan aplikasi yang *powerful* yang kaya dengan fitur [18].

II.13 Structure Analysis and Design

Struktur analisis dan desain adalah sebuah metodologi yang di gunakan pada rekayasa perangkat lunak untuk mendeskripsikan sistem ke arah fungsional. Pendekatan ini memecahkan masalah-masalah dalam aktivitas bisnis menjadi bagian-bagian kecil yang dapat disatukan kembali menjadi satu kesatuan yang utuh untuk memecahkan masalah [19].

Tools yang dapat digunakan pada pendekatan analisis pengembangan sistem secara terstruktur adalah:

1. ERD (*Entity Relationship* Diagram) adalah satu model jaringan yang menggunakan susunan data yang disimpan dalam sistem. ERD merupakan model jaringan data yang menekankan pada struktur dan hubungan antar data [20].
2. Diagram Konteks adalah diagram yang menggambarkan sebuah sistem secara menyeluruh yang selanjutnya akan lebih didetailkan oleh DFD. Konteks diagram juga biasa disebut sebagai DFD level 0.

3. Data *Flow* Diagram atau biasa di singkat DFD merupakan serangkaian diagram yang menggambarkan kegiatan-kegiatan yang ada dalam satu sistem. Teknik pembangunan DFD dimulai dengan menggambarkan sistem secara global dan dilanjutkan dengan analisis masing-masing bagian. Pada awalnya digambarkan konteks diagram yang menggambarkan sebuah sistem secara menyeluruh yang akan diinvestigasi. Konteks diagram tersebut dapat dikatakan sebagai DFD level 0. Analisis sistem yang lebih detail selanjutnya dapat dilakukan dengan menggambarkan DFD level 1,2 dan seterusnya.
4. Spesifikasi Proses merupakan tabel yang berisi keterangan deskripsi dari semua proses yang terdapat di DFD. Logika proses harus dituliskan secara jelas baik menggunakan bahasa deskriptif atau *pseudo code* (tidak boleh campuran).
5. Kamus Data (*Data Dictionary*) merupakan fakta tentang data dan kebutuhan-kebutuhan informasi dari sistem informasi. Dengan menggunakan *data dictionary*, analis sistem dapat mendefinisikan data yang mengalir dalam sistem dengan lengkap [21].

II.14 Object Oriented Analysis and Design

Analisis dan desain berorientasi objek adalah cara baru dalam memikirkan satu masalah dengan menggunakan model yang dibuat menurut konsep sekitar dunia nyata. Tujuan dari analisis berorientasi objek adalah untuk mengembangkan model yang menggambarkan perangkat lunak komputer karena bekerja untuk memenuhi seperangkat persyaratan yang ditentukan *user* [22]. *Tools* yang dapat digunakan pada pendekatan analisis pengembangan sistem secara objek yaitu menggunakan UML.

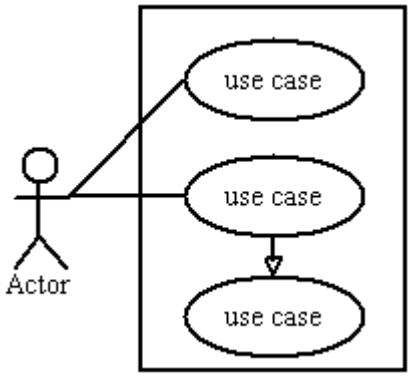
Unified Modelling Language (UML) adalah sebuah bahasa yang telah menjadi standar dalam industri untuk visualisasi, merancang dan mendokumentasikan sistem piranti lunak. UML menggunakan *class* dan *operation object* dalam konsep dasarnya, maka ia lebih cocok untuk penulisan piranti lunak dalam bahasa bahasa berorientasi objek [23]. Dalam membangun block UML ada tiga hal yang harus diperhatikan, yaitu *object* (memodelkan konsep), *relationship* (menghubungkan *object*), dan *diagram* (*grouping* yang saling menghubungkan

antara *object* dan *relationship*. Diagram yang umum dipakai dalam analisis dan desain adalah:

1. *Use Case Diagram*

Use Case Diagram menggambarkan fungsionalitas yang diharapkan dari sebuah sistem. Yang ditekankan adalah "apa" yang diperbuat sistem, dan bukan "bagaimana". Sebuah *Use case* merepresentasikan sebuah interaksi antara aktor dengan sistem. Sebuah *use case* dapat meng-*include* fungsionalitas *use case* lain sebagai bagian dari proses dalam dirinya. Secara umum diasumsikan bahwa *use case* yang di-*include* akan dipanggil setiap kali *use case* yang meng-*include* dieksekusi secara normal. Sebuah *use case* dapat di-*include* oleh lebih dari satu *use case* lain, sehingga duplikasi fungsionalitas dapat dihindari dengan cara menarik keluar fungsionalitas yang serupa. Sebuah *use case* juga dapat meng-*extend usecase* lain dengan *behaviour*-nya sendiri. Sementara hubungan generalisasi antar *use case* menunjukkan bahwa *use case* yang satu merupakan spesialisasi dari yang lain.

Dasar menentukan sebuah *use case* adalah *use case* merupakan sesuatu yang menyediakan beberapa hasil terukur kepada pengguna atau sistem eksternal. *Use case* harus memiliki sangat jelas kriteria lulus / gagal. Pengembang, tester, penulis teknis, dan pengguna harus secara eksplisit tahu apakah sistem memenuhi kasus penggunaan atau tidak. Setiap bagian dari *use case* yang memenuhi tes sederhana ini mungkin menjadi kandidat yang baik untuk *use case* [21]. Dasar pembangunan *use case diagram* dapat dilihat pada Gambar II.5.



Gambar II.5 Dasar Pembangunan *Use Case Diagram*

2. *Use Case Scenario*

Sebuah diagram yang menunjukkan *use case* dan aktor mungkin menjadi titik awal yang bagus, tetapi tidak memberikan detail yang cukup untuk desainer sistem untuk benar-benar memahami persis bagaimana sistem dapat terpenuhi. Cara terbaik untuk mengungkapkan informasi penting ini adalah dalam bentuk penggunaan *use case scenario* berbasis teks per *use case* nya. Berikut adalah dasar format penulisan *use case scenario* [24]. Dasar pembangunan *use case scenario* dapat dilihat pada Tabel II.2.

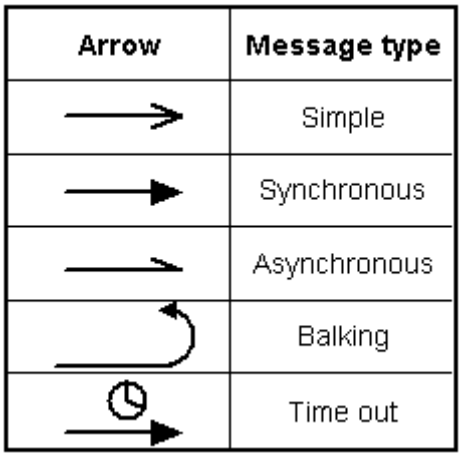
Tabel II.2 Dasar Pembangunan *Use Case Scenario*

Use Case Name	Berisi nama dari *use case* yang akan digunakan	
Goal In Context	Menjelaskan apa yang aktor coba untuk dapatkan dari *use case*	
Description	Menjelaskan gambaran dari *use case*	
Related Use Case	Daftar *use case* yang berhubungan dengan *use case* tersebut	
Successful End Condition	Kondisi *use case* jika berhasil	
Failed End Condition	Kondisi *use case* jika gagal	
Actors	Daftar aktor yang dapat mengakses *use case*	
Trigger	Aktifitas yang dilakukan untuk mengawali *use case*	
Main Flow	Step	Action
	1	Deskripsi urutan aksi dari aktifitas *use case*
	2	
Extension	Step	Branching Action
	2.1	Deskripsi urutan aksi lain selain urutan aksi utama

3. *Sequence Diagram*

Sequence diagram menggambarkan interaksi antar objek di dalam dan di sekitar sistem berupa *message* yang digambarkan terhadap waktu. *Sequence diagram* terdiri atar dimensi vertikal (waktu) dan dimensi horizontal (objek-objek yang terkait). *Sequence diagram* biasa digunakan untuk menggambarkan skenario atau rangkaian langkah-langkah yang dilakukan sebagai respons dari sebuah *event* untuk menghasilkan *output* tertentu. Diawali dari apa yang men-*trigger* aktivitas tersebut, proses dan perubahan apa saja yang terjadi secara internal dan *output* apa yang dihasilkan.

Masing-masing objek, termasuk aktor, memiliki *lifeline* vertikal. *Message* digambarkan sebagai garis berpanah dari satu objek ke objek lainnya. Pada fase desain berikutnya, *message* akan dipetakan menjadi operasi/metoda dari *class*. *Activation bar* menunjukkan lamanya eksekusi sebuah proses, biasanya diawali dengan diterimanya sebuah *message* [23]. Untuk objek-objek yang memiliki sifat khusus, standar UML mendefinisikan icon khusus untuk objek *boundary*, *controller* dan *persistent entity*. Dasar pembangunan *sequence diagram* dapat dilihat pada Gambar II.6.



Gambar II.6 Dasar Pembangunan *Sequence Diagram*

4. *Class Diagram*

Class adalah sebuah spesifikasi yang jika diinstansiasi akan menghasilkan sebuah objek dan merupakan inti dari pengembangan dan desain berorientasi objek. Class menggambarkan keadaan (atribut/properti) satu sistem, sekaligus menawarkan layanan untuk memanipulasi keadaan tersebut (metoda/fungsi). Class diagram menggambarkan struktur dan deskripsi *class*, *package* dan objek beserta hubungan satu sama lain seperti *containment*, pewarisan, asosiasi, dan lain-lain. Class memiliki tiga area pokok:

1. Nama dan stereotype
2. Atribut
3. Metoda

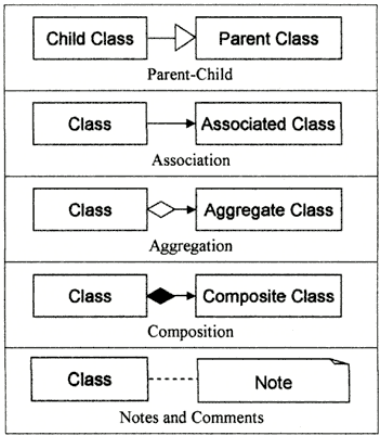
Atribut dan metoda dapat memiliki salah satu sifat berikut:

1. *Private*, tidak dapat dipanggil dari luar *class* yang bersangkutan.
2. *Protected*, hanya dapat dipanggil oleh *class* yang bersangkutan dan anak-anak yang mewarisinya.
3. *Public*, dapat dipanggil oleh siapa saja.

Class dapat merupakan implementasi dari sebuah *interface*, yaitu *class* abstrak yang hanya memiliki metoda. *Interface* tidak dapat langsung diinstansiasikan, tetapi harus diimplementasikan dahulu menjadi sebuah *class*. Sesuai dengan perkembangan *class* model, *class* dapat dikelompokkan menjadi *package*. Kita juga dapat membuat diagram yang terdiri atas *package*.

Class memiliki tipe-tipe *relationship*, diantaranya:

1. Asosiasi, yaitu hubungan statis antar *class*. Umumnya menggambarkan *class* yang memiliki atribut berupa *class* lain, atau *class* yang harus mengetahui eksistensi *class* lain.
2. Agregasi, yaitu hubungan yang menyatakan bagian terdiri atas dimana ketika satu *class* di-*share* atau direferensikan kepada objek yang ada di *class* lain.
3. Pewarisan, yaitu hubungan hirarkis antar *class*. Class dapat diturunkan dari *class* lain dan mewarisi semua atribut dan metoda *class* asalnya dan menambahkan fungsionalitas baru, sehingga ia disebut anak dari *class* yang diwarisinya. Kebalikan dari pewarisan adalah generalisasi.
4. Komposisi, yaitu jenis relasi *class diagram* yang kuat dimana jika sebuah *class* tidak bisa berdiri sendiri dan harus merupakan bagian dari *class* yang lain, maka *class* tersebut memiliki relasi *Composition* terhadap *class* tempat dia bergantung tersebut. Sebuah *relationship composition* digambarkan sebagai garis dengan ujung berbentuk jajaran genjang berisi/solid.
5. Depedensi, salah satu jenis relasi *class* diagram yang lemah dimana objek dalam suatu *class* akan bekerja sangat singkat dengan objek yang ada pada *class* lain.



Gambar II.7 Dasar Pembangunan *Class Diagram*

5. *Activity Diagram*

Activity diagram menggambarkan berbagai alur aktivitas dalam sistem yang sedang dirancang, bagaimana masing-masing aliran berawal, *decision* yang mungkin terjadi, dan bagaimana mereka berakhir. *Activity* diagram juga dapat menggambarkan proses paralel yang mungkin terjadi pada beberapa eksekusi. Sebuah aktivitas dapat direalisasikan oleh satu *usecase* atau lebih. Aktivitas menggambarkan proses yang berjalan, sementara *usecase* menggambarkan bagaimana aktor menggunakan sistem untuk melakukan aktivitas. Standar UML menggunakan segiempat dengan sudut membulat untuk menggambarkan aktivitas. *Decision* digunakan untuk menggambarkan *behaviour* pada kondisi tertentu. Untuk mengilustrasikan proses-proses paralel (*fork* dan *join*) digunakan titik sinkronisasi yang dapat berupa titik, garis horizontal atau vertikal. *Activity* diagram dapat dibagi menjadi beberapa *object swimlane* untuk menggambarkan objek mana yang bertanggung jawab untuk aktivitas tertentu [23].

II.15 Pengujian Unit (Unit Testing)

Pengujian unit merupakan pengujian yang menguji bagian terkecil dari kode, biasanya fungsi individu, sendirian, dan terisolasi. Ketika pengujian menggunakan beberapa sumber daya eksternal, seperti *database*, hal ini bukanlah pengujian unit. Pengujian unit pada dasarnya hanya memberikan fungsi yang diuji dengan beberapa input, dan kemudian memeriksa apakah *output* dari fungsi benar [25].

II.16 Pengujian Integrasi (Integration Testing)

Pengujian integrasi merupakan pengujian mengenai bagaimana menguji bagian-bagian sistem bekerja sama. Pengujian integrasi serupa dengan pengujian unit, tetapi memiliki satu perbedaan besar yaitu ketika pengujian unit terisolasi dari komponen lain sedangkan pengujian integrasi tidak. Sebagai contoh, pengujian unit kode *database* tidak akan berhubungan dengan *database* yang asli, sedangkan pengujian integrasi berhubungan dengan *database* asli [25].

II.17 Pengujian Performa (Performance Testing)

Pengujian performa adalah teknik investigasi yang dilakukan untuk menentukan atau memvalidasi kecepatan, skalabilitas, dan stabilitas dari aplikasi atau produk yang diuji. Tujuan pengujian performa biasanya berkaitan untuk mengetahui *response time*, *throughput* (transaksi per detik), dan tingkat pemanfaatan sumbe daya (*resources-utilization*).

Response time adalah ukuran seberapa responsif suatu aplikasi terhadap *client request*, sedangkan *throughput* merupakan jumlah unit kerja yang dapat ditangani per unit waktu, misalkan permintaan per detik, hits per detik, panggilan per hari, dan lain-lain [24]. Berikut ada beberapa definisi yang ada dalam pengujian performa [27] [28]:

1. #Sample yaitu mewakili banyaknya *sample* yang dieksekusi.
2. *Average latency* artinya rata-rata *latency* setiap *request* yang di eksekusi.
3. *Latency* (penundaan) adalah waktu untuk blok data tertentu untuk mendapatkan dari satu titik ke titik lain. *Latency* diukur dengan mengirimkan paket yang akan dikembalikan kepada pengirim dan waktu pulang-pergi dianggap *latency*.
4. *Average response time* artinya rata-rata waktu waktu *respon* setiap *request* yang dieksekusi.
5. *Response time* yaitu ukuran kinerja transaksi individu atau *query* dan jumlah waktu saat pengguna mengirimkan permintaan sampai saat aplikasi menunjukan bahwa permintaan telah selesai.

6. 90% *line* (90 *percentile* of *the response times*) merupakan salah satu *metric* yang paling menarik dan menunjukan bahwa "90% dari sampel selesai dalam waktu 0-x" atau "90% dari sampel mengambil tidak lebih dari saat ini".
7. 95% *line* (95 *percentile of the response times*) menunjukan bahwa "95% dari sampel selesai dalam waktu 0-x" atau "95% dari sampel mengambil tidak lebih dari saat ini".
8. 99% *line* (99 *percentile of the response times*) menunjukan bahwa "99% dari sampel selesai dalam waktu 0-x" atau "99% dari sampel mengambil tidak lebih dari saat ini".
9. *Minimum response time* artinya waktu tersingkat untuk *sample* dengan label yang sama.
10. *Maximum response time* artinya waktu terpanjang untuk *sample* dengan label yang sama.
11. *Average bandwidth* artinya ukuran *traffic* yang dibuat per detik dalam satuan *Bytes*.
12. *Average throughput* (Hits/s), Rata-rata jumlah permintaan HTTP/s per detik yang dihasilkan oleh tes.
13. *Error* %, tingkat kesalahan per label.

II.18 Blazemeter

Blazemeter merupakan suatu platform pengujian performa yang berbasis *cloud* untuk *web* dan aplikasi *mobile* yang dibuat untuk profesional *developers, operations, devops* dan QA. Blazemeter dapat melakukan hal sebagai berikut [27]:

1. Menguji *load* setiap *website*, *web app*, *mobile app,* API ataupun *web service*.
2. Menemukan dan memperbaiki hambatan performa.
3. Merealisasikan perencanaan kapasitas.
4. *Performance benchmarking*.

II.19 Web Service

Web service adalah suatu sistem perangkat lunak yang dirancang untuk mendukung interoperabilitas dan interaksi antar sistem pada suatu jaringan. *Web service* memiliki *interface* yang menggambarkan kumpulan operasi yang diakses melalui jaringan, menjalankan sebuah atau kumpulan tugas secara spesifik, dideskripsikan menggunakan standar notasi XML formal yang disebut sebagai *service description* yang menyediakan semua rincian kebutuhan untuk interaksi dengan *service* (layanan) termasuk *message* format (rincian tentang operasi), transportasi protokol, dan lokasinya [29].

II.20 RESTful Web Services

REpresentational State Transfer (REST) merupakan sebuah metode dalam menyampaikan *resource* melalu media *web*. *Resource* sendiri bisa didefinisikan sebagai segala sesuatu yang dapat disimpan didalam sebuah *computer* dan ditampilkan sebagai urutan bit, misalnya dokumen, *row* dalam *database*, atau hasil dari sebuah perhitungan algoritma [30]. REST adalah model arsitekur yang pada dasarnya memanfaatkan teknologi dan protokol yang sudah ada seperti HTTP (*Hypertext Transfer Protocol*) dan XML.

Arsitektur REST dibangun dengan sifat sebagai berikut [31]:

1. *Addressability*

Dalam prinsip ini seluruh *resource* harus tersedia melalui sebuah alamat unik, pengalamatan ini dilakukan dengan menggunakan URI (*Unique Resource Identifier*s).

2. *Uniform Interface*

RESTful service menampilkan semua *resource* dan interaksinya dengan *interface* yang seragam, dalam metode REST antarmuka yang digunakan adalah dengan menggunakan HTTP. HTTP menawarkan semua operasi yang diperlukan, dikenal, dan tersebar luas. Semua interaksi antara klien dan *resource* didasarkan pada metode dasar HTTP. Metode-metode interaksi yang digunakan dengan dasar HTTP ditunjukkan pada Tabel II.3.

3. Representation-oriented

Representasi menjelaskan dalam bentuk apa data sedang dipertukarkan antara *client* dan *server*. Pada umumnya data dipertukarkan dalam bentuk XML, JSON, dan HTML.

4. Statelessness

Setiap interaksi antara *client* dan *server* harus memiliki *state* sendiri. Jadi *server* hanya akan memantau *resource state* bukan *client session*.

Berikut merupakan penggunaan metode HTTP dalam REST *Web Service*:

Tabel II.3 Metode HTTP REST *Web Service*

Metode	Deskripsi
GET	Mendapatkan (*read*) sebuah sumber daya (*resource*) yang diidentifikasi dengan URI (*Uniform Resource Identifier*)
POST	Mengirimkan sumber daya (*resource*) ke *server*. Digunakan untuk membuat (*create*) sumber daya baru.
PUT	Mengirimkan sumber daya (*resource*) ke *server*. Digunakan untuk memasukkan (*insert*) atau memperbarui (*update*) sumber daya yang tersimpan.
DELETE	Menghapus (*delete*) sumber daya (*resource*) yang diidentifikasi dengan URI
HEAD	Mendapatkan meta data (*response header*) dari sumber daya(*resource*) yang diidentifikasi dengan URI.

II.21 JSON (JavaScript Object Notation)

JavaScript Object Notation (JSON) adalah format pertukaran data yang ringan, mudah dibaca dan ditulis oleh manusia, serta mudah diterjemahkan dan dibuat (*generate*) oleh komputer [32]. JSON terbuat dari dua struktur yaitu:

1. Kumpulan pasangan nama/nilai. Pada beberapa bahasa, hal ini dinyatakan sebagai objek (*object*), rekaman (*record*), struktur (*struct*), kamus (*dictionary*), tabel hash (*hash table*), daftar berkunci (*keyed list*), atau associative array.
2. Daftar nilai terurutkan (*list of values*). Pada kebanyakan bahasa, hal ini dinyatakan sebagai larik (*array*), vektor (*vector*), daftar (*list*), atau urutan (*sequence*).

JSON menggunakan bentuk sebagai berikut:

1. Objek

   Objek adalah sepasang nama/nilai yang tidak terurutkan. Objek dimulai dengan { (kurung kurawal buka) dan diakhiri dengan } (kurung kurawal tutup). Setiap nama diikuti dengan : (titik dua) dan setiap pasangan nama/nilai dipisahkan oleh , (koma).

2. Larik

   Larik adalah kumpulan nilai yang terurutkan. Larik dimulai dengan [(kurung kotak buka) dan diakhiri dengan] (kurung kotak tutup). Setiap nilai dipisahkan oleh , (koma).

3. Nilai

   Nilai (value) dapat berupa sebuah string dalam tanda kutip ganda, atau angka, atau true atau false atau null, atau sebuah objek atau sebuah larik. Struktur-struktur tersebut dapat disusun bertingkat.

4. String

   String adalah kumpulan dari nol atau lebih karakter Unicode, yang dibungkus dengan tanda kutip ganda. Di dalam string dapat digunakan *backslash escapes* "\" untuk membentuk karakter khusus. Sebuah karakter mewakili karakter tunggal pada string. String sangat mirip dengan string C atau Java.

5. Angka, Format oktal dan heksadesimal tidak digunakan.