PolarSSL voor Ruby

Ontwikkeling van een Ruby C extension om de PolarSSL security library aan te spreken vanuit de programmeertaal Ruby.

Afstudeerverslag

Michiel Sikkes
Studentnummer 20062643

Referaat

Auteur: Michiel Sikkes
Opdracht: Het ontwerpen en implementeren van een software wrapper voor waarmee functionaliteit uit de PolarSSL C-softwarebibliotheek gebruikt kan worden in een Ruby programma.

In de periode mei 2013 t/m oktober 2013 heb ik deze software binnen het bedrijf Offspark B.V. ontwikkeld in het kader van mijn afstuderen. Dit verslag beschrijft de tools en ontwerp- en implementatiekeuzes die gemaakt zijn. De afstudeeropdracht is uitgevoerd door Michiel Sikkes voor de opleiding Informatica aan de Haagse Hogeschool.

Descriptoren:

- PolarSSL library
- SSL connectie
- Cryptografie
- Software wrapper
- De programmeertaal C
- De programmeertaal Ruby
- Geheugenallocatie
- Garbage Collection
- Test Driven Development
- Open source
- Lean
- Kanban
- eXtreme Programming
- Ruby on Rails
- MVC

Voorwoord

Als er een compiler voor een programmeertaal wordt ontwikkeld is het vaak een ontzettend grote mijlpaal als het punt bereikt wordt waarop de compiler zichzelf kan compilen. Het betekent dat de compiler “volwassen” geworden is.

De euforie van dit volwassen worden voelde ik ook een beetje tijdens het werken aan deze afstudeeropdracht: ik was bezig de programmeertaal Ruby uit te breiden met mijn eigen creaties, zodat andere programmeurs daarmee aan de slag konden gaan. Ik programmeerde namelijk nieuwe functionaliteit in een programmeertaal zodat andere programmeurs die functionaliteit weer in hun eigen programma’s konden gebruiken. Ik werd een volwassen programmeur.

De kennis voor deze uitdaging had ik niet kunnen vinden zonder mijn begeleider Paul Bakker en zijn bedrijf Offspark B.V. Hoewel ik me er een klein beetje voor schaamde, schroomde Paul niet om mij op weg te helpen met de meest basale principes van de programmeertaal C onder te knie te krijgen, die eigenlijk elke low-level programmeur in de kinderwieg wel zou moeten snappen. Wat had ik het als high level software ontwikkelaar toch altijd makkelijk gehad.

Zonder mijn zakenpartner Bob Jansen was me de uitvoer van deze opdracht ook niet gelukt. Blijkbaar hebben wij samen de afgelopen jaren zo een degelijk bedrijf opgezet dat het kwa tijd en kosten mogelijk was een afstudeeropdracht naast de reguliere zakelijke werkzaamheden uit te voeren.

Dank aan iedereen die mijn gebrabbel over *wrappers*, *pointers*, *Ruby* en *Intercity* wilde aanhoren en feedback heeft willen geven aan alles wat er nodig was om deze opdracht uit te voeren. Hallo Joshua en Youri!

Tot slot dank aan mijn docenten en begeleiders van de Academie voor ICT & Media in Zoetermeer, met in het bijzonder Vincent Broeren, Remco Ruijsenaars , Tim Cocx, Arno Nederend en Cees van Diest. Zij hebben ervoor gezorgd dat ik in de afgelopen 7 jaar een studie heb kunnen volgen en toegestaan dat ik een full-time bedrijf heb kunnen opzetten.

Michiel Sikkes
2 oktober 2013, Gouda

Inhoudsopgave

Inleiding

Dit verslag is geschreven in het kader van de afstudeeropdracht van Michiel Sikkes bij het bedrijf Offspark B.V. Het doel van dit verslag is inzage geven in proces dat is gevolgd zodat bepaald kan worden of er op een HBO niveau afgestudeerd is.

Het onderwerp van de afstudeeropdracht is de *software library* PolarSSL. Deze *library* wordt door Offspark B.V. ontwikkeld en verkocht. De library is geschreven voor de programmeertaal C. Het doel van deze afstudeeropdracht was de functionaliteit uit deze library beschikbaar te maken voor programmeurs van een andere programmeertaal, Ruby. Dit heeft geresulteerd in de ontwikkeling van het product *PolarSSL for Ruby*, een zogenoemde *wrapper library*.

Het verslag is opgedeeld in drie delen. In deel 1 wordt de opdracht, het afstudeerbedrijf, de achtergrond van de opdracht en de totstandkoming van het Plan van Aanpak beschreven. In het 2e deel wordt beschreven hoe de opgeleverde producten tot stand zijn gekomen en welke beslissingen hier aan ten grondslag gelegen hebben. Het derde deel bevat een evaluatie van de gekozen aanpak en de opgeleverde producten.

In deze afstudeeropdracht wordt de competentie van drie beroepstaken bewezen:

1. Selecteren van standaardsoftware
2. Ontwerpen systeemdeel
3. Bouwen applicatie

In hoofdstuk 2 zullen deze beroepstaken toegelicht worden in de context van de afstudeeropdracht.

Leeswijzer

In dit verslag zijn achtergrondinformatie en vaktermen nader uitgelegd in groene kaders. Hiernaast is een voorbeeld van een dergelijk kader weergegeven.

Dit is een voorbeeld van een theoretisch kader.

De woorden die *schuingedrukt* zijn in de verklarende woordenlijst opgenomen. Deze lijst is te vinden aan het eind van dit verslag bij de bijlagen.

Code-voorbeelden en termen uit de programmeertalen Ruby en C zijn weergegeven met een `monospace` lettertype. Een voorbeeld hiervan is bijvoorbeeld het `struct` datatype uit de programmeertaal C. Uitvoerbare code-voorbeelden zijn opgenomen in een kader:

```
In dit kader wordt (een deel van) een script of programma getoond.
```

Deel 1 - Opdracht en Plan van Aanpak

1. Achtergrond

Bedrijf

Offspark B.V. is een bedrijf gevestigd in Rijswijk en opgericht door mijn begeleider Paul Bakker. Het bedrijf levert producten en adviesdiensten die te maken hebben met cybersecurity, veilige hardware en software, en cryptografie. Voor het oprichten van Offspark B.V. was Paul Bakker één van de initiatiefnemers en een voortrekker van de *Crypto & High Security* afdeling bij IT beveiligingsspecialist Fox-IT.

Kenmerkend aan de werkwijze van Offspark B.V. is dat er met de *Lean*[1]-filosofie gewerkt wordt. Deze filosofie is een afgeleide van het Toyota Production System (TPS)[2], een productieproces ontwikkeld door autofabrikant Toyota. Het belangrijkste kenmerk van deze filosofie is dat er gezorgd wordt zo min mogelijk energie te verspillen, ofwel zo min mogelijk verspilling te veroorzaken. Elke activiteit die niet direct bijdraagt aan waarde voor de eindgebruiker wordt gezien als verspilling.

Het hoofdproduct van Offspark B.V. is de open source *software library* PolarSSL. Deze library staat centraal binnen deze afstudeeropdracht.

PolarSSL

PolarSSL biedt functionaliteit aan software ontwikkelaars aan om hun programma's veilig online te laten communiceren. De hoofdfunctionaliteit is de mogelijkheid om *Secure Socket Layer (SSL)* verbindingen op te zetten.

Secure Socket Layer (SSL) is een protocol waarmee er tussen twee internetclients een beveiligde verbinding opgezet kan worden.

PolarSSL is geschreven in de programmeertaal C en kan door C programmeurs geïntegreerd worden. Zo kunnen programmeurs in hun eigen programma's gebruik maken van de functionaliteit die PolarSSL beschikbaar maakt.

Er zijn ook andere libraries die deze functionaliteit aanbieden. Het populairste alternatief is de *OpenSSL library*. Ten opzichte van OpenSSL onderscheid PolarSSL zich op twee vlakken. Ten eerste is PolarSSL modulair opgebouwd waardoor programmeurs niet de hele library in hun eigen programma's hoeven te mee te leveren, maar ook specifieke delen kunnen kiezen. Op embedded systemen is dit een voordeel want daar is een beperkte geheugenruimte beschikbaar. De mogelijkheid om een bepaalde set modules uit PolarSSL te kiezen verlaagd het geheugengebruik ten opzichte van het integreren van de hele library. Het tweede vlak waar PolarSSL zich op onderscheid is dat de documentatie, indeling en functienamen van de bibliotheek erg leesbaar zijn ingedeeld.

Naast het opzetten van SSL verbindingen biedt de library functionaliteit om data te *encrypten* en te *decrypten*, keys te genereren en certificaten te verifiëren.

---

[1] http://en.wikipedia.org/wiki/Lean_manufacturing

[2] http://en.wikipedia.org/wiki/Toyota_Production_System

Enkele toepassingen en organisaties die PolarSSL gebruiken zijn:

- Regel- en besturingssystemen van liften
- De Nederlandse overheid
- OpenVPN
- Linksys routers

PolarSSL wordt middels een open source licentie vrijgegeven en middels een commerciële licentie verkocht. Met de open source licentie kan iedereen die bibliotheek wilt integreren in zijn eigen software dit doen, mits deze software ook onder een open source licentie wordt vrijgegeven.

Voor klanten die PolarSSL in commerciële software willen integreren, verkoopt Offspark B.V. commerciële licenties met extra ondersteuning. Met deze licentievorm hoeft de partij die de library in de eigen software integreert zijn code niet vrij te geven en kan deze de eigen software op elke manier doorverkopen.

De programmeertaal Ruby, Ruby on Rails en Ruby gems

Ruby is een object-georiënteerde programmeertaal die in 1993 bedacht is door de Japanner Yukihiro Matsumoto. De motivatie van Matsumoto om Ruby te ontwikkelen is dat hij vond dat andere programmeertalen niet bijdragen aan het gemak en plezier waarmee programmeurs hun werk doen. Hij stelde dat de programmeertalen die in 1993 bestonden niet genoeg gemak aan de programmeur gaven om goed zijn werk te doen.
Kenmerkend aan Ruby is dat de leesbaarheid van de code voorop staat en zo veel mogelijk de reguliere spreektaal representeert. Hierdoor zijn Ruby programma's over het algemeen ook goed te lezen voor niet-programmeurs.

De populariteit van de taal Ruby explodeerde toen de Deen David Heinemeier Hansson het webontwikkelframework *Ruby on Rails* uitbracht. Dit framework voert het principe van *convention over configuration* door. Dit principe komt tot uiting in de manier hoe Ruby on Rails voorschrijft om volgens een standaard methode te werken. Met deze methode kunnen programmeurs veel sneller webapplicaties ontwikkelen ten opzichte van andere bestaande frameworks zoals .NET MVC.

Veel Ruby programmeurs zijn na deze groei van Ruby plugins ontwikkelen die andere Ruby programmeurs in hun eigen programma's kunnen integreren. Deze plugins worden verpakt als *Ruby gems* en dit formaat is de standaard manier om binnen de Ruby programmeertaal libraries en plugins te verspreiden en installeren.

Voorbeeldcase Intercity

Tijdens deze afstudeeropdracht is een webapplicatie ontwikkeld om aan te tonen dat het in deze opdracht ontwikkelde product *PolarSSL for Ruby* geïntegreerd kan worden in andere Ruby programma's.

Intercity is een webapplicatie bedoeld voor bedrijven en freelance ontwikkelaars die het *Ruby on Rails framework* gebruiken om webapplicaties te bouwen. Met *Intercity* kunnen zij hun eigen internetservers inrichten zodat er Ruby on Rails applicaties op gehost kunnen worden. Op deze manier hebben zij geen kennis van serverconfiguratie- of beheer nodig en hoeven ze hier geen extra personeel voor in te huren.

De gewenste werking is dat een Ruby ontwikkelaar een server toe kan voegen in zijn *Intercity* account. Deze server kan gehuurd zijn bij een hostingpartij of de ontwikkelaar kan eigen hardware hebben die in verbinding staat met het internet.

De ontwikkelaar kan *Intercity* instrueren om de server op afstand te installeren en te configureren en alle benodigde software te installeren om Ruby on Rails applicaties op te hosten. *Intercity* voert deze taken via een beveiligde verbinding op een server uit.

De belangrijkste stap is het toevoegen van Ruby on Rails applicaties aan de server. De ontwikkelaar voert de informatie over de applicatie in de webapplicatie in, waarna er op zijn server databases en processen worden gestart zodat de Ruby on Rails applicatie *deployed* kan worden. Het *deployen* van een webapplicatie houdt in dat de programmacode uit een versiebeheersysteem gedownload wordt, de database voorbereid wordt en op de server de programmaprocessen starten zodat de Ruby on Rails applicatie te benaderen via het internet. Om de applicatie te kunnen deployen dient de ontwikkelaar eerst zichzelf en andere ontwikkelaars toegang te geven tot de server als *deployer*.

Situatie bij aanvang afstuderen

Bij aanvang van mijn afstuderen werd PolarSSL actief doorontwikkeld door een klein team onder leiding van Paul Bakker. Binnen Offspark B.V. is er een roadmap voor functionaliteit die nog ontwikkeld moet worden. Daarnaast wordt er actief naar feedback van klanten geluisterd om bugs of nieuwe functionaliteit te implementeren. Het volgende doel voor de library is het uitbrengen van versie 1.1.3 waarin er nieuwe functionaliteit wordt toegevoegd en de interne structuur van de library wordt verbeterd.

Daarnaast is er een project bezig om PolarSSL in contact te brengen met meer ontwikkelaars. Met het huidige prijsmodel is PolarSSL vooral voor grotere bedrijven interessant. Offspark is nieuwe manieren aan het proberen om het licentiemodel van PolarSSL interessanter te maken voor kleinere afnemers. Eén van deze manieren is om de commerciële licenties met een maandelijks abbonnementsmodel te gaan verkopen in plaats van een eenmalig contract, of een jaarlijkse overeenkomst. Het idee is dat de financiële drempel voor kleinere ontwikkelaars daardoor lager komt te liggen.

Verder is er een wens om PolarSSL in meerdere toepassingen te kunnen inzetten. Zo is er ooit een project geweest om de bibliotheek ook in iPhone en iPad applicaties te kunnen inzetten. Of PolarSSL nog steeds werkt met de meest recente versies van de iPhone en iPad software, iOS is onbekend.

Rol binnen de organisatie

Mijn rol als afstudeerder is geweest om als zelfstandige ontwikkelaar te werken aan de afstudeeropdracht. Het probleem en de doelstelling vielen binnen deze afstudeeropdracht en hadden geen directe afhankelijkheden met de andere projecten binnen Offspark B.V.

Ik heb de producten die ik op ging leveren onder persoonlijke titel in de open source gemeenschap uitgegeven zonder dat hier expliciete controle van code of ontwerp voor nodig was. Verder heb ik feedback kunnen vragen op de momenten dat dit nodig was.

2. Afstudeeropdracht

Probleemstelling

Het bedrijf Offspark B.V. levert de library *PolarSSL*. Deze library bevat modules die programmeurs kunnen aanroepen in hun eigen programma's zodat zij beveiligde verbindingen kunnen opzetten en data kunnen *encrypten*. Offspark is bezig met verschillende projecten om PolarSSL bij een grotere groep ontwikkelaars onder de aandacht te brengen.

Bij aanvang van deze afstudeeropdracht kon de PolarSSL library enkel gebruikt worden door softwareontwikkelaars die programma's ontwikkelen met de programmeertaal C. Om een grotere groep ontwikkelaars aan te spreken is er de wens om de library beschikbaar te maken voor ontwikkelaars die in de programmeertaal Ruby werken. Het probleem is dat het niet mogelijk is een library gemaakt voor de taal C vanuit een Ruby programma aan te spreken.
Het idee is dat het beschikbaar maken van de PolarSSL library voor de taal Ruby ervoor zorgt dat een grotere groep ontwikkelaars wordt aangesproken.

Doelstelling en uitgangspunten

Om dit probleem op te lossen wordt er binnen deze opdracht een zogenaamde *wrapper* ontwikkeld waarmee de functionaliteit uit de *PolarSSL library* beschikbaar gemaakt wordt binnen een Ruby programma. Het doel is om aan het eind van deze afstudeerperiode enkele versies van deze *wrapper* uitgegeven te hebben en dat de werking aangetoond wordt met de integratie in een andere Ruby applicatie. De *wrapper* krijgt de naam *PolarSSL for Ruby*.

Er zijn meerdere technieken beschikbaar om een *wrapper* tussen een C library en een Ruby programma te ontwikkelen. Aan de hand van een vergelijking wordt een keuze gemaakt welke techniek het best geschikt is voor de ontwikkeling *PolarSSL for Ruby*.

Daarnaast wordt er een webapplicatie gemaakt die als voorbeeldcase voor de integratie van *PolarSSL for Ruby* dient. Deze webapplicatie krijgt de naam *Intercity*. Aan de hand van de functionaliteit die benodigd is voor deze webapplicatie zal bepaald worden welke functionaliteit uit *PolarSSL* gekozen wordt voor *PolarSSL for Ruby*. Daarnaast wordt met deze webapplicatie bewezen dat *PolarSSL for Ruby* geïntegreerd kan worden in een andere Ruby applicatie.

Het belangrijkste uitgangspunt is dat *PolarSSL for Ruby* wordt uitgebracht als *Ruby gem* zodat Ruby ontwikkelaars deze in hun eigen software kunnen integreren. De requirements worden gekozen met de opdrachtgever en aan de hand van de nog te ontwerpen webapplicatie *Intercity*. De wrapper *wordt* onder een open source licentie uitgebracht. De wens is om zo vroeg mogelijk een eerste versie uit te geven er feedback verzameld kan worden en de merknaam neergezet wordt.

Er wordt ook geprobeerd de *Ruby wrapper* geschikt te maken voor de iOS ontwikkeltool *RubyMotion*.

Invulling beroepstaken

In deze afstudeeropdracht wordt de competentie voor drie beroepstaken bewezen.

De beroepstaak **Selecteren van standaardsoftware** is ingevuld met het selecteren van een techniek waarmee een koppeling gelegd kan worden tussen *PolarSSL* en de programmeertaal *Ruby*. Met een vergelijking aan de hand van enkele criteria wordt er een keuze gemaakt.

De beroepstaak **Ontwerpen systeemdeel applicatie** is ingevuld middels het komen tot een applicatie-ontwerp voor de ontwikkelde features voor de *PolarSSL for Ruby wrapper* en de voorbeeldcase *Intercity*.

De beroepstaak **Bouwen applicatie** wordt ingevuld met het implementeren van de code voor de *wrapper* en het ontwikkelen van de voorbeeldcase *Intercity*. Tijdens de uitvoering wordt hier met behulp van versiebeheer, compilers, testtools en een softwareontwikkelmethodiek invulling aan gegeven.

3. Totstandkoming Plan van Aanpak

In dit hoofdstuk is beschreven hoe het Plan van Aanpak voor dit afstudeerproject tot stand is gekomen. Ik heb de keuze gemotiveerd voor de projectuitvoeringsmethode Kanban en de softwareontwikkelmethode eXtreme Programming. Daarnaast heb ik gemotiveerd op basis waarvan ik de requirements voor de *PolarSSL for Ruby wrapper* zou gaan bepalen. Tevens heb ik beschreven welke rol van de voorbeeldcase *Intercity* binnen dit afstudeerproject inneemt en hoe tot een globale uitvoering van activiteiten is gekomen.

Kiezen methode projectuitvoering

Voor de algehele projectaanpak heb ik gekozen voor de methode *Kanban*[3].

Om tot deze keuze te komen heb ik de volgende aspecten in overweging genomen:

1. Offspark is een klein bedrijf zonder hiërarchie of vaste procedures.
2. Offspark volgt het Lean principe: waste minimaliseren en resultaat vergroten en software zo vroeg mogelijk naar gebruikers en klanten uitbrengen.
3. Ik voer de opdracht zelfstandig uit.
4. De functionaliteit en werking van de op te leveren producten zijn afhankelijk van elkaar.
5. Er is geen uitgewerkte lijst met requirements voor de te ontwikkelen software.
6. De afstudeeropdracht moet een (commerciële) versterking zijn voor het hoofdproduct.
7. Ik heb geen ervaring met de programmeertaal C of cryptografie.
8. Andere projecten en activiteiten buiten deze opdracht zijn niet afhankelijk van het resultaat van deze afstudeeropdracht.

Binnen de bedrijfscultuur Offspark leeft de *Lean*-filosofie. Het idee van *Lean* is dat een organisatie snel hoge kwaliteit resultaat kan behalen door het op te leveren werk op te delen die los van elkaar direct klantwaarde opleveren.

De gekozen methode *Kanban* bevat een werkwijze waarin features één voor één afgerond worden in plaats van meerdere features in een grote release. Hiermee vult de methode de behoefte om op een *Lean* manier de taken uit deze afstudeeropdracht uit te voeren en kon ik het best aansluiten bij de bedrijfscultuur van Offspark B.V.

Een watervalmethode had niet binnen deze afstudeeropdracht gepast. Bij waterval zouden namelijk eerst alle features in zijn geheel worden ontworpen voordat ze worden gedocumenteerd en vrijgegeven. Het vooraf opstellen van het totale ontwerp is strijdig met het principe van *Lean* waarin wordt gesteld dat elke set functionaliteit afzonderlijk als klein deel werk door alle fasen van ontwikkeling gaat.

Een principe van *Kanban* is ***Agree to pursue incremental, volutionairy change***. Dit principe heeft me in staat gesteld te beginnen met het ontwerpen, ontwikkelen, documenteren en uitgeven van een eerste feature voor ik aan de andere features hoefde te beginnen. Eventuele bevindingen of veranderingen die ik zou ontdekken kon ik incrementeel doorvoeren in de volgende functionaliteit binnen de opdracht.

---

[3] http://nl.wikipedia.org/wiki/Kanban

Omdat het doel van de *Lean*-filosofie is om zo veel mogelijk directe klantwaarde op te leveren zonder resources te verspillen moet de projectmethode een controlemechanisme bevatten die waarschuwt als er te veel resources verspilt wordt aan activiteiten die geen directe klantwaarde opleveren. *Kanban* biedt een dergelijk mechanisme.

Dit mechanisme is gebaseerd op *The Theory of constraints*[4]. Het mechanisme zorgt ervoor dat je beperkingen oplegt in de hoeveelheid werk dat in elke ontwikkelfase uitgevoerd mag worden. In het theoretisch kader op de volgende pagina wordt uitgelegd hoe de *Theory of constraints* binnen *Kanban* toegepast is.

In mijn Plan van Aanpak heb ik deze beperkingen als volgt ingedeeld op mijn *Kanban board*. Een *Kanban board* wordt gebruikt om een overzicht te houden op de features die nog gedaan moet worden in de *Backlog* en wat de status is van de features waar nu aan gewerkt wordt.

Backlog	Prototype / Design (2)	Testing / Development (1)	Documentation (1)	Releasing (1)	Done

De nummers in de kolommen geven aan hoeveel features in die fase van ontwikkeling aanwezig mogen zijn. Ik heb ervoor gekozen om elke fase na *prototype / design* maximaal één feature te laten bevatten zodat deze zo spoedig mogelijk *released* kon worden. Voor *prototype / design* fase heb ik een maximum van 2 features opgelegd. Op die manier kon ik alvast met een prototype of een vooronderzoek voor een nieuwe feature beginnen als bleek dat ik even vast zou zitten met een feature in één van de andere stadia.

Er was een risico aan het gebruik van *Kanban*. Dat risico is dat het ontwerp en de implementatie van de losse features niet goed op elkaar afgestemd konden worden. Door het ontwikkelen met losse features is er geen totaaloverzicht op het softwareontwerp. Daardoor kan het voorkomen dat beslissingen bij een eerder ontwikkelde feature niet aansluiten bij de implementatie in een latere feature.

Dit wordt door *Kanban* ondervangen door kleine features op te leveren. Kleine veranderingen zijn namelijk beter te behappen en je kan specifiek terugzoeken waar een probleem is ontstaan, ten opzichte van een groot ontwerp in één keer beoordelen.

De andere oplossing die ik voor dit probleem gekozen heb is te werken volgens de softwareontwikkelmethodiek *eXtreme Programming (XP)*. Het proces van XP bevat activiteiten die ervoor zorgen dat de ontworpen software op elk moment uitbreidbaar is. In de volgende paragraaf beschrijf ik deze activiteiten.

---

[4] http://en.wikipedia.org/wiki/Theory_of_constraints

Werking Kanban met theory of constraints

Backlog	Prototype (2)	Development (1)	Done
Feature: Server aanmaken	Feature: SSL connectie opzetten	Testomgeving opzetten	
Feature: Applicatie aanmaken	Feature: Encryptie		
Feature: Server configureren			

In deze is te zien dat in het stadium **Prototype** maximaal 2 taken tegelijkertijd uitgevoerd kunnen worden. In **Development** is dat er één. Dit houdt in dat er nooit meer dan één taak **in development** mag zijn. Taken worden van links naar rechts in de verschillende stadia opgeleverd en doorgegeven aan het volgende stadia. Een taak mag pas doorgaan naar een ander stadia, als daar ruimte voor is binnen het volgende stadium.

Als er één taak **in development** is en er zijn **twee** taken voltooid in de **prototype** fase, betekent dat daarom dat het hele ontwikkeltraject stopt. Namelijk: Er mag geen nieuwe taak van de **backlog** naar het stadium **prototype** omdat daar al twee taken in staan te “wachten”. Deze mogen op hun beurt niet naar het stadium **development** omdat daar nog een taak in ontwikkeling is. Pas als de taak in **development** doorschuift naar **done** kan er weer een nieuwe taak toegevoegd worden aan het proces.

Door deze werkwijze wordt direct opgemerkt als er een bottleneck ontstaat binnen het project want het project staat in feite stil. Deze herkenning heeft twee voordelen.

Het eerste voordeel is probleemherkenning. Als er een bottleneck herkent wordt kan op dat moment bekeken worden wat het probleem is dat de taak **in development** blijft staan. Dan kan er gekeken worden naar een oplossing.

Het tweede voordeel is *waste reducing*. Dat komt omdat alle aandacht van het team verlegd wordt naar de plek van de bottleneck, waardoor deze sneller opgelost wordt. Kan de oplossing niet gevonden worden, dan wordt de keuze gemaakt om de functionaliteit of het ontwerp daarvan te versimpelen.

Het idee is dat door de *constraint* van deze maximale chunks werk, er creatiever en doelgerichter door het team wordt gewerkt om de taak af te maken, zodat er weer een nieuwe taak door kan gaan naar een volgend stadium.

Kiezen softwareontiwkkelmethodiek

In de vorige paragraaf heb ik beschreven dat ik *Kanban* heb gekozen heb als projectmanagementmethode. Om invulling te geven aan de verschillende fasen van mijn *Kanban board* heb ik de softwareontwikkelmethodiek eXtreme Programming[5] gekozen.

Om tot deze keuze te komen zijn volgende aspecten van invloed geweest:

1. De methodiek is waste reducing, erop gericht om met zo min mogelijk energie een volledige functionaliteit op te leveren.
2. Geen uitgebreid up-front design zodat precies genoeg vastgelegd werd om de feature op te leveren.
3. Faciliteert het opleveren van losse features en niet het opleveren in een heel systeem in één keer.
4. Resulteert in features die productieklaar (getest, gedocumenteerd) zijn.
5. Resulteert in features die uitbreidbaar blijven binnen het toekomstige geheel.

eXtreme Programming (XP) beschrijft een proces waarbij user stories van een backlog vertaald worden naar kleine releases. XP sluit daarom aan bij de *Kanban* werkwijze om de software in kleine delen volledig op te leveren en direct klantwaarde te leveren.

De regel binnen XP is om het software-ontwerp zo klein en simpel mogelijk te houden zodat er weinig investering vooraf nodig is voor begonnen kan worden met de ontwikkeling van een feature,

Omdat er geen uitgebreid ontwerp vooraf wordt gemaakt moet er gelet worden op de uitbreidbaarheid van het ontwerp van de losse onderdelen. Als dit niet wordt gedaan kan het zijn dat er aan het begin van het project keuzes gemaakt worden die botsen met het ontwerp van features die later ontworpen worden. XP biedt hiervoor de technieken *Test Driven Development* en *refactoring* aan. Deze twee technieken zorgen ervoor dat een ontwerp blijft functioneren en altijd zo uitbreidbaar mogelijk gemaakt wordt.

Eventuele alternatieve softwareontwikkelmethoden die ik toe had kunnen passen zijn SCRUM en waterval / RUP.

SCRUM is echter een methode die niets zegt over de technieken om goede software te ontwikkelen zoals XP dat doet met *TDD* en *refactoring*. SCRUM is bedoeld om een vooraf vastgestelde set aan requirements in behapbare blokken binnen een team op te leveren en dat zo effectief mogelijk te doen. Omdat ik deze opdracht alleen uitgevoerd heb zijn de processen die SCRUM waardevol maken niet op mij van toepassing.

RUP wordt ook incrementeel uitgevoerd, maar heeft net als waterval een nadruk op het vooraf uitdenken van de requirements en het ontwerp van het systeem. Waterval en RUP past daarom niet bij het karakter van *Lean* waarmee zo spoedig mogelijk directe klantwaarde opgeleverd wordt en dit met zo min mogelijk energieverspilling wordt gedaan.

---

[5] http://www.extremeprogramming.org

Totstandkoming fasering

Het belangrijkste uitgangspunt van de opdracht was om zo snel mogelijk naar een eerste *release* van *PolarSSL for Ruby* toe te werken.

Om dit te bereiken heb ik eerst vastgesteld dat ik moest bepalen welke requirement er in de eerste versie van *PolarSSL for Ruby wrapper* ontwikkeld moest worden. Vervolgens heb ik bepaald dat ik een *wrapping techniek* moest vinden waarmee ik de *wrapper* kon ontwikkelen. Met deze twee onderdelen kon ik vervolgens beginnen aan de ontwikkeling van de eerste feature volgens het vastgestelde *Kanban* en *eXtreme Programming* proces.
Met de zo vroeg mogelijk uitgave van de eerste feature kon ik het eerste uitgangspunt van de afstudeeropdracht behalen.

Ik heb besloten om na de eerste feature te beginnen met de ontwikkeling van de voorbeeldcase *Intercity*. Uit de oplevering van *Intercity* kon ik namelijk bepalen wat de volgende requirements gingen worden voor *PolarSSL for Ruby*. Als laatste kon ik de ontwikkelde functionaliteit in *PolarSSL* dan integreren in de voorbeeldcase *Intercity* om aan te tonen dat de *Ruby wrapper* gebruikt werd in een externe applicatie.

Ik had er ook voor kunnen kiezen om eerst de voorbeeldcase *Intercity* uit te werken voordat ik begon met het selecteren van de *wrapping techniek* en het ontwikkelen van de eerste feature. Hiermee zou ik de eerste uitgave van *PolarSSL for Ruby* echter aanzienlijk vertragen zonder dat daar een reden voor was. Want, de eerste feature kon ik door mijn opdrachtgever laten bepalen.

Met deze redenatie kwam de globale fasering neer op:

1. Vaststellen requirement eerste feature
2. Selecteren techniek om een C library beschikbaar te stellen aan Ruby developers.
3. Uitbrengen eerste feature vanuit opdrachtgever.
4. Ontwerp en ontwikkelen *Intercity* voor volgende feature *PolarSSL for Ruby*.
5. Uitbrengen feature in *PolarSSL for Ruby*.
6. Integreren *PolarSSL for Ruby* in voorbeeldcase *Intercity*.

Vaststellen werkwijze ontwikkeling feature

Om conform *Kanban* en *eXtreme Progamming* te werken heb ik gekozen om de ontwikkeling van een feature in een aantal stappen te doorlopen. Met deze stappen heb ik de activiteiten binnen eXtreme Programming vertaald naar een werkwijze voor mijn afstudeeropdracht.

1. Prototype / design
2. Development / Testing
3. Documentation
4. Releasing

Omdat *eXtreme Programming* voorschrijft precies genoeg uit te denken om aan de implementatie van een feature te beginnen heb ik de stap **prototype / design** toegevoegd. Binnen deze stap heb ik een *spike* uitgevoerd als dat nodig was. Dit heb ik gedaan zodat ik een bepaalde oplossingsrichting in de vorm van een prototype eerst kon ontdekken, voor ik definitief zou kiezen om hier het ontwerp en de uiteindelijke implementatie op te baseren.

Een **spike** is een activiteit binnen van eXtreme Programming. In een spike wordt zo snel mogelijk een prototype ontwikkeld om een oplossingsrichting te onderzoeken. Deze code wordt niet als geschikt beschouwd voor implementatiecode en wordt daarom ook niet Test Driven ontwikkeld.

De volgende stap die ik vastgesteld heb is **development / testing**. Dit is de stap uit *eXtreme Programming (XP)* waar de feature middels *Test Driven Development (TDD)* uitgewerkt wordt tot een werkend ontwerp met werkende code.

De stap **documentation** is toegevoegd om elke feature direct bruikbaar te maken voor Ruby programmeurs. In de documentatie kunnen andere ontwikkelaars lezen hoe zij de klassen en methoden die ontwikkeld zijn in een feature kunnen integreren in hun eigen programma's.

De laatste stap **releasing** heb ik toegevoegd om de feature uit te brengen op *GitHub.com*. Dit is de website waar de code beschikbaar wordt gesteld en waar de documentatie bekeken kan worden. Daarnaast wordt de *Ruby wrapper* beschikbaar gemaakt als *Ruby gem* op *Rubygems.org*, de website waar Ruby programmeurs plugins en libraries kunnen downloaden.

Deel 2 - Uitvoering opdracht

4. Selecteren techniek Ruby wrapper

Voor ik ben beginnen aan het ontwerp en de implementatie van de eerste feature van de *Ruby wrapper* moest ik een techniek vinden die de *PolarSSL C library* en de *Ruby* programmeertaal met elkaar kan verbinden. In dit hoofdstuk is beschreven ik hoe ik gekomen ben tot de keuze om de techniek *Ruby C extensions* te gebruiken om *PolarSSL for Ruby* te ontwikkelen.

Om tot een vergelijking tussen verschillende tools te komen heb ik geïntentariseerd welke technieken er beschikbaar zijn. Deze technieken heb ik vergeleken:

- Ruby-FFI
- Swig
- Ruby C extensions

Vaststellen criteria

Allereerst heb ik geïnventariseerd wat de features en eigenschappen van deze drie tools waren. Dit heb ik gedaan omdat ik überhaupt nog geen overzicht had van wat een *Ruby wrapper* allemaal moet kunnen en hoe deze in een Ruby programma geïntegreerd kan worden.

Uit deze inventarisatie kwam naar voren dat de technieken een *bridge* tussen een C-library en de Ruby programmeertaal beschikbaar stellen waarin er Ruby objecten en methoden gekoppeld worden aan functie-aanroepen in de C library. De code die ik ontwikkeld heb met behulp van één van deze tools, is de kern van de *wrapper* geworden.

Op basis van deze kleine voorinventarisatie heb ik de volgende criteria gebruikt voor het vergelijken van de drie technieken:

1. Kan de PolarSSL library eenvoudig gewrapt worden.
2. Werkt de techniek met zowel dynamic als static libraries.
3. Genereert de techniek onderhoudbare en controleerbare code.
4. Is er duidelijke documentatie beschikbaar.
5. Is de techniek actueel en wordt deze frequent onderhouden.

Allereerst was belangrijk dat de techniek voor *PolarSSL* gebruikt kon worden. Als de techniek op de een of andere manier niet samen kon werken met *PolarSSL* dan zou deze tool afvallen.

Als tweede punt was het belangrijk dat de wrapper met zowel dynamic als met static gecompilede libraries gebruikt kon worden. Ik kwam erachter dat PolarSSL standaard als static library gecompiled wordt als de source handmatig gecompiled wordt. Wordt PolarSSL vanuit een package manager of binnen een Linux distributie geïnstalleerd, dan wordt er vaak een dynamic library gecompiled. Om deze reden moest de *wrapper* beide opties ondersteunen.

Voor software die *PolarSSL* integreerd is het cruciaal dat de code op een veilige manier geïmplementeerd wordt. Dat betekent dat gegevens die in het geheugen van een programme bijgehouden worden op de juiste momenten leeggehaald wordt zodat een aanvaller geen mogelijkheid heeft om gevoelige data in te zien. Om deze reden was het belangrijk dat de techniek in de vergelijking onderhoudbare en controleerbare code genereert.

Heb criteria voor duidelijke documentatie was belangrijk want dat vergemakkelijkt de implementatie en zorgt ervoor dat er ontwikkelaars die de *wrapping code* na mij moeten onderhouden documentatie beschikbaar hebben om wijzigingen te doen.

Als laatste was het belangrijk dat de techniek actueel is en goed onderhouden wordt. Als dit zo is dan worden eventuele bugs waarschijnlijk sneller opgelost, wordt de wrapping techniek continu verbetert, en is er een community beschikbaar om vragen aan te stellen en om voorbeelden te vinden.

Vergelijking uitvoeren

Om een keuze te maken tussen de drie technieken heb ik elk criteria per techniek afzonderlijk beoordeeld en de bevindingen gecombineerd. Op basis van dit overzicht heb ik een conclusie getrokken.

Ruby-FFI

Op het eerste gezicht leek Ruby-FFI de beste optie. Op de pagina van Ruby-FFI op GitHub.com zag ik dat er frequent updates en releases werden gedaan. Dit gaf mij aan dat de tool in ieder geval ontwikkeld en onderhouden werd. Ook was er genoeg documentatie aanwezig die aangaf hoe je verschillende datatypes, structs en functies middels FFI kon aanspreken.

Een ander voordeel was dat je de code voor de *wrapper* in Ruby kan schrijven. Door de FFI library in je *wrapper* code te includen kon je via de FFI module direct functies uit de C-library uitvoeren zonder daar C-code voor te hoeven schrijven. Ook zou Ruby FFI dan het geheugenbeheer uitvoeren. Hiermee voldeed de Ruby-FFI aan het criteria om gemakkelijk de PolarSSL library te kunnen wrappen.

Een nadeel van Ruby-FFI was dat je niet kon beïnvloeden of controleren wat er in de internals gebeurd. Zo kon je niet verifiëren op welke manier FFI geheugen alloceert en of geheugen met gevoelige data na gebruik correct overschreven werd.

Toen ik op basis van de documentatie wilde beginnen aan mijn prototype kwam ik erachter dat er alleen dynamisch gecompileerde libraries gebruikt konden worden. Het was met Ruby-FFI niet mogelijk om een statische library aan te spreken.

SWIG

SWIG heeft een andere aanpak dan Ruby-FFI. Volgens de documentatie kan SWIG wrapping code genereren op basis van C of C++ libraries naar high level talen zoals Python, Perl, PHP en Ruby. Op de SWIG pagina op GitHub zag ik dat deze tool ook recente updates en releases had gehad.

De documentatie om met SWIG te beginnen was lastiger te vinden. Ik moest een aantal keren verschillende pagina's van de documentatie lezen om te begrijpen hoe het werkte. Ik zag wel dat er heel veel documentatie aanwezig was met veel opties om voor verschillende talen wrappers te genereren.

SWIG kon met zowel dynamische als statische libraries werken. Met SWIG definieer je namelijk in een speciale definitietaal de structuur van de library die gekoppeld moet worden. Op basis van deze definitie genereert SWIG C-code waarin gebruik gemaakt wordt van Ruby C extensions, de 3e tool uit deze vergelijking. Dit maakt SWIG een code-generator om *wrapper code* te genereren die uitgevoerd wordt middels de *Ruby C extension* techniek.
De code die gegenereerd wordt door SWIG valt daarom te controleren. Echter, wordt er op basis van je definitietaal elke keer nieuwe C code gegenereert. Als je hier vervolgens aanpassingen in wilt maken kun je de SWIG definitie niet uitbreiden want dit overschrijft de bestaande C *wrapping code*.

Ruby C extensions

Ik kwam erachter dat het schrijven van *C extensions* de standaard ondersteunde manier van de Ruby programmeertaal is om C libraries aan te spreken. Met *C extensions* wordt de *wrapper code* in C geschreven die interne Ruby functiecalls aanroept om aan de Ruby programmeertaal objecten, datatypes en klassen toe te voegen.

Het vinden van documentatie voor deze techniek was lastig. De Ruby broncode bevat wel een tutorial waarin staat welke C-functies een Ruby extension kan gebruiken om de taal Ruby uit te breiden met nieuwe objecten.
Ik heb daarnaast een aantal blog posts gevonden waarin de auteurs stap voor stap uitleggen hoe je een Ruby C extension schrijft.

In de documentatie en blogposts las ik dat de code van een *Ruby C extension* door de Ruby package management tool *RubyGems* automatisch gecompiled wordt bij installatie. Omdat dit compilen bij installatie gebeurt kon zowel met een statische gecompilede library als een dynamisch gecompilede *PolarSSL* library gewerkt worden.

Als je een Ruby C extension maakt dan schrijf je zelf de C code die de C library verbind met Ruby. Om deze reden zou ik met deze optie volledige controle hebben over hoe de *PolarSSL library* beschikbaar gemaakt zou worden aan de Ruby programmeur en zou ik de code voor de *wrapper* op mijn eigen manier in kunnen richten.

Conclusie

Uit de vergelijking van deze drie wrappingtechnieken heb ik gekozen om *Ruby C extensions* te gebruiken om *PolarSSL for Ruby* in te schrijven. Het resultaat van de vergelijking is weergegeven in de matrix op de volgende pagina.

	Eenvoudig wrappen	Dynamic en static	Controleerbare & onderhoudbare code	Duidelijke documentatie	Frequent onderhouden
Ruby-FFI	+	-	-	+	+
SWIG	+	+	+/-	+/-	+
Ruby C extensions	+/-	+	+	+/-	+

Met SWIG waren er geen technische problemen maar daarmee was het lastiger om de code die gegenereerd was te controleren en aan te passen. Als er problemen in de gegenereerde code gevonden worden dan zou het beheren van de aanpassingen hiervan moeilijk worden, omdat de wijzigingen bij een hergeneratie weer verloren zouden gaan.

Een klein nadeel van het gebruik van *Ruby C extensions* was dat de documentatie minder volledig is dan die van Ruby-FFI en SWIG. Daar stond wel tegenover dat RubyGems.org een officiële guide had om een Ruby C extension als RubyGem uit te brengen. Ruby-FFI en SWIG hadden deze documentatie niet.

De doorslaggevende eigenschappen om te kiezen voor de C extensions waren dat dit de officieel ondersteunde manier was vanuit Ruby zelf en dat de code van de *wrapper* volledig te controleren viel.

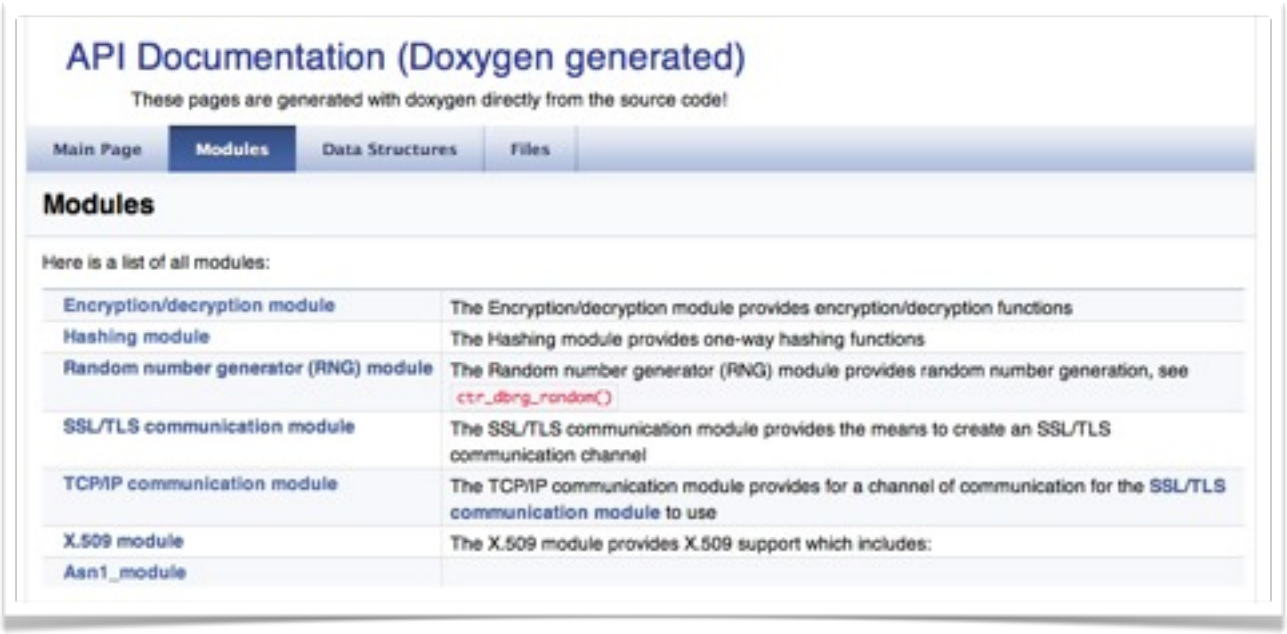
5. Ontwikkelen eerste feature wrapper

In dit heb ik beschreven hoe de eerste feature van de *PolarSSL for Ruby wrapper* is ontwikkeld. De nadruk van dit hoofdstuk ligt op de toepassing van het eXtreme Programming en de stappen die ik in mijn Plan van Aanpak gedefinieerd heb. Daarnaast heb ik geïllustreerd hoe de ontwikkelomgeving tot stand is gebracht.

Requirement vaststellen

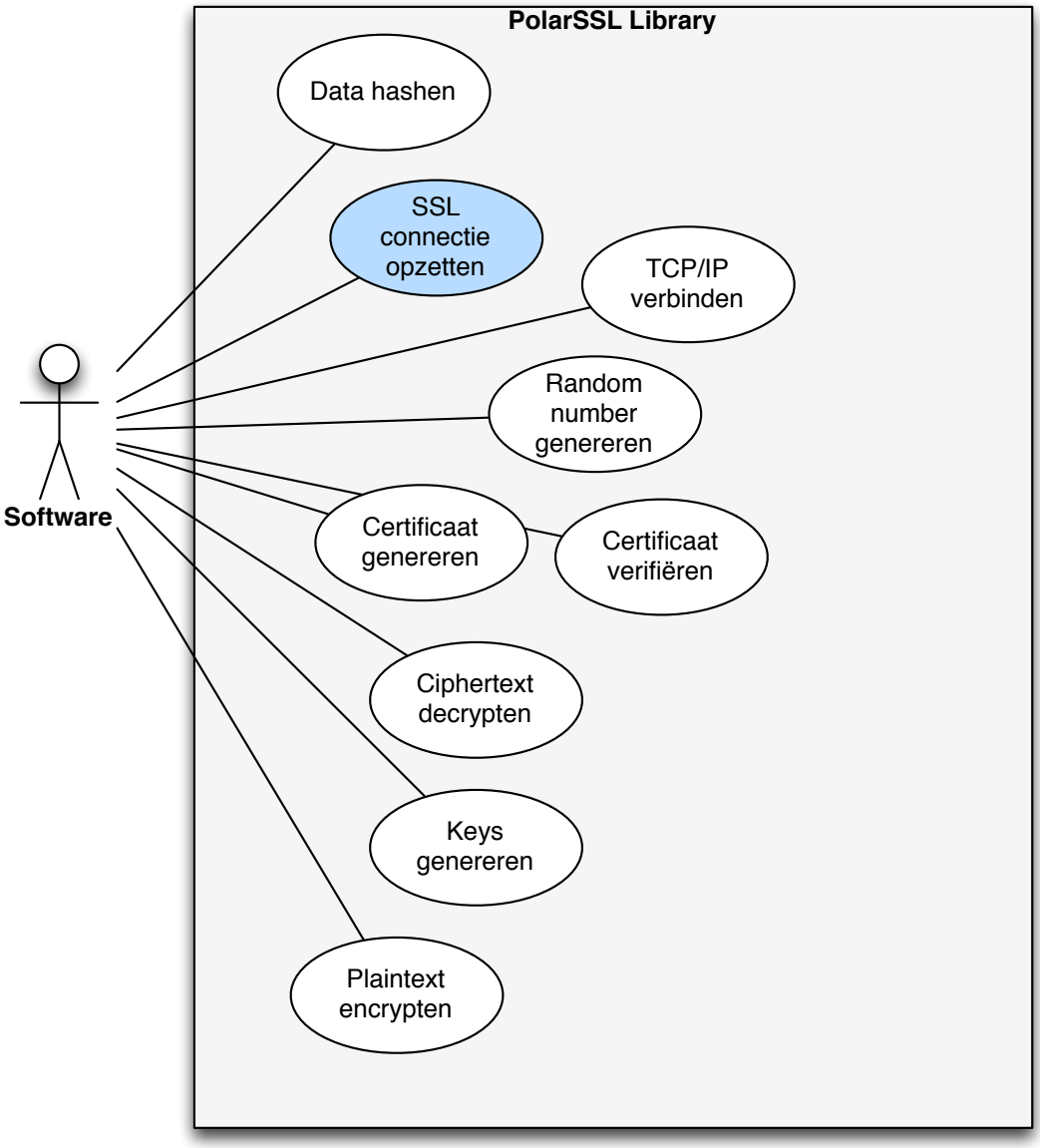
De feature die als eerst ontwikkeld is voor *PolarSSL for Ruby* is het opzetten van een *SSL connectie*. Om tot deze keuze te komen heb ik de huidige functionaliteit van de *PolarSSL library* in een overzicht van use cases vertaald en voorgelegd aan mijn opdrachtgever. Op deze manier heeft deze in één oogopslag kunnen bekijken welke use case de eerste prioriteit moest krijgen.

Om tot dit overzicht te komen heb ik een analyse gedaan op basis van de *PolarSSL library* documentatie. De documentatie bevat namelijk een overzicht van de functionele modules. De modules in de documentatie zien eruit als volgt:



Enkele van deze modules zijn onderverdeeld in submodules. Deze submodules heb ik als aparte use cases opgenomen. De *Encryption/decryption* module bevat zoals de naam suggereert functionaliteit om data te *encrypten* en te *decrypten*. Dit heb ik gedaan zodat de requirements van de individuele use cases zo klein mogelijk werden en dat daardoor de ontwikkeling van de eerste feature voor de *wrapper* zo snel mogelijk opgeleverd kon worden.

Het resultaat van de analyse van de modules uit de *PolarSSL library* documentatie heeft geleid tot de use cases die worden weergegeven in het overzicht op de volgende pagina.


PolarSSL Library
Data hashen
SSL connectie opzetten
TCP/IP verbinden
Random number genereren
Certificaat genereren
Certificaat verifiëren
Ciphertext decrypten
Keys genereren
Plaintext encrypten
Software

Prototype en design

Om tot een ontwerp te komen heb ik op basis van een voorbeeld uit de documentatie van *PolarSSL* een prototype gemaakt. Middels dit prototype heb ik een inventarisatie gemaakt van de datastructuren en functies die nodig zijn om in minimale vorm een *SSL connectie* op te zetten. Ik had het eerste ontwerp ook kunnen maken op basis van de *PolarSSL* documentatie. In de documenatie worden echter meer datastructuren en functies beschreven dan benodigd zijn voor het opzetten van de connectie. Als ik het ontwerp op basis van de documentatie had gemaakt dan had ik de feature groter ontworpen dan benodigd was. Door het programmeren van het prototype heb ik vastgesteld welke functies er minimaal aangeroepen moeten worden en heb ik het kleinst mogelijk ontwerp gebruikt voor het afronden van deze feature.

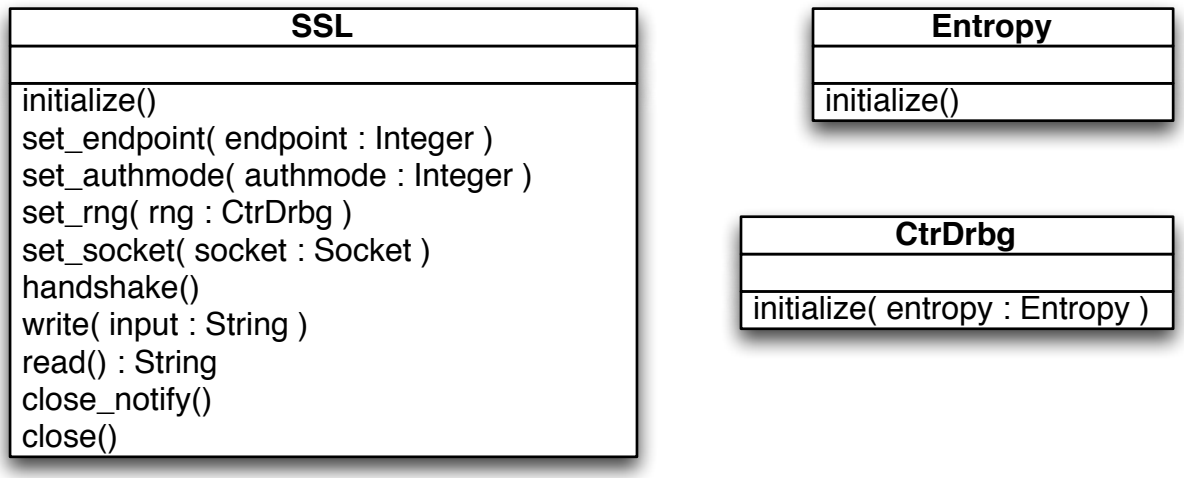
Vertalen procedurele interface PolarSSL naar een objectgeörienteerd ontwerp

Om tot een objectgeörienteerd ontwerp te komen heb ik de datastructuren en functies uit het prototype vertaald naar klassen en methoden voor in *PolarSSL for Ruby*. Omdat de programmeertaal C een procedurele programmeertaal is, was het niet direct evident welke klassen en methoden ik in dit objectgeörienteerde ontwerp op moest nemen. Ik heb daarom enkele richtlijnen vastgesteld in *Bijlage 4: API guidelines Ruby wrapper* waarmee de C implementatie van *PolarSSL* vertaald konden worden naar de *wrapper*. Deze richtlijnen zorgen ervoor dat de features van *PolarSSL for Ruby* op een consistente manier ontworpen worden en dat het voor toekomstige programmeurs duidelijk is hoe zij nieuwe features kunnen ontwikkelen.

In de volgende paragraaf geef ik weer hoe het klasse-ontwerp van deze feature is ontstaan aan de hand van één van deze richtlijnen.

Richtlijn voor klasse-definitie

Met de toepassing van de richtlijnen ben ik tot het volgende ontwerp gekomen voor de *SSL connectie feature* van de *Ruby wrapper*.



Ik ben tot deze klasse-indeling gekomen door te analyseren welke `struct` variabelen ik in mijn prototype nodig had om de *SSL connectie* op te zetten. In C bevat een variabele van het type `struct` namelijk een

aantal andere variabelen. Dit komt overeen met het concept van een klasse met attributen bij objectgeörienteerd ontwerp. Daarnaast merkte ik op dat dezelfde `struct` variabele mee wordt gegeven als parameter aan functies waarvan de naam met dezelfde prefix begon.

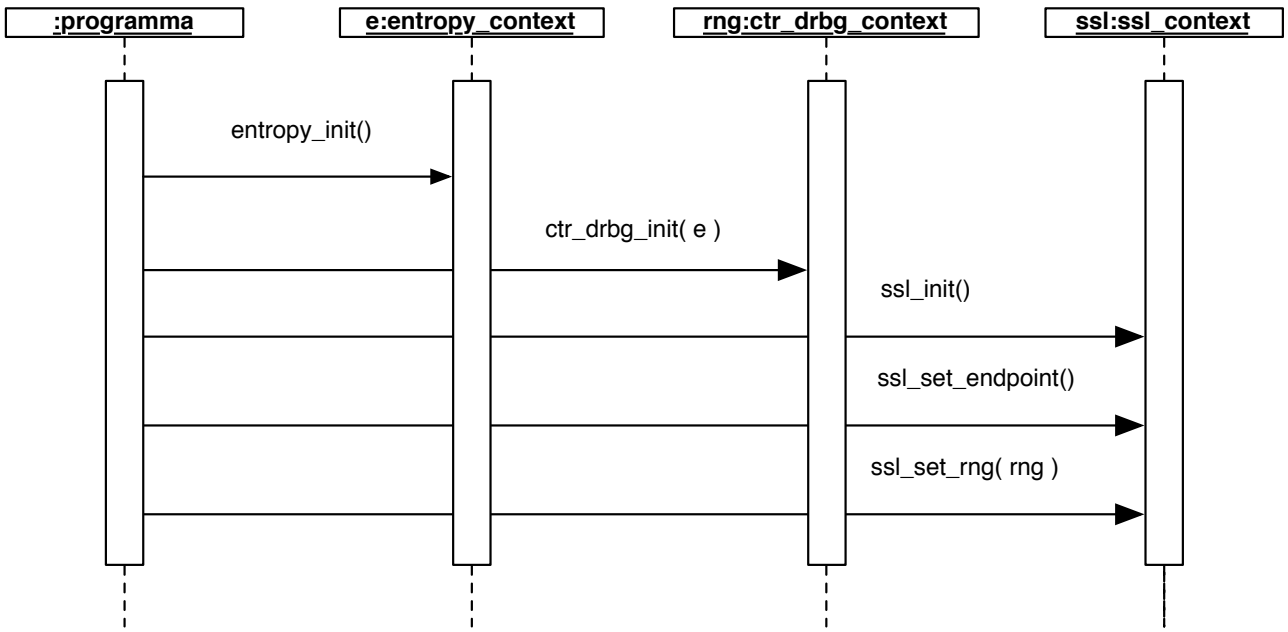
In de volgende tabel zijn een aantal van deze functies opgenomen om de overeenkomsten te illustreren. In deze tabel is in de meest rechter kolom te zien dat er functies zijn die ogenschijnlijk bij elkaar horen omdat zij met dezelfde prefix beginnen en de variabele met een type struct als eerste parameter mee krijgen.

struct type	variabele	functie-interface

`ssl_context`	`ssl`	`ssl_init( &ssl )`
		`ssl_set_endpoint( &ssl, ...)`
		`ssl_set_authmode( &ssl, ...)`

`ctr_drbg_context`	`ctr_drbg`	`ctr_drbg_init( &ctr_drbg, ...);` `ctr_drbg_random( &ctr_drbg, ...);`
`entropy_context`	`entropy`	`entropy_init( &entropy );` `entropy_gather( &entropy );`

Aan de hand van deze overeenkomsten heb ik een sequentiediagram getekend waarin de variabele van een type `struct` als object gemodelleerd is de functies als berichten tussen deze objecten. Het volgende diagram geeft een deel weer van het prototype-programma.



Met deze werkwijze heb ik de procedurele interface van *PolarSSL* library omgezet in een objectgeörienteerd ontwerp voor *PolarSSL for Ruby*.

In het klasse-ontwerp heb ik ervoor gekozen om de *setter* methoden zoals `set_endpoint()`, `set_authmode() en set_rng()` voor de implementatie niet te modelleren als attributen op de klasse. De

reden hiervoor is dat deze functies direct een *PolarSSL* functie uitvoeren en geen *instance variabele* op het Ruby object instellen.

Development en testing

Met het ontwerp uit de **prototype / design** fase van de feature had ik genoeg informatie om met de implementatie te beginnen. Voor ik aan de implementatie ben begonnen heb ik eerst een testomgeving opgezet conform de werkwijze volgens eXtreme Programming (XP). XP schrijft namelijk voor om een *unit test* te schrijven voordat er begonnen wordt met de implementatie van het ontwerp.

Gebruik MiniTest als testframework

Voor het opzetten van de testscripts heb ik het framework **MiniTest** gekozen. Er zijn ook andere testframeworks beschikbaar voor Ruby. Ik heb **MiniTest** gekozen omdat dit standaard meegeleverd wordt met Ruby. Hierdoor hoeven ontwikkelaars voor het gebruik van *PolarSSL for Ruby* geen extra dependencies te installeren. Een andere reden is dat **MiniTest** een kleine codebase heeft waardoor het uitvoeren van de testscripts sneller is dan met andere frameworks. Ik had er ook voor kunnen kiezen geen framework te gebruiken en zelf een testlibrary te schrijven. Dit heb ik niet gedaan omdat een testframework commando's bevat om automatisch alle testscripts uit te voeren en daarover te rapporteren. Zonder framework had ik deze functies zelf moeten implementeren.

Opzetten Git versiebeheer

Om de werking van het testframework te testen op de *continuous integration* service TravisCI.com heb ik na het bouwen van het eerste testscript een repository opgezet met de versiebeheertool *Git* en is de code geplaatst op de hosting website *GitHub.com*. *TravisCI* integreert namelijk met *Git* en voert bij elke codewijziging die gedaan wordt automatisch alle testscripts uit.

Continuous integration is het proces dat waarborgt dat code die geschreven is automatisch getest en geïntegreerd kan worden met andere code.

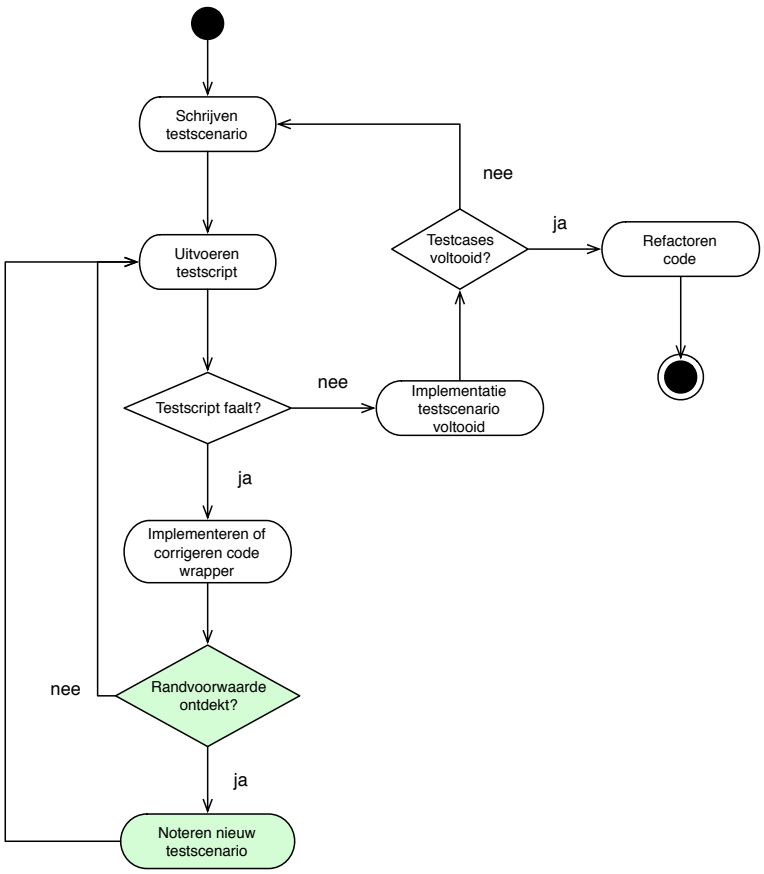
De continuous integration service **TravisCI.com** biedt open source projecten de gratis dienst om bij elke codewijziging alle testscripts uit te voeren en daarover te rapporteren.

Ik heb voor *Git* en *GitHub.com* gekozen omdat *PolarSSL*, de Ruby programmeertaal en andere Ruby-projecten ook van deze tools gebruik maken. Op deze manier is het gemakkelijker voor ontwikkelaars en gebruikers binnen de Ruby gemeenschap om met *PolarSSL for Ruby* aan de slag te gaan.

Test-driven implementeren

Nadat het testframework, de *continuous integration* service en het versiebeheer opgezet was heb ik de code van de *SSL connectie* feature aan de hand van een testscript geïmplementeerd. Ik heb ervoor gekozen om in dit tescript de gehele werking van de feature uit te schrijven. Op deze manier heb ik een implementatie-doel vastgelegd waar ik gestructureerd naartoe kon werken.

De werkwijze die ik gevolgd heb is weergegeven in de volgende flowchart:



Tijdens de implementatie van deze feature heb ik de twee groene processtappen toegevoegd aan de Test Driven Development werkwijze die ik gevolgd heb tijdens het gehel project. In het Plan van Aanpak zag de werkwijze er namelijk uit volgens de witte processtappen.

De toegevoegde processtappen garanderen dat ik tijdens de implementatie kijk of ik randvoorwaarden niet afgedekt heb. Het eerste testscript van deze feature gaat er namelijk vanuit dat er een succesvolle *SSL connection* opgezet kan worden met de website *https://nu.nl/*. Maar, het testscript bevat geen controle wat er gebeurt als er bijvoorbeeld geen internetverbinding is. Het resultaat is dat er geen foutafhandeling voor deze situatie plaats vind en dat het programma crasht. Dit allemaal omdat dit niet in een testscript voorkwam tijdens de implementatie. Door deze nieuwe processtappen te volgen heb ik voor het hele project gegarandeerd dat alle mogelijke randvoorwaarden afgedekt worden.

Documenteren

Na het implementeren van de testscripts en de code in de *wrapper* heb ik de documentatie opgezet. Dit heb ik gedaan door elke klasse en elke methode te voorzien van commentaar in de code. Dit commentaar heb ik vervolgens met de tool **rdoc** omgezet in een via een browser te bekijken *API documentatie*.

Een voordeel hiervan is dat ik de documentatie niet in een apart systeem hoef te beheren. Dat vergemakkelijkt het

De **API documentatie** van een library bevat een overzicht van beschikbare functies en hoe deze aangeroepen kunnen worden.

bijwerken van de documentatie. Een ander voordeel is dat de documentatie automatisch meegenomen wordt met de wijzigingen van de code in de versiebeheertool. Dat zorgt ervoor dat de ontwikkeling van de documentatie net als de code terug te lezen is in de historie en dat het bijvoorbeeld gemakkelijk is om documentatie te genereren voor een oude versie van de *wrapper*.

Releasen

Na het schrijven van de documentatie heb ik de feature released en daarmee het doel gehaald om *PolarSSL for Ruby* zo snel mogelijk aan de buitenwereld te tonen. Voor de nummering van de release heb ik gebruik gemaakt van de conventie van *Semantic Versioning*[6] *(Zie onderstaand kader)*. Deze conventie stelt een bepaalde structuur van het versienummer voor zodat te zien is welke impact een nieuwe release heeft voor de gebruiker.

Summary Semantic Versioning

Given a version number MAJOR.MINOR.PATCH, increment the:

MAJOR version when you make incompatible API changes,
MINOR version when you add functionality in a backwards-compatible manner, and
PATCH version when you make backwards-compatible bug fixes.

Additional labels for pre-release and build metadata are available as extensions to the MAJOR.MINOR.PATCH format.

Van: http://semver.org

[6] Semantic Versioning (http://semver.org)

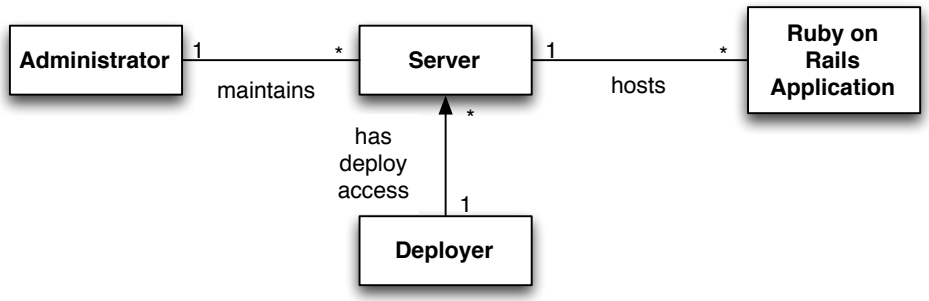
6. Ontwerpen voorbeeldcase Intercity

In dit hoofdstuk beschrijf ik hoe de webapplicatie *Intercity* is ontworpen. Deze webapplicatie is ontworpen en gebouwd om twee redenen. Allereerst om als input te dienen voor de op te stellen requirements voor *PolarSSL for Ruby*. Daarnaast wordt *Intercity* gebruikt om aan te tonen dat *PolarSSL for Ruby* daadwerkelijk in een andere Ruby applicatie geïntegreerd kan worden.

Allereerst beschrijf ik hoe vanuit een domeinmodel verschillende Use Cases opgesteld zijn en hoe deze in relatie met elkaar staan. Daarna leg ik uit hoe deze use cases en het domeinmodel vertaald zijn naar een applicatie-ontwerp voor één specifieke use case. Uit dit applicatie-ontwerp is gebleken welke functionaliteit er benodigd was om in *PolarSSL for Ruby* te ontwikkelen.

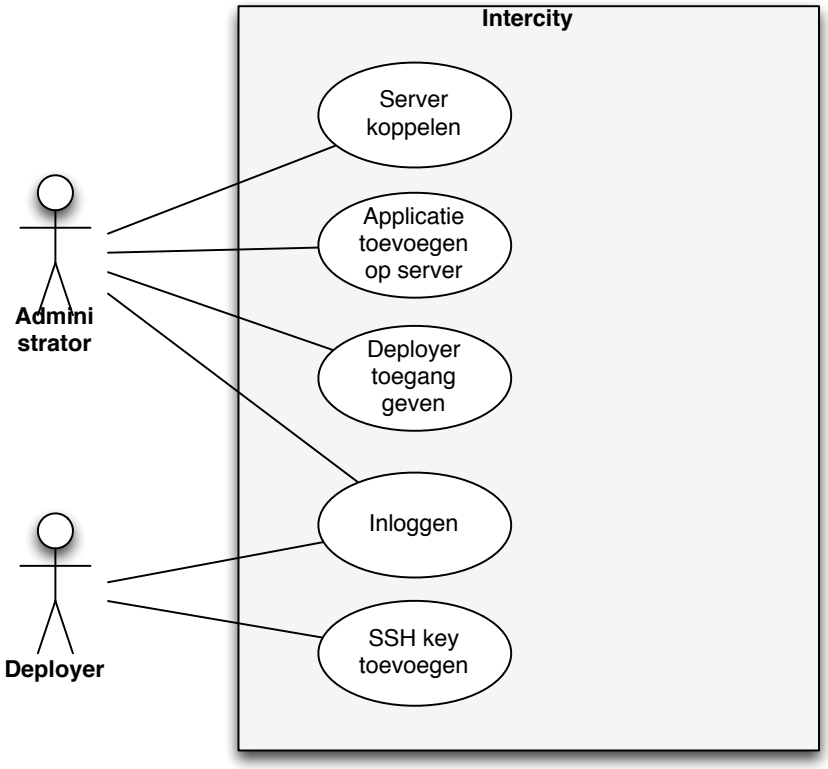
Vaststellen domeinmodel en use cases

Op basis van de gewenste functionaliteit heb ik het volgende initiële domeinmodel gemaakt. Dit model laat op abstract niveau zien welke rollen en objecten er voor het gebruik van *Intercity* relevant zijn.

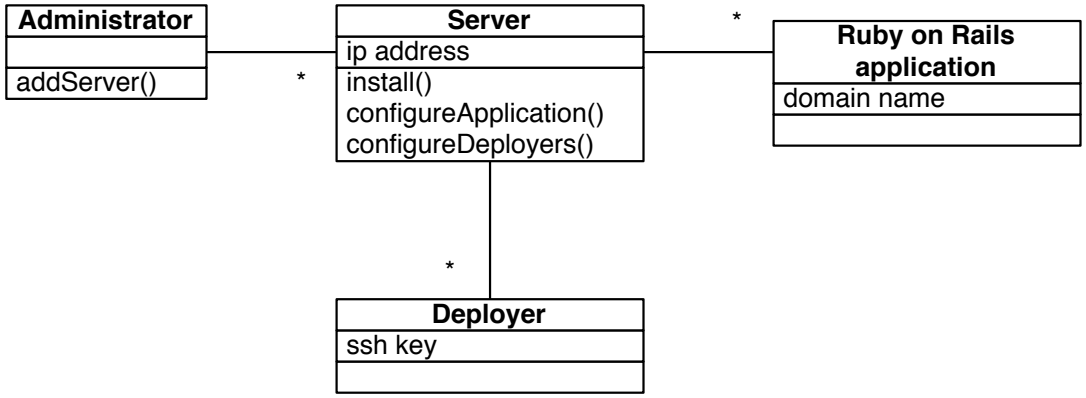


In het model is degene die de server toe voegt aan *Intercity* benoemd als *Administrator*. In een team van ontwikkelaars wordt dit namelijk de beheerder van de server. Een ontwikkelaar die van een *Administrator* toegang krijgen om applicaties op een server te deployen is *Deployer* genoemd.

Vervolgens heb ik geïnventariseerd welke acties de verschillende gebruikers van de applicatie uit zouden moeten kunnen voeren met de klassen uit dit initiële domeinmodel. Deze acties heb ik vertaald naar *use cases* in het onderstaande diagram. Dit heb ik gedaan zodat ik aan de hand van het uitschrijven van deze use cases attributen en verantwoordelijkheden kon ontdekken voor de klassen in het domeinmodel.



Op basis van de uitschreven use cases uit dit diagram heb ik het initiële domeinklassediagram uitgebreid met attributen en verantwoordelijkheden die benodigd waren voor het voltooien van de use cases.



In dit domeinmodel heb ik ervoor gekozen om de *Server* klasse de verantwoordelijkheid te geven om zichzelf te installeren, applicaties te configureren en deployers toegang te geven. De reden hiervoor is geweest dat het voor deze elk van deze verantwoordelijkheden nodig is om met de server te verbinden via het ip adres om vervolgens commando's op de server uit te voeren. Ik had er bijvoorbeeld ook voor kunnen kiezen de configureApplication() op de *Ruby on Rails application* klasse te modelleren. Dit heb ik niet gedaan omdat een Ruby on Rails applicatie in principe onafhankelijk van een bepaald type serverconfiguratie kan draaien en het de keuze aan een server is om de applicatie aan het internet beschikbaar te stellen. Als deze verantwoordelijkheid onderdeel zou zijn van de Ruby on Rails application klasse, dan zou deze klasse

te kennis bevatten hoe te verbinden met verschillende servers terwijl dat geen kerntaak is van een Ruby on Rails application.

Ontwerpen op basis van prototype

Voordat ik verder ging met het applicatieontwerp heb ik in een *spike* een werkend prototype ontwikkeld met de minimale functionaliteit om een Ruby on Rails applicatie op een server beschikbaar te maken. Dit heb ik gedaan omdat ik op dit punt niet precies wist welke tools en gegevens ik nodig had om een Ruby on Rails applicatie op een server te installeren. Daarnaast wilde ik kijken of ik de tot nu toe opgestelde functionaliteit kon versimpelen. Als ik de functionaliteit kon versimpelen kon ik wellicht de ontwerp- en implementatietijd verkorten waardoor ik *Intercity* sneller zou kunnen lanceren en eerder aan de integratie met *PolarSSL for Ruby* kon beginnen. Tijdens het ontwikkelen van het prototype zou ik dan langzamerhand mijn applicatie-ontwerp kunnen laten ontstaan.

Ik had ook kunnen kiezen niet te beginnen met het prototype en op basis van de informatie die ik op dat moment had het volledige applicatie-ontwerp eerst kunnen uitwerken. Dit zou resulteren in een ontwerp dat waarin ik veel aannames zou moeten doen die in de werkelijkheid anders uit zouden kunnen pakken. Dat zou me veel tijd kosten zonder er zeker van te kunnen zijn dat het een goed ontwerp werd. Daarnaast kon ik door de implementatie van het prototype wellicht komen tot creatievere oplossingen die ik door enkel abstract te modelleren niet had ontdekt.

Weglaten *Deployer* actor

Tijdens het maken van het prototype realiseerde ik me dat ik de eerder opgestelde use cases moest veranderen, waardoor er een minder groot applicatieontwerp nodig was.

Allereerst kwam ik erachter dat ik de actor *Deployer* en de bijbehorende use cases *Inloggen* en *SSH key toevoegen* kon weglaten. Omdat het prototype geen inlogfunctionaliteit had beschouwde ik de *Administrator* als enige gebruiker. Als *Administrator* moest ik handmatig een *SSH key* van een *Deployer* aan een server toevoegen. Ik bedacht me een voorbeeldsituatie waarin de persoon *Deployer* zijn *SSH key* ook via bijvoorbeeld email aan de *Administrator* kunnen overdragen waarna die hem dan aan de *Server* in de webapplicatie kon toevoegen. Omdat de *Administrator* in beide gevallen een handmatige handeling moest doen om een *Deployer* toegang te geven besloot ik dat de actor *Deployer* overbodig was geworden.

SSH keys worden gebruikt om personen of programma's toegang te geven tot een server.

Het weglaten van deze actor zou me tijd besparen omdat ik geen inlogfunctionaliteit voor de *Deployer* gebruikers meer hoefde te maken. Daarnaast hoefde ik de schermen voor het toevoegen en het goedkeuren van een *SSH keys* niet meer te ontwikkelen. Dit gaf me tijdswinst terwijl het doel van de originele functionaliteit nog steeds door de gebruikers bereikt kon worden.

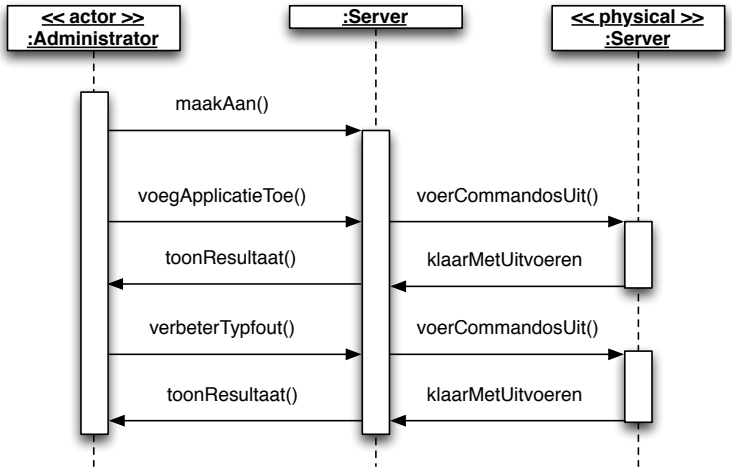
Herdefiniëren use cases *Applicatie toevoegen* en *SSH keys instellen*

Tijdens het maken van het prototype heb ik er ook voor gekozen om een extra use case *Serverwijzigingen toepassen* op te nemen. Deze use case wordt gebruikt door de use cases *Applicatie toevoegen aan server* en *SSH keys instellen* zoals weergegeven in het diagram hierna. De reden voor deze wijziging was dat er ik bij het prototypen van de gebruikersinteractie achter kwam dat er een scherm nodig was waar de gebruiker de serverwijzigingen kon controleren en toepassen. Fouten tijdens het configureren van de server konden dan op dit scherm teruggekoppeld worden zodat er een vervolgactie op ondernomen kon worden.

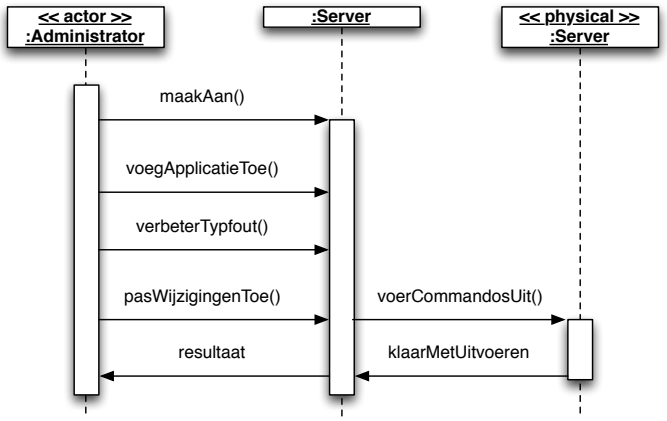
Als ik deze tussenstap niet hand ontworpen dan zou in de gebruikersinterface het toevoegen van applicatiegegevens of het toevoegen van een SSH key direct op de server toegepast worden. Als iemand een typfout zou maken bij het instellen van een applicatie en per ongeluk de wijzigingen op zou slaan dan zouden deze wijzigingen direct toegepast worden op de server. Het uitvoeren van commando's op de server

De initiële ontwerpsituatie en het uiteindelijke ontwerp is weergegeven in de volgende sequentiediagrammen:

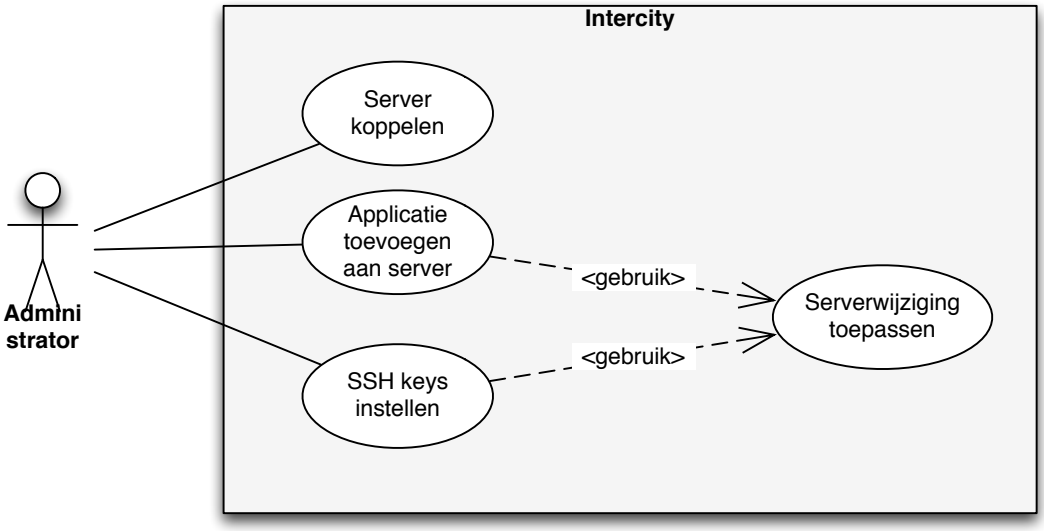
Initieel ontwerp



Uiteindelijk ontwerp



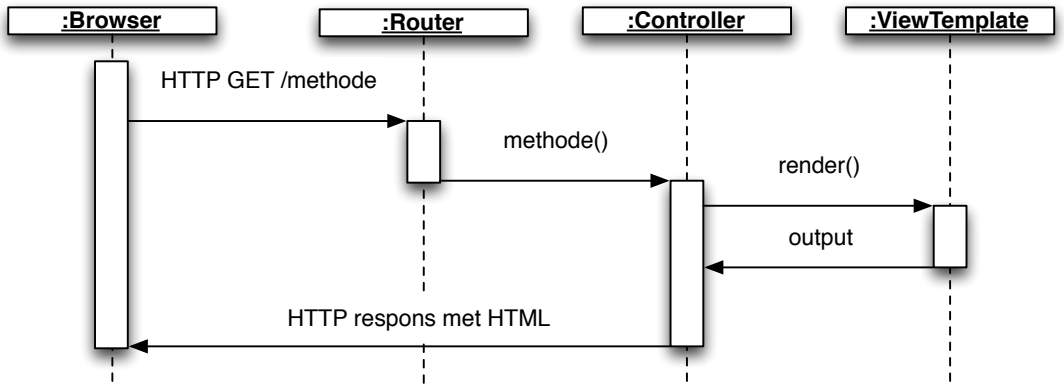
De use cases zijn op basis van deze inzichten veranderd in:



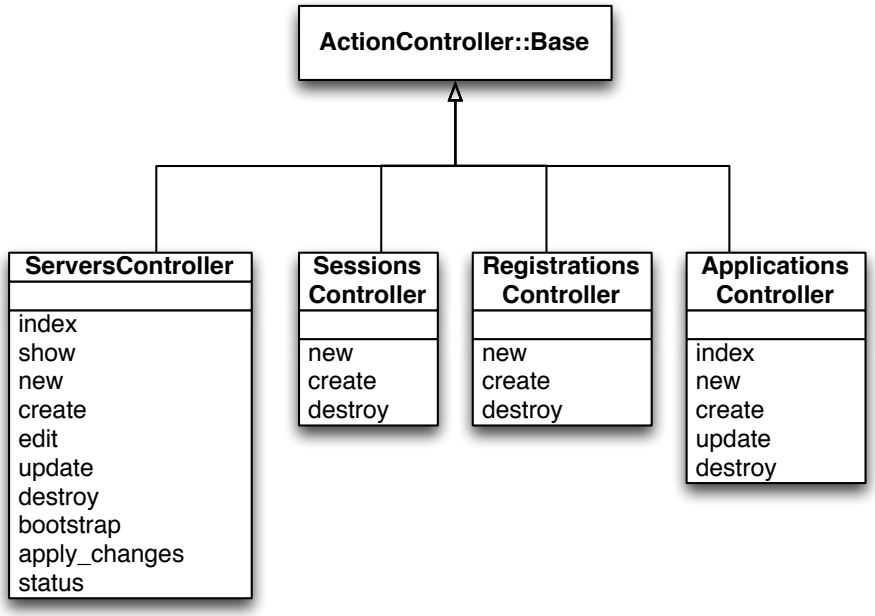
Ontwerpen applicatie met MVC

Intercity wordt met het Ruby on Rails webontwikkelframework ontwikkeld. Ruby on Rails is een framework dat gebruik maakt van het Model View Controller (MVC) design pattern. Op basis van het prototype heb ik een definitieve indeling van *models* en *controllers* gemaakt. Models en Controllers zijn in het framework klassen met methodes en attributen. De View-laag bestaat uit template-bestanden die door een Controller gebruikt worden om de interface van de webpagina op te bouwen.

In het volgende sequentiediagram is weergegeven hoe een verzoek vanuit de webbrowser van een gebruiker afgehandeld binnen het Ruby on Rails framework.



In het volgende klassediagram zijn de ontworpen controllers weergegeven:



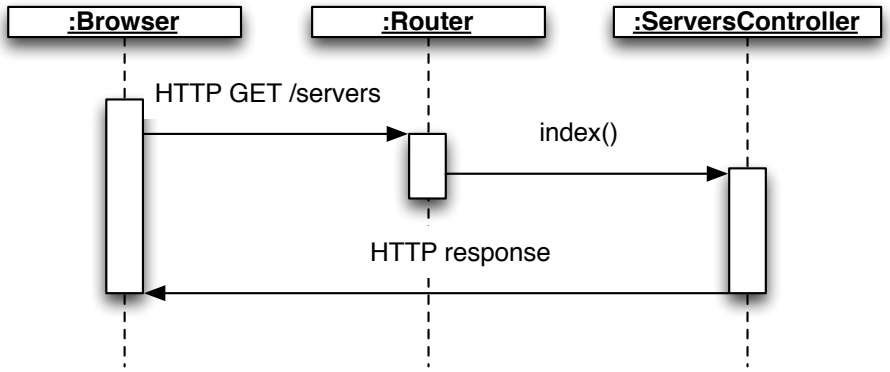
Voor het ontwerp van de methoden op de *controllers* heb ik gekozen om het CRUD pattern te volgen. Dit is te zien aan het feit dat de controllers een vaste set aan methoden hebben met dezelfde naamgeving zoals *new, create* en *destroy, index, show, edit* en *update*. De reden dat ik dit geïmplementeerd heb is dat ik hiermee gebruik kon maken van de ingebouwde functionaliteit van Ruby on Rails framework om zogenoemde *resources* beschikbaar te maken via een *RESTful* interface. Hierdoor herkent de applicatie automatisch welke controller methode uitgevoerd moet worden als er een HTTP verzoek van de webbrowser van de gebruiker binnen komt.

Bijvoorbeeld, als de applicatie een HTTP verzoek `GET /servers` binnen krijgt, bepaald de Router automatisch dat dit verzoek afgehandeld moet worden door de `index()` methode op de `ServersController`.

CRUD staat voor *Create, Read, Update, Delete* en is een interface patroon om op een uniforme manier data-manipulaties op objecten aan te brengen.

REST staat voor Representational State Transfer en is een protocol dat beschrijft hoe applicaties een uniforme data-manipulatielaag naar de buitenwereld kunnen aanbieden.

HTTP staat voor Hypertext Transfer Protocol en beschrijft een aantal commando's waarmee applicaties met elkaar kunnen communiceren. Browsers en websites zijn voorbeelden van HTTP applicaties.



Door dit patroon te volgen hoefde ik deze koppelingen niet per CRUD actie handmatig te configureren en kon het Ruby on Rails framework dit op basis van de opbouw van de *controller* klassen herkennen. Tevens kon ik hierdoor gebruik maken van andere functionaliteit binnen het framework zoals het automatisch genereren van internetadressen op basis van database-objecten.

Een andere optie was het verzinnen van eigen methoden en de koppelingen met de HTTP verzoeken handmatig te configureren. Naast dat dit me extra tijd en meer code zou opleveren waren er nog twee andere voordelen waar ik dan geen gebruik van kon maken.

Ten eerste is *CRUD* en *REST* een patroon dat veel gebruikt wordt om de data-manipulatielaag van webapplicaties te abstraheren. Op die manier snapt elke ontwikkelaar hoe er via HTTP commando's met een webapplicatie gecommuniceerd kan worden, zonder de interne details daarvan te weten. Hierdoor biedt je automatisch al een koppeling voor andere programmeurs met je webapplicatie aan, zonder dat je hier een aparte *API* laag hoeft te ontwikkelen.

Als tweede kan het voorkomen dat de HTTP standaard aangepast wordt en met daarbij het *REST* communicatieprotocol. Door de conventie van Ruby on Rails te volgen zorgde ik ervoor dat deze implementatielaag binnen het framework afgeschermd blijft. Mocht de standaard aangepast worden dan wordt dit binnen het framework intern bijgewerkt en zou ik mijn controllers niet opnieuw hoeven indelen.

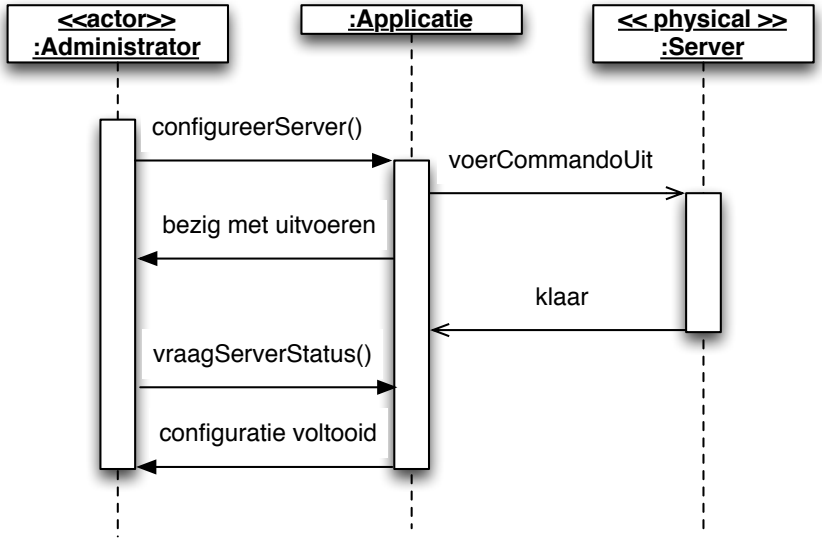
Ontwerpen commando-executie op servers

De kernfunctionaliteit van *Intercity* is het configureren van een server op basis van de configuratie die ingevoerd is in de webapplicatie. In de use case *Serverwijzigingen toepassen* wordt er door de applicatie op een server ingelogd om commando's uit te voeren die de benodigde mappenstructuren en processen klaar zet om een *Ruby on Rails applicatie* op een server te *deployen*.

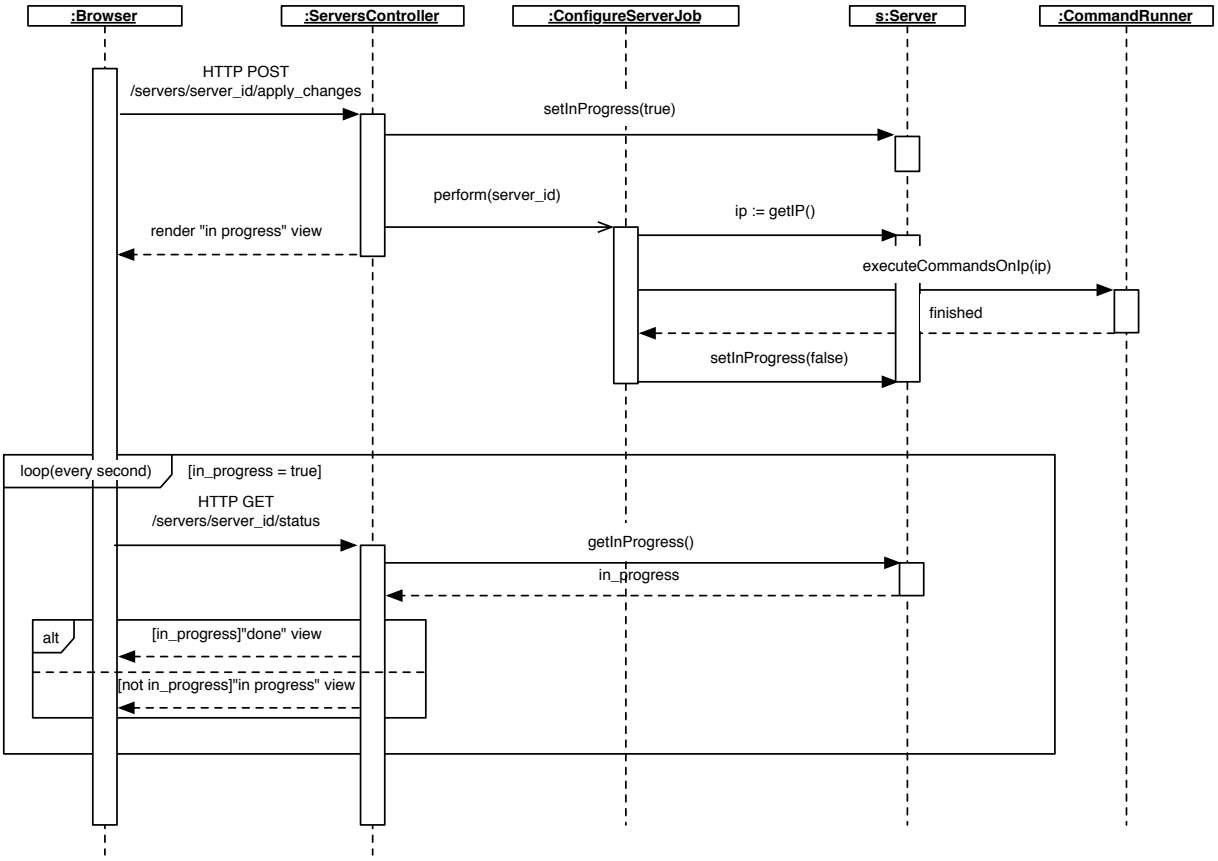
Een probleem dat ik in deze use case moest oplossen is dat het opzetten van een serververbinding en het uitvoeren van de commando's enkele seconden tot minuten kon duren. Dat deze taken enige tijd konden duren was een probleem omdat Ruby on Rails een HTTP verzoek en de interne communicatie tussen objecten synchroon uitvoert. Dus, pas nadat het hele HTTP verzoek en de commando's op de server uitgevoerd zijn stuurt de applicatie pas een respons naar de webbrowser van de gebruiker. Omdat de meeste browsers een maximum timeout limiet hebben van 30-120 seconden wordt de verbinding met de applicatie al verbroken, voordat de taken op de server uitgevoerd zijn. De gebruiker krijgt dan een foutmelding van de

browser te zien, terwijl de commando's op de server nog lopen. Omdat de foutmelding onterecht en te vroeg getoond wordt, kan de gebruiker dan niet een gebruiksvriendelijke manier terugkeren naar de applicatie. Daarnaast wilde ik vanuit het oogpunt van de gebruikerservaring, meteen een statusindicator kunnen weergeven dat de commando's op de achtergrond uitgevoerd worden en een statusterugkoppeling geven als deze voltooid zijn.

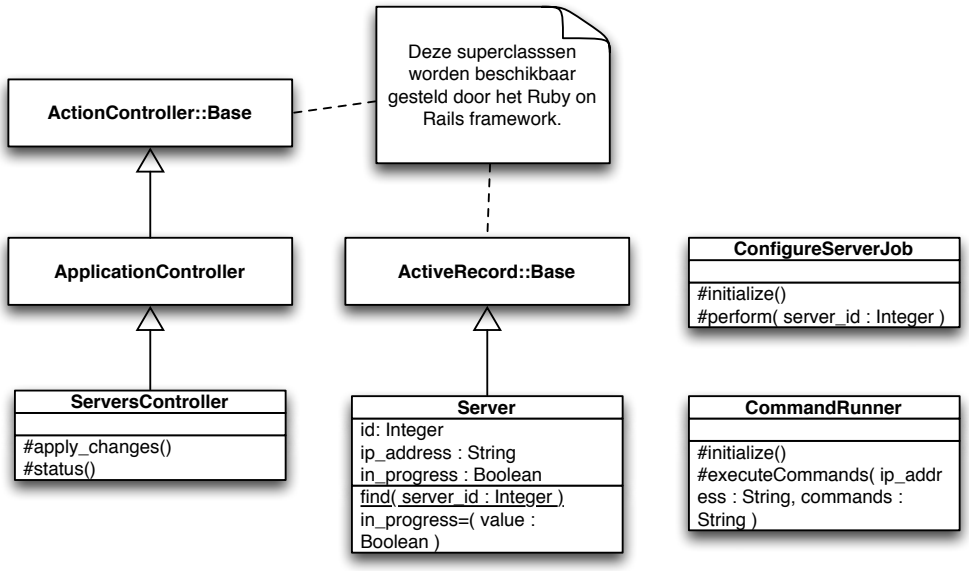
De interactie die ik wilde bereiken is weergegeven in het volgende sequentiediagram:



Deze interactie is als volgt ontworpen in de applicatie:



Dit diagram heeft geleid tot het klasse-ontwerp in het diagram hieronder. In dit diagram zijn de naamgevingen van enkele attributen en methoden veranderd. Dit is omdat het klasse-diagram voor het implementatiestadium is opgesteld. In het implementatiestadium worden de attributen en methoden weergegeven zoals ze geïmplementeerd zijn.



Met dit ontwerp zijn een aantal keuzen gemaakt die volgens een aantal redenen tot stand zijn gekomen.

In het ontwerp is er een `Server` klasse gemodelleerd. Deze klasse is het *model* binnen het *MVC* design pattern wat in dit ontwerp gebruikt is. Deze klasse overerft van de klasse `ActiveRecord::Base`. Dit is een klasse uit het *Ruby on Rails framework* dat onder andere zorgt voor het ophalen en opslaan van informatie uit de database.

Het attribuut `ip_address` is op het model vastgelegd zodat de gebruiker van de applicatie bij het aanmaken van een server in de applicatie het ip adres kan op slaan. Het ip adres wordt uiteindelijk gebruikt om binnen de `CommandRunner#executeCommands()` methode met de server verbinding te maken.

Het andere attribuut dat belangrijk is voor het configureren van de servers is het *in_progress* attribuut. Dit attribuut is toegevoegd om twee redenen. De eerste reden is te zien in het sequentiediagram in het gedeelte van de `loop(every second)`. De `ServersController` controleert in de methode `status()` op deze manier welk *view template* er aan de gebruiker weergegeven moet worden. De reden dat ik deze toestand opsloeg in de database middels het *Server* model is dat de `ServersController` dan op een manier konden controleren welke status we in de interface moesten geven. Een alternatief was deze status live op te halen van de daadwerkelijke server door elke keer een verbinding op te bouwen om te controleren of de commando's nog uitgevoerd worden. Omdat een connectie openen enige milliseconden kan duren en ook extra serverbelasting met zich mee brengt vond ik dit geen goede optie. Enerzijds betekent het vertraging in de user interface, anderzijds betekent het dat er ontzettend veel processing kracht op de server nodig is om bijvoorbeeld een paar honderd gebruikers tegelijkertijd de status van hun servers te laten oproepen.
z
In het sequentiediagram is te zien dat in de `ServersController` het attribuut `in_progress` op de waarde `true` wordt ingesteld. Nadat de commando's via de `ConfigureServerJob` uitgevoerd zijn wordt dit attribuut weer op de waarde `false` ingesteld. De reden dat dit op deze twee losse plekken wordt gedaan is dat de `ConfigureServerJob#perform()` methode asynchroon uitgevoerd wordt. In de implementatie wordt dit gedaan door een instantie van de `ConfigureServerJob` aan een *queing service* door te geven. Deze *queuing service* is een proces dat draait buiten het applicatieproces en handelt de doorgegeven *jobs* af. Het effect

hiervan is dat het enkele momenten kan duren voor een job gestart wordt. Als het `in_progress` attribuut pas in de `ConfigureServerJob` op `true` gezet zou worden dan zouden er twee problemen ontstaan. Allereerst zou de user interface van de applicatie niet direct reageren op het starten van de `apply_changes` actie door de gebruiker. Als tweede zou het kunnen voorkomen dat iemand (per ongeluk) twee keer de actie start als de queuing service de eerste job nog niet gestart heeft. Het resultaat hiervan is dat de *queuing service* twee jobs tegelijkertijd start die in conflict kunnen zijn met elkaar.

Als laatste wil ik het hebben over de indeling van de klassen die ik gekozen heb.

De *Server* klasse uit vorig model dient als representatie voor de fysieke server waarmee verbonden wordt. Deze klasse is dus enkel verantwoordelijk voor het bijhouden van de gegeven en toestand van een server en bevat geen verdere implementatie.

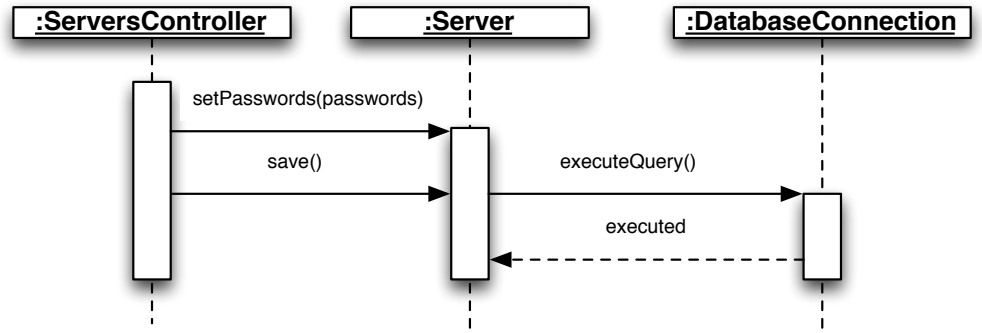
De functionaliteit die de `ConfigureServerJob` klasse bevat had ik ook in de *Server* klasse kunnen inbouwen. Ik heb hier niet voor gekozen omdat de `ConfigureServerJob` klasse naar de *queuing service* gestuurd moet worden. Deze service moet zo min mogelijk overhead hebben om zo snel mogelijk te kunnen draaien. Als de hele instantie van de *Server* klasse hierheen gestuurd zou worden zou de *queuing service* veel meer geheugen nodig hebben om te draaien. Daarnaast kon ik de *model*code van de `Server` die enkel toestand en attributen bijhoudt scheiden van de verantwoordelijkheid van het asynchroon commando's uitvoeren. Zo kon ik de manier om asynchroon commando's uit te voeren updaten zonder de code van het `Server` model te hoeven aanpassen. De laatste keuze voor deze scheiding is dat ik op die manier per uitgevoerde `ConfigureServerJob` bijvoorbeeld een logfile bij kon gaan houden en deze presenteren aan de gebruiker. Omdat een instantie van de `Server` klasse de representatie van een server is over zijn hele bestaan, kon ik hier geen tijdsindeling van afzonderlijke *jobs* in modelleren.

Versleutelen van servergegevens

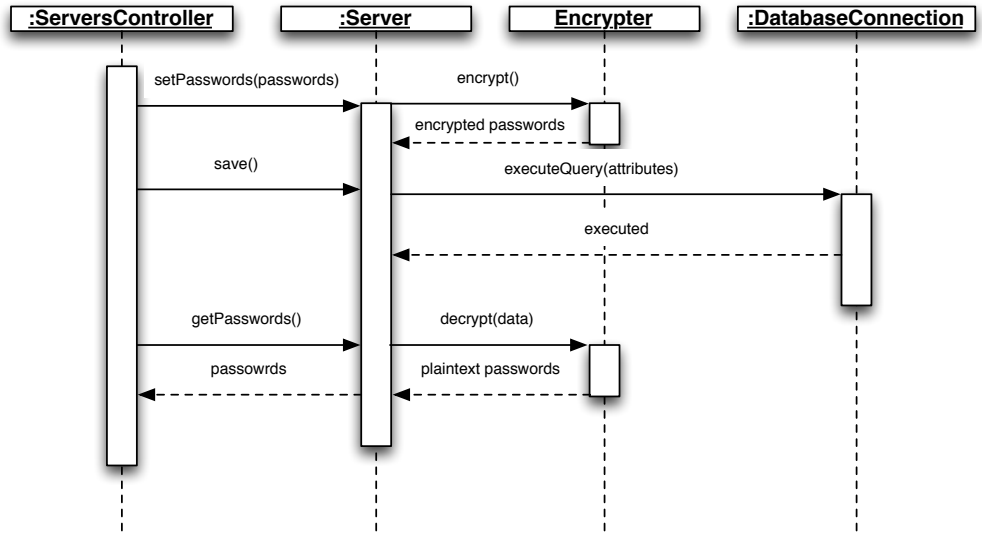
In de afgelopen paragrafen is beschreven hoe de use cases tot stand zijn gekomen met in het specifiek de use case om *Serverwijzigingen toe te passen*. In deze use case worden de servergegevens die in *Intercity* door de gebruiker ingesteld worden, geconfigureerd op de daadwerkelijke fysieke server zodat daar Ruby on Rails applicaties op gehost kunnen worden.

In de vorige paragraaf is het `Server` model aan bod gekomen dat verantwoordelijk is voor de opslag van deze gegevens in de database. Deze gegevens zijn onder andere wachtwoorden van databases en *SSH keys* om verbinding te leggen met een server. Deze gegevens wilde ik versleuteld opslaan in de database. Dit zou voorkomen dat iemand deze gegevens kon lezen als deze toegang zou kunnen krijgen tot de database server, een backup van de database of middels een SQL injection in de database zou uitlezen. Uit het ontwerp in deze paragraaf is de requirement ontstaan om in *PolarSSL for Ruby* een module te ontwikkelen waar data mee versleuteld en ontsleuteld kon worden.

In het *Ruby on Rails* framework worden gegevens van een *model* op de volgende manier opgehaald en opgeslagen. In het volgende diagram is het *Server* model als voorbeeld genomen dat gewijzigd wordt vanuit de `ServersController`.



Het ontwerp dat ik bedacht heb is als volgt:



Met dit ontwerp wilde ik toewerken naar het toepassen van het *Serializer* pattern. Dit pattern wordt gebruikt om tijdens het uitlezen of opslaan van gegevens een formaat-translatie te doen. In mijn geval was deze translatie van *plaintext* data naar *ciphertext* encrypted data en terug. De serializer klasse `Encrypter` hoeft hierbij geen kennis te hebben van het object dat de oorspronkelijke data beheert en kan zo als standaard gebruikt worden voor het *encrypten* en *decrypten* van de data van het `Server` model, maar ook van alle andere modellen.

Het voordeel aan het gebruik maken van dit pattern is dat de interne code van de modellen om data uit de database op te slaan en uit te lezen niet aangepast hoeft te worden. Het enige dat moet gebeuren is dat het model aangegeven krijgt dat bepaalde attributen met behulp van de `Encrypter` klasse gelezen en geschreven dienen te worden. Op deze manier kan de methode van encryptie ook aangepast worden, zonder dat het *Server* model veranderd hoeft te worden.

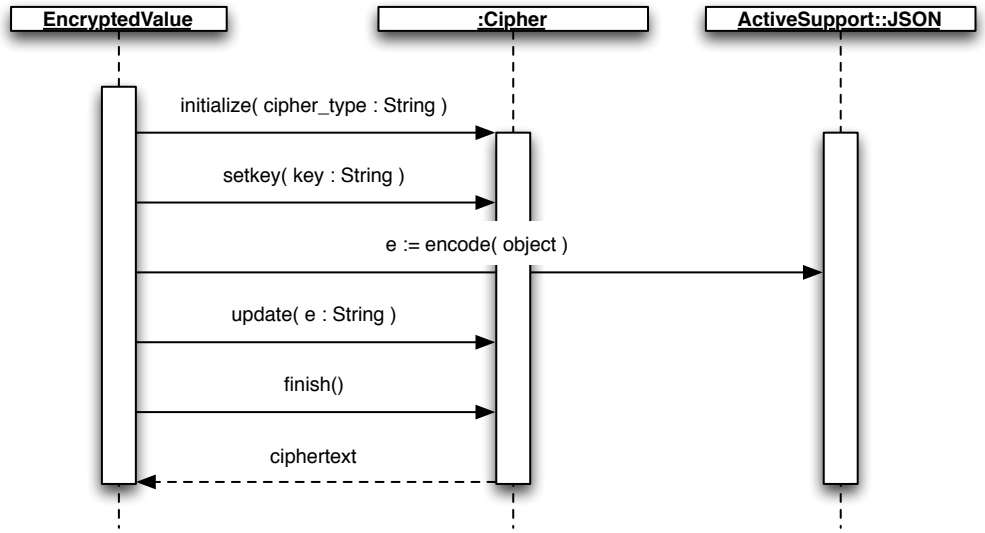
Een alternatief was het aanpassen van alle *getter* en *setter* methoden op de models waarvan de attributen versleuteld in de database opgeslagen moesten worden. Dit betekent meer code om te onderhouden en meer code in het geheugen van de applicatie. Tevens bood het Ruby on Rails framework een conventie aan om een attribuut middels het *Serializer* pattern te transformeren.

Het ontwerp van deze `Encrypter` klasse gaf mij informatie om een requirement te definiëren om in *PolarSSL for Ruby te ontwikkelen*. Namelijk, het kunnen *encrypten* en *decrypten* van gegevens. De ontwikkeling van deze requirement wordt in het volgende hoofdstuk beschreven.

7. Ontwerpen en implementeren Cipher klasse

Uit het ontwerp van de voorbeeldcase *Intercity* bleek dat er de wens was om op een manier data versleuteld op te slaan in de database. In dit hoofdstuk wordt beschreven hoe de functionaliteit van *PolarSSL for Ruby* is ontworpen om data te versleutelen en te ontsleutelen voor de *Encrypter* uit *Intercity*.

Het resultaat van de implementatie dat in dit hoofdstuk beschreven is wordt in het volgende sequentiediagram weergegeven:



In dit diagram is te zien dat de klasse *EncryptedValue* een aantal methoden uitvoert op een instantie van de *Cipher* klasse. Deze *Cipher* klasse is hetgeen dat in dit hoofdstuk ontworpen en ontwikkeld wordt en spreekt de functionaliteit aan die in de *PolarSSL libary* beschikbaar is om data te versleutelen en te ontsleutelen. De klasse *ActiveSupport::JSON* wordt gebruikt om de versleutelde data in de database op te slaan als het formaat *JSON*. Deze keuze zal in de volgende paragrafen ook gemotiveerd worden.

Dit hoofdstuk zal vergeleken met de andere hoofdstukken een stuk technischer zijn. Het grootste deel van de implementatie van de *Ruby wrapper* omvatte het leren kennen van principes van de programmeertaal C, het opbouwen van datastructuren en het principe van geheugenmanagement. Ik heb met schematische weergaves van code afgewisseld met daadwerkelijke code-voorbeelden geprobeerd een zo goed mogelijk beeld te geven van de uitdagingen die ik bij de implementatie ondervond.

Selecteren methode van encryptie

Er zijn meerdere methoden om een stuk data te versleutelen. Binnen de cryptografie worden deze methoden *ciphers* genoemd. Elke *cipher* heeft in de *PolarSSL library* zijn eigen implementatie met zijn eigen module met functies. Om te kunnen bepalen welke module ik uit de *PolarSSL library* in *PolarSSL for Ruby* moest implementeren moest ik eerst kiezen welke *cipher* ik kon gebruiken in *Intercity*.

Met mijn opdrachtgever heb ik besloten de *Advanced Encrypten Standard (AES)* cipher te gebruiken.

In de documentatie van *PolarSSL* heb ik gevonden dat er binnen *AES* drie modes zijn waarmee data versleuteld kan worden. Omdat elke mode anders geïmplementeerd wordt heb ik een vergelijking om te weten te komen welke *AES mode* het meest geschikt was.

AES staat voor Advanced Encryption Standard en is de opvolger van *DES (Data Encryption Standard)*.

Ik had de keuze uit drie *AES* modes en ik heb gekozen voor de *AES CTR* mode. Deze modus ondersteund namelijk om *plaintext* van een variabele lengte als input te gebruiken voor de versleuteling naar de *ciphertext*. Dit was nodig omdat de gegevens die binnen de *Intercity* webapplicatie versleuteld gaan worden van variabele lengte waren. Als ik voor de *AES ECB* modus had gekozen kon ik maar 16 bytes versleutelen. Met de CBC modus moest de lengte van mijn input data een veelvoud zijn van 16 bytes. Het had wel mogelijk geweest om CBC te gebruiken door bijvoorbeeld een willekeurig karakters aan mijn input data toe te voegen om precies een veelvoud van 16 bytes te bereiken. Maar, hiermee zou de encryptie gevoelig zijn voor *padding attacks*[7].

Mode	Mogelijke lengte encryptie	
AES ECB	16 bytes	-
AES CBC	Blokken van precies 16 bytes	-
AES CTR	Variabele lengte	+

Controle uitvoer versleutelde data met Base64 encoding

Een eerste probleem waar ik tegen aan liep was dat de uitvoer van de versleuteling in binaire bits was. Deze binaire bits kon ik niet op een leesbare manier op het scherm zetten om te controleren of de encryptie goed gegaan is. Ik had dus een manier nodig om te controleren of de uitvoer van de versleuteling op een manier te kunnen weergeven.

Base64 is een manier om binaire bits te converteren naar ASCII tekens zodat deze op een scherm of in tekstvorm opgeslagen kunnen worden.

Een gerelateerd probleem was dat ik de uitvoer met een externe bron moest controleren. Dat de functies uit het prototype goed uitgevoerd werden garandeerde nog niet dat de functies met de juiste parameters aangeroepen zijn. Het uitvoeren van de functies met de verkeerde parameters kan leiden tot verkeerde uitvoer van de versleuteling waardoor de uitvoer niet meer ontsleuteld kon worden en dus onbruikbaar is.

Om te controleren of mijn prototype de goede *ciphertext* als uitvoer genereerde heb ik ervoor gekozen om deze uitvoer om te zetten naar *Base64*. Op deze manier kon ik de binaire uitvoer uit het prototype vergelijken met de uitvoer van externe encryptietools die online te vinden zijn. Deze encryptietools gebruiken namelijk Base64 als uitvoer formaat om weer te geven.

---

[7] http://en.wikipedia.org/wiki/Padding_oracle_attack

Nadat ik met het prototype geïnventariseerd had welke datatypen en functies ik nodig had, kon ik beginnen met het ontwerpen van de *AES* module in *PolarSSL for Ruby*. Toen ik op het punt stond hiermee te beginnen, kreeg ik echter een tip van mij opdrachtgever.

Ontwerp van de Cipher module

Nadat ik het *AES* prototype werkend had gekregen kwam mijn opdrachtgever met het advies om eens te kijken naar de *Cipher* module in de *PolarSSL library*. Van deze module had ik nog niets gezien omdat deze niet in het module-overzicht van de documentatie voor kwam. Mijn opdrachtgever gaf me aan dat deze module gebruikt kon worden als abstractielaag voor alle *ciphers* die de *PolarSSL library* als losse modules aanbied. Dit zou betekenen dat ik met het implementeren van deze module in *PolarSSL for Ruby* alle typen *cipher*s aan kon roepen. Dus ook de *AES CTR* cipher mode.

Om dit te verifiëren besloot ik wederom een prototype in de programmeertaal C te maken om te ontdekken welke datastructuren en functie-aanroepen benodigd waren om in de *Cipher* module te implementeren in *PolarSSL for Ruby*. In de volgende tabel is weergeven welke datastructuren en functie-calls ik nodig had om een stuk data via *AES CTR* te versleutelen.

Datastructuren	Functiecalls
`struct cipher_info_t`	`cipher_info_from_string()`

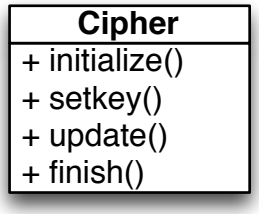
`struct cipher_context_t`	`cipher_init_ctx()`
	`cipher_setkey()`
	`cipher_reset()`
	`cipher_update()`
	`cipher_finish()`

In deze tabel zijn in de linkerkolom de datastructuren weergegeven met in de rechterkolom de functiecalls die een instelling of manipulatie doen met dat datatype. Dat wil zeggen dat de vier functies die naast het *cipher_context_t* datatype weergegeven zijn allemaal een operatie uitvoeren op dezelfde instantie van deze struct.

Datatype `struct` in C

Een `struct` is binnen de programmeertaal C een datatype waarin andere variabelen opgeslagen zijn. In een objectgörienteerde programmeertaal zouden deze variabelen gezien kunnen worden als attributen van een object.

Op basis van deze relatie tussen de datastructuren en de functiecalls kwam ik tot het volgende initiële klasse-ontwerp voor de *Cipher* klasse in *PolarSSL for Ruby:*



In dit diagram is te zien dat de functiecalls uit *PolarSSL for Ruby* nagenoeg één op één overgenomen zijn naar het klasse-ontwerp. Hiervoor had ik twee overwegingen.

Allereerst wilde ik de *interface* van *PolarSSL for Ruby* in de eerste versies zo veel mogelijk laten lijken op de *interface* die de *PolarSSL library* zelf ook hanteerde. Dit verhoogt de voorspelbaarheid van de library voor programmeurs die nu al bekend zijn met de *PolarSSL for Ruby* library. Dit is belangrijk voor de adoptie van *PolarSSL for Ruby*. Dat komt omdat er dan zo min mogelijk frictie is om met de *Ruby wrapper* te beginnen als er tutorials en guides worden gevolgd die origineel voor de *PolarSSL library* geschreven zijn.

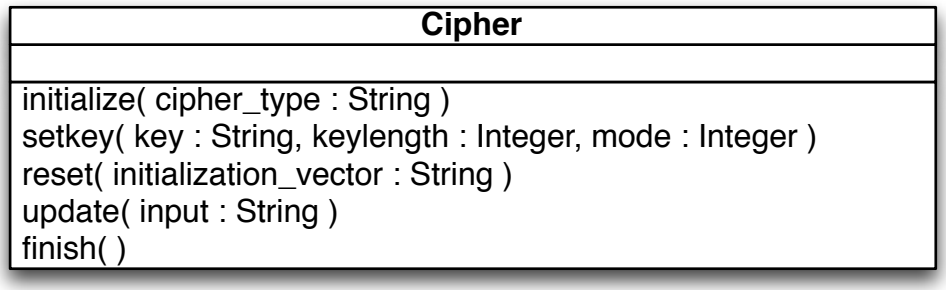
Om het voor een *Ruby programmeur* helemaal gemakkelijk te maken om data te versleutelen had ik ook een klasse-*interface* kunnen ontwerpen waarin de programmeur maar één methode uit hoefde te voeren om data te versleutelen, in plaats van de twee methods *update()* en *finish()*. Echter, een reden dat deze methodes gescheiden zijn is dat je hiermee data die in delen ontvangen wordt direct kunt encrypten en die toe kan voegen aan een *buffer*. Zo zorg je dat er nooit een groot deel onversleutelde data in het geheugen van het programma staat. Deze functionaliteit wilde ik in het ontwerp van de *Cipher* klasse niet teniet doen.

Wat wellicht ook op valt is dat ik geen ontwerp voor de *struct* `cipher_info_t` ontworpen heb. De reden hiervoor is dat het gebruik van deze struct en de functiecall `cipher_info_from_string()` enkel intern gebruikt zal worden in de `initialize()` methode op de *Cipher* klasse.

In de volgende paragrafen zal ik beschrijven hoe verschillende onderdelen van de *Cipher* klasse geïmplementeerd zijn in *PolarSSL for Ruby* en hoe deze klasse beschikbaar gemaakt is aan *Ruby* code.

Resultaat publieke interface Cipher klasse

De uiteindelijke publieke interface van de *Cipher* klasse in *PolarSSL for Ruby* ziet er als volgt uit:



Wat wellicht op valt is dat er geen klasse is ontworpen voor de `struct cipher_info_t`, terwijl dit datatype wel voorkwam in de tabel in de vorige paragraaf. De reden dat ik deze klasse niet heb geïmplementeerd is dat deze voor de gebruiker van de *Cipher* klasse niet meer relevant is. Deze `struct` kon namelijk binnen de `initialize()` methode opgezocht worden en direct in het interne geheugen van de Ruby klasse ingesteld worden in C-code. De Ruby programmeur hoeft hier dan geen kennis van te hebben. Omdat op deze struct verder ook geen handelingen verricht hoeven worden om deze *Cipher* klasse te gebruiken, vond ik het niet nodig deze wel te implementeren.Opbouw *wrapper* code

Wat ik tijdens het implementeren van de *Ruby wrapper* in feite heb gedaan is het toevoegen van nieuwe klassen en datatypen aan de programmeertaal Ruby. Ofwel, ik programmeerde de programmeertaal Ruby, zodat andere programmeurs gebruik konden maken van de klassen die de *C code* van *PolarSSL for Ruby* aan hen beschikbaar stelde.

Een probleem dat ik moest oplossen is hoe ik in de procedurele programmeertaal *C* een klasse kon definiëren met methoden die uitgevoerd konden worden op een instantie van die klasse. Middels het volgen van enkele *tutorials* en de *Ruby C extension README* kwam ik erachter dat ik in mijn *wrapper* moest definiëren welke *C functie* in mijn code uitgevoerd moest worden als een Ruby programmeur een methode in Ruby uitvoerde op mijn klassen. Zo is de publieke interface van de *Cipher* klasse vertaald naar de volgende C-interface in de code van de *Ruby wrapper*:

```
void  Init_cipher();
VALUE rb_cipher_allocate( VALUE klass );
VALUE rb_cipher_initialize( VALUE self, VALUE cipher_type );
VALUE rb_cipher_reset( VALUE self, VALUE initialization_vector );
VALUE rb_cipher_setkey( VALUE self, VALUE key, VALUE key_length, VALUE operation );
VALUE rb_cipher_update( VALUE self, VALUE rb_input );
VALUE rb_cipher_finish( VALUE self );
void  rb_cipher_free( rb_cipher_t *rb_cipher );
```

De benaming van de functie `Init_cipher()` wordt opgelegd door *Ruby* zelf. Met deze benaming kon de *Ruby interpreter* herkennen welke C-functie er uitgevoerd moet worden bij het inladen van de klasse met naam *Cipher*. In deze functie wordt gedefinieerd welke klassen, objecten en methoden er aan *Ruby* beschikbaar worden gesteld.

De functies reflecteren in de naamgeving zo veel mogelijk de originele *PolarSSL library* functies en de publieke interface van de *Cipher* klasse in Ruby. De reden hiervoor was dat ik wilde dat het continu duidelijk was welke functie uit Ruby, welke C-functie aan zou roepen en welke *PolarSSL* functie daar bij hoorde. Dit maakt het voor andere programmeurs direct traceerbaar om een stuk code op te zoeken of een fout op te sporen. Om deze reden heb ik ook de parameters dezelfde namen gegeven als in de publieke interface van de *Ciper* klasse.

De reden dat ik de functies de prefix `rb_` heb gegeven is dat ik op die manier geen conflicten kon krijgen met de functies uit de *PolarSSL library* zelf. Met deze prefix gaf ik aan dat deze functies door *Ruby* aangeroepen werden.

Implementatie van de Cipher klasse in Ruby

In de vorige paragraaf is een overzicht gegeven van de C-functies die de code van de *Ruby wrapper* heeft gekregen om de *Cipher* klasse te implementeren. Deze interface is het resultaat van een zoektocht hoe ik binnen de C-code datatypen vanuit Ruby code kon inlezen en converteren naar primitieve datatypen die standaard in C beschikbaar zijn.

Het basisprobleem dat ik in de wrapper moest oplossen was het in elk van bovenstaande functies beschikbaar hebben van een variabele van het type `struct cipher_context_t` uit de *PolarSSL library*. Deze variabele

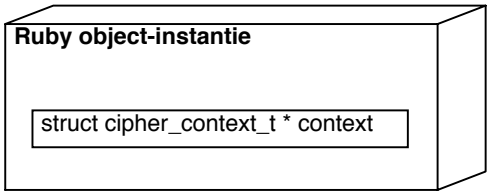
bevatte namelijk de informatie die ik aan elk van de functies uit de *cipher* module van de *PolarSSL library* moest meegeven.

Ik heb in twee soorten richtingen gezocht om dit probleem op te lossen:

1. Het op een manier meegeven van de cipher_context als parameter aan elke functie.
2. Het “opslaan” van de cipher_context als één van de private instance variabelen op het *Ruby* klasse object.

Deze eerste optie sprak mij het minst aan omdat dit zou betekenen dat deze parameter dan in de publieke Ruby interface van de *Cipher* klasse zou voorkomen en de Ruby programmeur deze dan elke keer zou opslaan en meegeven in de Ruby code. Om deze reden besloot ik deze optie te laten vallen en op zoek te gaan naar een oplossing in de richting van optie 2, het opslaan van de `cipher_context` variabele ergens binnen de interne geheugenruimte van de *Ruby* klasse.

Ik vond in de *Ruby C extension* documentatie dat er de mogelijkheid was om een variabele met het *C pointer* type op te slaan in het interne geheugen van een Ruby object. In de volgende afbeelding is dit schematisch weergegeven:



De kubus representeert de Ruby object-instantie waarmee gecommuniceerd wordt tussen de C code van de wrapper en de Ruby code van het programma dat de wrapper gebruikt. Dit object wordt elke keer doorgegeven aan de functies in de wrapper code als er een aanroep plaatsvind vanuit de Ruby code. Middels speciale functies binnen de Ruby C extension code kon ik deze opgeslagen pointer ophalen. Op deze manier had ik in alle functies toegang tot dezelfde cipher_context die ik nodig had voor het uitvoeren van de benodigde PolarSSL library functies. In de volgende twee code voorbeelden wordt de werking hiervan geïllustreerd:

Dit is een deel van de implementatie uit de *wrapper* waarin een `cipher_context` struct ingebed wordt in het Ruby object:

```
VALUE rb_cipher_allocate( VALUE klass )
{
    cipher_context_t *context;
    context = ALLOC( cipher_context_t );

    return Data_Wrap_Struct( klass, 0, -1, context ); /* cipher_context struct wordt ingebed */
}
```

Dit is een deel van de implementatie waarin de eerder gealloceerde `cipher_context` struct opgehaald wordt uit het Ruby object en gebruikt in een *PolarSSL library* functie:

```
VALUE rb_cipher_initialize( VALUE self, VALUE cipher_type )
{
    cipher_context_t *context;

    Data_Get_Struct( self, cipher_context_t, context ); /* cipher_context struct wordt opgehaald */

    cipher_init_ctx( context, cipher_info ); /* cipher_context wordt gebruikt in de
                                                aanroep van een PolarSSL functie */

    return self;
}
```

Een consequentie van deze oplossing was dat ik handmatig geheugenruimte moest alloceren om de `cipher_context` struct in op te slaan. In de programmeertaal C is het patroon dat als je zelf geheugen alloceert, dat je er ook zelf zorg voor moet dragen dat je dit handmatig vrijgeeft of expliciet door een andere methode laat vrijgeven. Als het geheugen niet op een expliciete manier door de programmeur wordt vrijgegeven, dan betekent dit dat deze geheugenruimte niet meer beschikbaar gemaakt kan worden voor andere programma's op hetzelfde systeem. Dit resulteert in een zogeheten *memory leak*.

Het voordeel van de oplossing die ik geïmplementeerd heb is dat ik hiermee automatisch gebruik kon maken van de *Ruby Garbage Collector*. De *Garbage Collector* zorgt er namelijk voor dat het geheugen dat ik voor de `cipher_context` klasse alloceer automatisch vrijgegeven wordt als het betreffende object in het Ruby programma van de gebruiker niet meer actief is.

Geheugenallocatie

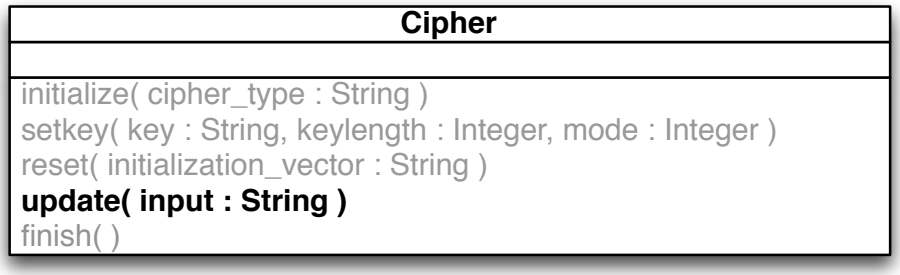
Binnen computerprogramma's wordt er geheugen gealloceerd om gegevens en variabelen in op te slaan. Bij de meeste *high level* talen zoals Java, C#, Ruby, PHP wordt dit door de taal zelf verzorgd. In C dient de programmeur er zelf voor te zorgen dat er geheugen gealloceerd wordt, maar ook weer vrijgegeven wordt.

Bij deze *high level* talen is er over het algemeen een **Garbage Collector** routine geïmplementeerd. Deze routine zorgt ervoor dat variabelen en gealloceerd geheugen dat niet meer beschikbaar hoeft te zijn vrijgegeven wordt aan het besturingssysteem.

Een **memory leak** ontstaat als een programma geheugen alloceert maar vervolgens niet vrijgeeft aan het besturingssysteem. Dit zorgt ervoor dat dit geheugen niet meer beschikbaar is voor andere programma's op het systeem. Het gehele geheugen van een computer kan hierdoor vollopen.

Implementatie van de `update` methode

De functie-aanroep heb ik in *PolarSSL for Ruby* geïmplementeerd met één parameter, zoals in het ontwerp van de *Cipher* klasse weergegeven is:



De interface van de originele *PolarSSL library* functie die door deze Ruby-methode aangesproken moest worden was als volgt:

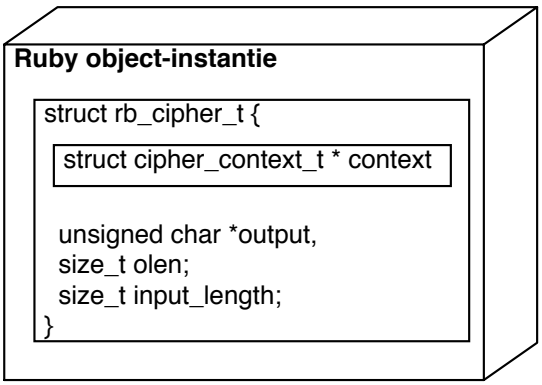
```
int cipher_update       (   cipher_context_t *      ctx,
                            const unsigned char *   input,
                            size_t                  ilen,
                            unsigned char *         output,
                            size_t *                olen
                        )
```

De reden dat ik alle parameters behalve de *input string* heb weggelaten heb dat deze andere parameters enkel als tijdelijke interne variabelen voor het versleutelen benodigd zijn. Ze bevatten geen informatie die de gebruiker van *PolarSSL library* zelf zou willen opgeven want het zijn geen opties of instellingen die de versleuteling beïnvloeden.

Omdat ik deze variabelen niet middels de publieke interface van de *Cipher* klasse heb geïmplementeerd moest ik een manier vinden om deze in de *wrapper* code intern wel bij te kunnen houden.

Het probleem was dat er maar één struct in de interne C-geheugenruimte van een Ruby object opgeslagen kon worden, zoals beschreven in de vorige paragraaf. Ik kon mijn de variabelen die benodigd waren voor de *cipher_update()* functie niet naast de `cipher_context_t` struct opslaan. Ik moest dus een oplossing vinden om zowel de `cipher_context_t struct` als de variabelen die ik nodig had voor de `cipher_update()` methode op te slaan in de interne geheugenruimte van mijn Ruby object.

Als oplossing heb ik een nieuw `struct` datatype met de naam `rb_cipher_t` gedefinieerd waarin ik de extra variabelen en de `cipher_context` op kon slaan. Schematisch ziet het Ruby-object er dan als volgt uit:



Door de originele `struct cipher_context_t` in mijn eigen *struct* op te slaan hoefde ik maar één *struct* op te slaan in het interne Ruby object dat meegegeven wordt aan elke C-functie in de *wrapper*. Omdat ik nu de `cipher_context_t` struct genest via mijn eigen `struct` kon ophalen kon ik deze ook meegeven aan de originele functiecalls die ik aan moest roepen in de *PolarSSL library*.

Resultaat *Cipher* klasse

In het volgende testscript is te zien hoe de *Cipher* klasse uit *PolarSSL for Ruby* gebruikt kan worden in een Ruby programma. Dit is het resultaat van de implementatie van deze feature.

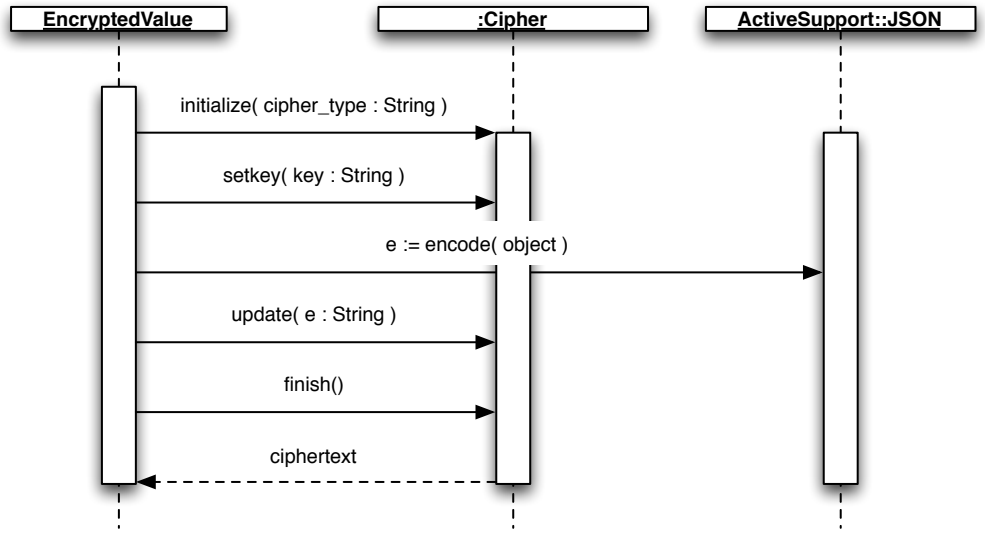
```
require 'test_helper'
require 'base64'
require 'securerandom'

class CipherTest < MiniTest::Unit::TestCase

  def test_aes_128_ctr_encrypt
    key = hex_to_bin("2b7e151628aed2a6abf7158809cf4f3c")
    iv = hex_to_bin("f0f1f2f3f4f5f6f7f8f9fafbfcfdfeff")
    input = hex_to_bin("6bc1bee22e409f96e93d7e117393172aae2d8a571e03ac9c9eb76fac45af8e51")
    should_encrypt_as =
hex_to_bin("874d6191b620e3261bef6864990db6ce9806f66b7970fdff8617187bb9fffdff")

    cipher = PolarSSL::Cipher.new("AES-128-CTR")
    cipher.setkey(key, 128, PolarSSL::Cipher::OPERATION_ENCRYPT)
    cipher.reset(iv)
    cipher.update(input)
    encrypted = cipher.finish

    assert_equal should_encrypt_as, encrypted
  end
end
```

Aan de hand van deze werkende testcase kon ik de integratie met *Intercity* implementeren. Deze integratie beschrijf ik in het volgende hoofdstuk.

10. Integreren *Cipher* klasse in *Intercity*

Aan de hand van het geïmplementeerde ontwerp van de voorbeeld webapplicatie *Intercity* en de testscripts en documentatie die ik geschreven heb voor de *Cipher* klasse kon ik de functionaliteit ontwikkelen die ervoor ging zorgen dat bepaalde gegevens van de *Server* klasse binnen *Intercity* versleuteld opgeslagen konden worden in de database.

Het doel was te komen tot de implementatie van het volgende ontwerp waartoe gekomen is in de vorige hoofdstukken:



Tijdens de implementatie van de *Cipher* klasse in de *wrapper code* kwam ik erachter dat ik voor het opslaan van de versleutelde gegevens nog geen rekening had gehouden met de zogenaamde *initialization vector*. Dit is een waarde die ik bij het voor het versleutelen moest genereren en tijdens het ontsleutelen mee moest kunnen geven aan de *Cipher* klasse. Ik moest daarom een manier vinden om zowel de gegenereerde *initialization vector* waarde als de *versleutelde binaire* waarde op één plek op te slaan.

Een **initialization vector** word gebruikt om de uitvoer van een encryption met eenzelfde key willekeurig te maken.

Ofwel, als het woord “hallo” encrypt zou worden met het de key “sleutel” dan zou hier zonder *initialization vector* altijd dezelfde waarde uitkomen. Op deze manier kan iemand door patronen te herkennen in versleutelde data kunnen raden wat andere data bevat als dit met dezelfde sleutel encrypt is.

Ik heb ervoor gekozen om de binaire bytes van de gegenereerde *initialization vector* vooraan de *ciphertext* te plakken. Omdat de lengte van de *initialization vector* altijd 16 bytes is kon ik voor het *decrypten* van de gegevens in de database-kolom de eerste 16 bytes uitlezen en de rest van de data beschouwen als de *ciphertext*.

Het opslagformaat is in de tabel op de volgende pagina gevisualiseerd met een kolom genaamd *mysql_passwords*. De cijfer- en letterduo’s die gescheiden zijn met een ‘:’ representeren elk een *byte* middels *hex notatie*.

Kolom *mysql_passwords* voor een *Server* record	
initialization vector (16 bytes)	*ciphertext* (*n* bytes)
3c:5a:3f:ee:2f:36:ac:64:32:43:d3:23:2a:f1:2c:b5	8c:23:1a:3f:..:..:..:..

Een alternatief was deze *initialization vector* in een aparte databasekolom op te slaan zodat vanuit een logisch standpunt duidelijk is dat er een scheiding is tussen de *initialization vector* en de *ciphertext* is. Vanuit een implementatie-oogpunt heb ik hier niet voor gekozen omdat de code die een kolom *encrypt* en *decrypt* middels het *Serializer* pattern binnen het Ruby on Rails framework op één kolom werkt. Door de gescheiden implementatie van dit pattern had ik geen toegang tot de andere databasekolommen voor het record. Daarnaast wilde ik niet dat iemand die een *database dump* zou bemachtigen al op basis van de kolommenstructuur kon zien welke methode er gebruikt werd voor de encryptie van gegevens. Door de *initialization vector* als binaire samen met de *ciphertext* op te slaan is het voor een buitenstaander moeilijker te raden welke manier van *encryption* gebruikt is.

Ik had er ook voor kunnen kiezen om de *initialization vector* en de *ciphertext* wel in dezelfde kolom op te slaan maar bijvoorbeeld te scheiden door een enkel karakter zoals een dubbele punt of een ander karakter. Dan zou ik in de implementatie de inhoud van de kolom eerst moeten splitsen op deze dubbele punt zodat ik de *initialization vector* en de *ciphertext* kon scheiden. Hier heb ik niet voor gekozen omdat het vinden van karakter in een reeks bytes meer stappen kost dan het van te voren uittellen van 16 bytes. Daarnaast zou het kunnen voorkomen dat de willekeurig gegenereerde *initialization vector* een dubbele punt bevatte waardoor de data in de kolom verkeerd gesplitst zou worden.

Deze implementatie heeft geresulteerd in de volgende Ruby code voor het *encrypten* van een Ruby object met een willekeurig gegenereerde *initialization vector*:

```
initialization_vector = SecureRandom.random_bytes(16)
object_json = ActiveSupport::JSON::encode(obj)

cipher = PolarSSL::Cipher.new('AES-128-CTR')
cipher.setkey(KEY, 128, PolarSSL::Cipher::OPERATION_ENCRYPT)
cipher.reset(initialization_vector)
cipher.update(object_json)

encrypted_object = cipher.finish()
encrypted_data = “#{initialization_vector}#{encrypted_object}”
```

En voor het *decrypten* van de data uit een database kolom:

```
initialization_vector = column_data[0..15]
ciphertext = column_data[15..-1]

cipher = PolarSSL::Cipher.new('AES-128-CTR')
cipher.setkey(KEY, 128, PolarSSL::Cipher::OPERATION_ENCRYPT)
cipher.reset(initialization_vector)
cipher.update(ciphertext)

ruby_object = ActiveSupport::JSON::decode(cipher.finish)
```

Deel 3 - Evaluatie

11. Procesevaluatie

In dit hoofdstuk evalueer ik de keuzes die ik in mijn Plan van Aanpak heb gemaakt en hoe ik deze tijdens de opdracht ten uitvoer gebracht heb. Ik beschrijf hoe *Kanban* en eXtreme Programming hebben geholpen bij het behalen van mijn doelen en of ik de globale planning die ik opgesteld heb behaald heb.

Kanban

Middels de toepassing van de principes van *Kanban* toheb ik dat ik vanaf het begin van het afstudeerproject alle energie kunnen richten op het opleveren van de eerste directe klantwaarde. Omdat *Kanban* me de mogelijkheid gaf om elke feature afzonderlijk te beschouwen heb ik aan het begin van mijn afstuderen geen extra energie hoeven besteden aan het uitdenken van requirements voor *Intercity* en het uitwerken van de andere features. Hiermee kon ik voldoen aan het doel om zo vroeg mogelijk een eerste feature van *PolarSSL for Ruby* te ontwerpen, testen en ontwikkelen en te releasen als open source project.

Achteraf gezien heb ik minder gebruik gemaakt van de methode van *Kanban* om problemen in het proces op te merken dan ik oorspronkelijk dacht. Ik heb alleen in het begin actief gebruik gemaakt van *theory of constraints* toepassing van Kanban. Dat heb ik gedaan door mijn aandacht in het begin op maximaal twee onderdelen van de opdracht te richten: de selectie van de *wrappingtechniek* en het leren van de programmeertaal C te oriënteren. Hierna ben ik de functionaliteit die ik moest ontwerpen en ontwikkelen één voor één gaan uitvoeren om elke functionaliteit zo snel mogelijk door de de stadia design, development, documentation en release te voeren.

In die zin heb ik de filosofie van *Kanban* juist erg consequent nageleefd tijdens dit project. Ik denk echter dat *Kanban* binnen een team waar meerdere ontwikkelaars, meerdere disciplines tegelijkertijd uitvoeren meer waarde heeft om problemen binnen een ontwikkelproces te herkennen.

eXtreme Programming

De activiteiten binnen eXtreme Programming heb ik erg consequent overgenomen bij het ontwikkelen van elke feature. Ik heb voor de *SSL connection feature*, het ontwerp van de webapplicatie *Intercity* en de *Cipher feature* elke keer een *prototype spike* uitgevoerd en op basis daarvan een globaal ontwerp opgesteld. Die heb ik vervolgens vertaald naar één of meerdere testcases om het ontwerp en de implementatie van de code vanuit die testcases evolutionair te laten ontstaan. De ontwikkelde code is tijdens het ontwikkelen continu automatisch getest door mij en de de opgezette *continuous integration* service. Na elke succesvolle testcase heb ik een *refactorronde* toegepast om de code en het ontwerp uitbreidbaar te maken.

Door deze stappen van eXtreme Programming te volgen heb ik ervoor kunnen zorgen dat er bij de oplevering van elke feature goede code ontstond die gedocumenteerd en getest was. De opgeleverde code van de ene feature kon daardoor goed uitgebreid worden met de volgende feature zonder dat het nodig was in het begin te veel ontwerpbeslissingen te maken die op dat moment nog niet relevant waren.

(Niet) Behalen planning

Ik heb niet alle onderdelen opgeleverd die ik had willen opleveren. Ik had bijvoorbeeld veel meer functionaliteit van de *PolarSSL library* willen implementeren in *PolarSSL for Ruby*. De hoofdoorzaak is geweest dat het veel tijd heeft gekost de concepten van de programmeertaal C, geheugenbeheer te leren en uit te zoeken hoe *Ruby C extensions* intern werkten. Dit was voor mij een totaal onbekend *low level* programmeergebied ten opzichte van de bekende *high level* object-georiënteerde werwijze. Met de kennis die ik hierdoor opgedaan heb van C en de interne werking van Ruby snap ik nu wel veel beter de andere tools die beschikbaar waren voor het wrappen van een andere taal in Ruby. Hierdoor kan ik nu veel sneller nieuwe features ontwikkelen in de *PolarSSL for Ruby* wrapper en snap ik veel beter hoe een programmeertaal intern functioneert.

Een andere oorzaak was dat het ontwerp en de ontwikkeling van de webapplicatie *Intercity* veel tijd in beslag heeft genomen. Tijdens de uitvoering heb ik zo veel mogelijk use cases proberen weg te snijden van mijn oorspronkelijke plan om toch zo veel mogelijk tijd te besteden aan de ontwikkeling van *PolarSSL for Ruby*. Als *Intercity* geen onderdeel van deze afstudeeropdracht was geweest had ik verder gekomen met de implementatie van *PolarSSL for Ruby*. Aan de andere kant is er nu wel een webapplicatie waar *PolarSSL for Ruby* in geïntegreerd is en waarmee bewezen kan worden dat de *wrapper* werkt. De basis van de *wrapper* is zodanig sterk met de aanwezige documentatie, inrichting van de code en het opzetten van het open source project dat het relatief eenvoudig is om de *wrapper* nu uit te gaan breiden met nieuwe functionaliteit.

Aan het onderdeel om *PolarSSL for Ruby* binnen de iOS development tool *RubyMotion* werkend te maken ben ik niet meer toegekomen om vorige redenen. Ik had wel kunnen kiezen om andere features uit *PolarSSL for Ruby* niet te ontwikkelen en wél aan de integratie met *RubyMotion* te beginnen. Echter zou het dit niet veel bijdragen aan de gekozen beroepstaken *Ontwerpen systeemdeel* en *Bouwen applicatie* kon uitvoeren. Het integreren van *PolarSSL for Ruby* zou namelijk vooral gaan over het configureren van *RubyMotion* om *PolarSSL* te kunnen compilen.

12. Productevaluatie

In dit hoofdstuk beschrijf ik per hoofdproduct of de gebruikte technieken en het opgeleverde resultaat voldoet aan de verwachting die ik had aan het begin van het afstuderen. Ik beoordeel daarmee het resultaat van de *PolarSSL for Ruby* wrapper en de webapplicatie *Intercity*.

PolarSSL for Ruby

Zoals in het vorige hoofdstuk is beschreven heb ik minder functionaliteit geïmplementeerd dan ik had gewild. Ik ben wel postitief over met de manier waarop de *wrapper* ontworpen en geïmplementeerd is. Er is een ontwikkelomgeving opgezet waar de C-code van automatisch gecompiled en getest wordt. Op deze manier is gewaarborgd dat de functionaliteit van de wrapper blijft werken als er updates worden gedaan en kan deze controle volledig automatisch uitgevoerd worden.

Daarnaast ben ik tevreden met de keuze voor *Ruby C extensions*. Het effect van deze keuze was wel dat ik in de programmeertaal C en de interne werking van Ruby moest duiken en dat heeft veel tijd in beslag genomen..Aan de andere kant heeft dit ertoe geleid dat er volledig controleerbare veilige code geïmplementeerd is en dat er geen afhankelijkheden zijn van tools die niet standaard met Ruby meegeleverd worden. Omdat de *Ruby C extensions* de officieel methode is om de Ruby programmeertaal uit te breiden werkt *PolarSSL for Ruby* met alle standaardtools die in de Ruby community bekend zijn en voldoet de *wrapper* aan de criteria die ik tijdens de vergelijking van de wrappingtechnieken gebruikt heb.

Resultaat

Er zijn in totaal drie releases van *PolarSSL for Ruby* afgerond:

- SSL connectie feature
- Eerste versie *Cipher* feature
- Uitbreiding *Cipher* feature met ondersteuning voor *initialization vectors*.

Volgens de statistieken die door *Rubygems.org* bijgehouden[8] worden is de *wrapper* in totaal **434** keer gedownload. De eerste SSL connectie feature is na release al **130** keer gedownload.

Tevens is op de website *GitHub.com* een source project gestart met bijbehorende *README* en *LICENSE* bestanden en is er online documentatie beschikbaar[9] waarmee Ruby programmeurs de werking van de *wrapper* kunnen inzien.

De C-code van de *wrapper* is geïmplementeerd volgens de coding standards van *PolarSSL*[10] en er is een document met richtlijnen opgeleverd waarmee nieuwe functionaliteit op een consistente manier geïmplementeerd kan worden in de *wrapper* en daarmee de ontwikkeling overgedragen worden aan andere

---

[8] http://rubygems.org/gems/polarssl/

[9] http://michiels.github.io/polarssl-ruby/doc/

[10] https://polarssl.org/kb/development/polarssl-coding-standards

programmeurs. De code is *Test Driven* en middels *refactoring* geïmplementeerd waardoor het ontwerp en de implementatie uitbreidbaar is.

Er zijn test scripts geschreven in het testframework *MiniTest* zodat deze automatisch uitgevoerd kunnen worden en daarmee als regressietesten dienen. De *continuous integration* service *TravisCI.org* is opgezet waarmee alle testscripts bij elke wijziging aan de code uitgevoerd wordt. Op deze manier wordt de werking van de huidige functionaliteit gewaarborgd.

Webapplicatie *Intercity*

Ik ben tevreden met de functionaliteit en het ontwerp dat geresulteerd heeft in de lancering van een beta versie van de webapplicatie *Intercity*. Op dit moment nodigen wij andere webbureaus uit om de applicatie te te proberen.

Het doel van *Intercity* is om te bewijzen dat *PolarSSL for Ruby* gebruikt kon worden in een webapplicatie. Met de integratie van de *Cipher* module die ervoor zorgt dat gevoelige data in de applicatie versleuteld wordt wordt er bewezen dat *PolarSSL for Ruby* een belangrijke taak binnen *Intercity* vervult.
Zoals ook in de vorige evaluatieparagrafen aangegeven is had ik gewild dat ik meer functionaliteit uit *PolarSSL* had willen ontwikkelen. Er worden binnen de huidige versie van *Intercity* bijvoorbeeld *SSH keys* gegenereerd door een andere SSL library dan *PolarSSL*. Ik had deze functionaliteit graag met *PolarSSL for Ruby* willen implementeren. Ik denk wel dat het versleutelen van data een grotere wens is dan het genereren van *SSH keys*. Bijna alle websites doen namelijk iets met persoonlijke gegevens van personen. Niet alle websites leveren de mogelijkheid om in te loggen op een server.

Naast de integratie van *PolarSSL for Ruby* ben ik ook tevreden met het ontwerp en de implementatie van het asynchroon kunnen installeren van een server op afstand. Deze functionaliteit gaat voorbij de functionaliteit van de meeste applicaties, waar enkel data invoer, weergave en -uitvoer vaak staan.

Resultaat

De volgende requirements zijn geïmplementeerd in de applicatie *Intercity*:

- User signup
- User login
- Servers beheren
- Applicaties beheren
- Uitvoeren configuratiescript op een server
- Encrypten van gegevens met *PolarSSL for Ruby*

Er is een beta-versie gelanceerd op http://intercityapp.com en er is een marketingwebsite opgezet op http://intercityup.com om beta-gebruikers aan te spreken. Enkele screenshots van de beta-versie zijn weergegeven in *Bijlage 5: Screenshots Intercity*.

13. Conclusie

Ik ben positief over het resultaat wat ik bereikt heb binnen deze afstudeerperiode. Er staat een sterke basis van het open source project *PolarSSL for Ruby* waar in snel tempo andere functionaliteit uit de *PolarSSL library* voor ontwikkeld kan worden door de aanwezige documentatie en testtools. De waarde van de *wrapper* is bewezen door de integrate in de voorbeeldcase *Intercity*. Van *Intercity* is een beta versie gelanceerd waar enkele gebruikers al bezig zijn om hun servers te configureren om Ruby on Rails applicaties op te hosten.

Waar ik nog enthousiaster over ben dan het opgeleverde resultaat is wat ik tijdens deze afstudeeropdracht geleerd heb. Ik heb een flinke kennisbasis opgebouwd van *cryptografie* door het implementeren van de functionaliteit uit *PolarSSL*. Dit heeft tot kennis van digitale beveiliging geleid die ik nog niet kende en kan gebruiken in mijn verdere carrière als programmeur.

Naast deze kennis van *cryptografie* heb ik geleerd hoe je in de programmeertaal C software kunt ontwikkelen, hoe *pointers* werken en wat er allemaal komt bij komt kijken als je handmatig geheugenbeheer moet doen. Dit heeft mij de basis gegeven om zowel een *low level* als *high level* programmeur te zijn. Dit staat in nauwe relatie met de kennis die ik opgedaan heb van de programmeertaal Ruby, en hoe je in een procedurele programmeertaal een objectgeörienteerde programmeertaal kunt implementeren.

Het aspect waar ik nog het meest tevreden mee ben is dat ik iets voor andere Ruby programmeurs heb kunnen doen en dat ik heb geleerd hoe je een programmeertaal programmeert. Ik heb de functionaliteit die de *PolarSSL* library verschaft beschikbaar kunnen maken aan alle Ruby programmeurs in de weide wereld die daar eerder nog geen toegang toe hadden. Het feit dat een veelvoud van programmeurs deze nieuwe functionaliteit kan gebruiken door mijn eenmalige investering is het mooiste dat er bestaat.

Bijlagen

Bijlage 1: Afstudeerplan

Afstudeerplan

Informatie afstudeerder en gastbedrijf (*structuur niet wijzigen*)

Afstudeerblok: 2013-1.2 (start uiterlijk 6 mei 2013)
Startdatum uitvoering afstudeeropdracht:
Inleverdatum afstudeerdossier volgens jaarrooster: 23 september 2013

Studentnummer: 20062643
Achternaam: dhr Sikkes (*) *weghalen niet van toepassing*
Voorletters: M
Roepnaam: Michiel
Adres: Kleiweg 82N
Postcode: 2801GJ
Woonplaats: Gouda
Telefoonnummer:
Mobiel nummer: 0624964508
Privé emailadres: michiel.sikkes@gmail.com

Opleiding: Informatica
Locatie: Zoetermeer (*) *weghalen niet van toepassing*
Variant: voltijd

Naam studieloopbaanbegeleider: T. Cocx
Naam begeleidend examinator: V. E. Broeren
Naam tweede examinator: R. Ruijsenaars

Naam bedrijf: PolarSSL (Offspark B.V.)
Afdeling bedrijf:
Bezoekadres bedrijf: Laan van Zuid Hoorn 49
Postcode bezoekadres: 2289 DC
Postbusnummer:
Postcode postbusnummer:
Plaats: Rijswijk
Telefoon bedrijf: 010 223 0411
Telefax bedrijf:
Internetsite bedrijf: https://polarssl.org

Achternaam opdrachtgever: dhr Bakker
Voorletters opdrachtgever: P (Paul)
Titulatuur opdrachtgever:
Functie opdrachtgever: Directeur
Doorkiesnummer opdrachtgever:
Email opdrachtgever: p.j.bakker@offspark.com

Achternaam bedrijfsmentor: dhr Bakker
Voorletters bedrijfsmentor: P (Paul)
Titulatuur bedrijfsmentor:
Functie bedrijfsmentor: Directeur
Doorkiesnummer bedrijfsmentor:
Email bedrijfsmentor: p.j.bakker@offspark.com
NB: bedrijfsmentor mag dezelfde zijn als de opdrachtgever

Doorkiesnummer afstudeerder:
Functie afstudeerder (deeltijd/duaal):

Titel afstudeeropdracht: Writing a C/C++ to Ruby Wrapper for PolarSSL and integrating it into a web application hosting product.

Opdrachtomschrijving *(toelichtende tekst verwijderen)*

1. Bedrijf

PolarSSL is een onderdeel van Offspark B.V. Binnen PolarSSL wordt er primair een open source C/C++ SSL library ontwikkeld en onderhouden. Tevens verkoopt PolarSSL commerciële licenties en levert het commerciële ondersteuning.

Offspark B.V. is een commercieel bedrijf dat zich voornamelijk richt op software ontwikkelaars die SSL beveiliging in hun techniek moeten gebruiken middels het product PolarSSL. Hierbij richt het zich onder andere specifiek op ontwikkelaars die deze software in embedded systemen dienen te verwerken.

Het bedrijf bestaat op dit moment uit drie ontwikkelende werknemers die zich primair richten op de ontwikkeling van PolarSSL. Binnen dit team zal mijn afstudeeropdracht ook plaatsvinden.

De directeur en oprichter van Offspark, Paul Bakker, heeft jarenlang ervaring in de security branche wegens het mede-oprichten van de Crypto-tak van het bedrijf Fox IT, een belangrijke leverancier van advies en producten op het gebied van informatiebeveiliging voor overheden en bedrijven.

De activiteiten van het PolarSSL team richten zich vooral op het ontwikkelen van features en oplossen van fouten binnen de SSL library, het leveren van ondersteuning aan de open source community en aan commerciële klanten en de promotie en verkoop van de library.

2. Probleemstelling

Op dit moment zijn de meeste klanten van PolarSSL grote internationale bedrijven die PolarSSL middels een commerciële licentie jaarlijks afnemen. De wens is dat PolarSSL ook meer gebruikt gaat worden door kleinere commerciële klanten om zo de schaal van het klantenbestand te vergroten.

Tot nu zijn er hier al een aantal veranderingen voor gedaan zoals het toegankelijker maken van de website en het schrijven van guides en tutorials om PolarSSL ook bij kleine commerciële ontwikkelaars op de kaart te zetten. Daarnaast hebben er kleine marktonderzoeken plaats gevonden om te beoordelen welke abbonnementsvormen en prijzen kleine ontwikkelaars aantrekkelijk zouden vinden voor het afnemen van PolarSSL.

Binnen deze opdracht wil het bedrijf deze activiteiten uitbreiden door nog een derde actie toe te voegen. Namelijk, het beschikbaar maken van de library in programmeertalen buiten C en C++ om. Het is daarom het idee om PolarSSL bruikbaar te gaan maken voor Ruby programmeurs zodat zij deze library in hun eigen Ruby software kunnen gaan integreren.

Er wordt op dit moment gedacht aan het maken van een Ruby wrapper library waarmee Ruby programmeurs de mogelijkheid krijgen via een Ruby interface de mogelijkheden van de library te gebruiken. Een wrapper is eenvoudig geschreven. Echter, is het voor wrappers gebruikelijk om niet alleen de functionaliteit van de originele library alleen over te nemen, maar deze functionaliteit ook op zo'n interface beschikbaar te stellen, dat het voldoet aan de Ruby-regels en conventies. Het zal dus nodig zijn op zoek te gaan naar deze conventies zodat de wrapper aan voelt alsof deze native in Ruby geschreven zou zijn.

Voor het schrijven van een wrapper zijn meerdere technieken en tools beschikbaar. Deze tools en technieken zullen geïnventariseerd en vergeleken moeten worden. Hieruit zal dan een keuze komen met welke techniek de wrapper geschreven gaat worden. Daarbij moet meegenomen worden dat de wrapper op de meestgebruikte besturingssystemen gebruikt kan worden maar dat deze ook geschikt

gemaakt zal moeten worden voor toepassing op embedded devices. De ontwikkelaars van Ruby zijn bezig om een variant die mRuby heet te gaan maken. Er zal onderzocht moeten worden of de wrapper ook geschikt gemaakt kan worden voor mRuby.

Een ander probleem is dat PolarSSL ontzettend veel functionaliteit aanbiedt. Er zal in overleg met de opdrachtgever een selectie gemaakt moeten worden van welke functionaliteit er in de Ruby wrapper beschikbaar gesteld wordt en een suggestie opgesteld worden voor doorontwikkeling.

Tevens missen er op dit moment nog concrete praktijkcases die aantonen hoe makkelijk PolarSSL te gebruiken is tegenover de OpenSSL library en wordt in de Ruby community het gebruik van OpenSSL standaard nagevolgd. Daarom gaat binnen deze opdracht ook een case ontwikkeld worden waarin wordt aangetoond hoe de Ruby “versie” van PolarSSL binnen een nieuw hostingproduct gebruikt is.

Met deze integratie hoopt Offspark B.V. dat PolarSSL binnen de Ruby community meer mensen commercieel gebruik zullen maken van PolarSSL.

3. Doelstelling van de afstudeeropdracht

Het doel van deze afstudeeropdracht is het schrijven van een wrapper om de huidige PolarSSL library heen zodat deze in Ruby applicaties gebruikt kan worden. Er wordt bewijs geleverd dat deze wrapper ook daadwerkelijk in gebruik genomen kan worden door deze te verwerken in een praktijkcase.

4. Resultaat

Aan het eind van de stageopdracht zal er een Ruby wrapper geschreven worden waarin de in samenspraak met de opdrachtgever geselecteerde functionaliteit van PolarSSL gebruikt kan worden.

Er zal een andere applicatie ontworpen en gebouwd worden die gaat bewijzen dat de Ruby wrapper in applicaties geïntegreerd kan worden.

Er is een vergelijking gemaakt van tools en technieken om een C-library in Ruby te wrappen. Hierbij is rekening gehouden dat de wrapper op de meest gebruikte besturingssystemen in gebruik genomen kan worden en of deze ook op embedded devices gebruikt kan worden door middel van mRuby.

5. Uit te voeren werkzaamheden, inclusief een globale fasering, mijlpalen en bijbehorende activiteiten

Mijlpaal	Activiteiten
Plan van Aanpak	• Methoden selecteren • Op te leveren producten inplannen • Fasering bepalen • Opdracht formuleren

Mijlpaal	Activiteiten
Selectie wrapping-techniek	• Inventarisatie middelen om een C-library te wrappen • Proof of concept / prototype voor geselecteerde techniek programmeren
Ontwerp praktijkcase	• Opstellen requirements praktijkcase • Modeleren architectuur praktijkcase • Requirements voor wrapper opstellen
Ontwikkeling Ruby wrapper	• Selecteren requirements met opdrachtgever • Selecteren requirements voor praktijkcase • Onderzoek en opstellen guidelines van generiek Ruby library-conventies (API guidelines) • Modelleren en vertaling PolarSSL naar Ruby library • Ontwikkelen wrapper • Opzetten API documentatie • Opzetten open source project Ruby wrapper • Testen van de wrapper (testplan)
Integratie praktijkcase en wrapper	• Programmeren praktijkcase • Testen van integratie met Ruby wrapper

6. Op te leveren (tussen)producten

- Plan van Aanpak
- Selectie-documentatie Ruby wrapping technieken
- Proof of Concept Ruby wrapper
- Requirements praktijkcase
- Architectuur praktijkcase
- Requirements Ruby wrapper
- API Guidelines Ruby wrapper
- Architectuur Ruby wrapper
- Programmacode Ruby wrapper
- API documentatie Ruby Wrapper
- Open source GitHub project Ruy wrapper
- Testplan Ruby wrapper
- Programmacode praktijkcase
- Integratie testplan Ruby wrapper en praktijkcase

7. Te demonstreren competenties en wijze waarop

1.3 Selecteren van standaardsoftware

Deze beroepstaak ga ik in deze opdracht invullen door te inventariseren op welke manieren je in Ruby een C-library kan wrappen. Vervolgens stel ik samen met de opdrachtgever een aantal criteria op waaraan de Ruby wrapper zal moeten voldoen, en bekijk ik welke techniek het best voldoet aan deze criteria.

3.2 Ontwerpen systeemdeel

Deze beroepstaak ga ik in deze opdracht op twee manieren invullen.

Allereerst door het ontwerpen van het systeemdeel van de praktijkcase die als input zal dienen voor de requirements van de wrapper.

Daarnaast wordt het systeemdeel van de Ruby wrapper zelf ontworpen. Dit ontwerp zal voortkomen uit de requirements van de praktijkcase en na een selectie van de functionaliteit die PolarSSL op dit moment aanbiedt. Het ontwerp zal moeten voldoen aan een op te stellen guideline voor Ruby libraries.

3.3 Bouwen applicatie

Deze beroepstaak ga ik vervullen door het ontwerp van de wrapper om te zetten in code zodat deze in de praktijkcase en in andere applicaties gebruikt kan worden. Dat doe ik door in de wrapper de Ruby en eventuele C code te schrijven die nodig is. In de praktijkcase zal het betekenen dat ik een deel van een Ruby on Rails applicatie bouw waar zowel Ruby, JavaScript, CSS en HTML wordt geschreven.

Tijdens het bouwen zal ik unit, functionele en acceptatietests schrijven om te valideren of zowel de interne onderdelen van de wrapper en de applicatie goed werken en zodat de opgestelde functionaliteit in acceptatietests wordt beschreven en te testen is.

De wijzigingen van de code en utility scripts zullen bijgehouden worden middels versiebeheersoftware Git. De code van de wrapper wordt in ieder geval open source beschikbaar gesteld op GitHub.

Bijlage 2: Plan van Aanpak

Inleiding

Dit Plan van Aanpak is geschreven in het kader van de afstudeeropdracht van Michiel Sikkes in de periode mei 2013 t/m oktober 2013. De afstudeeropdracht wordt uitgevoerd bij het bedrijf Offspark B.V. Dit bedrijf ontwikkeld en verkoopt de softwarebibliotheek *PolarSSL*. Deze softwarebibliotheek stelt programmeurs die in staat in hun programma's beveiligde verbindingen op te zetten en data te beveiligen.

De aanleiding voor de afstudeeropdracht is dat Offspark B.V. op zoek is om deze *PolarSSL library* aan een grotere groep ontwikkelaars aan te kunnen bieden. De afstudeeropdracht omvat in dit kader de ontwikkeling van een stuk software waarmee programmeurs die in de programmeertaal Ruby werken gebruik kunnen gaan maken van *PolarSSL*.

Het probleem is dat *PolarSSL* een softwarebibliotheek voor de programmeertaal C geschreven is en alleen geïntegreerd kan worden in software die geschreven is in C of een variant daarvan (zoals C++). Binnen de programma's die in de programmeertaal Ruby geschreven worden is het niet direct mogelijk een C-library te integreren. Als Offspark B.V. ook Ruby ontwikkelaars aan wilt spreken dan is het nodig om een koppeling tot stand te brengen tussen de *PolarSSL library* en de programmeertaal Ruby.

Het doel van de afstudeeropdracht is het ontwerpen en ontwikkelen van een *software wrapper* die het mogelijk maakt instructies uit de *PolarSSL library* aan te spreken binnen een Ruby programma. Binnen de opdracht zal deze *software wrapper* de naam *PolarSSL for Ruby* krijgen.

Er zijn meerdere technieken beschikbaar om een *software wrapper* tussen de taal C en Ruby te ontwikkelen. Daarom zal er eerst een vergelijking gedaan worden om een geschikte techniek te selecteren.

Omdat er de wens ligt om in een vroeg stadium met potentiële gebruikers in contact te komen wordt er één feature uit de gehele *PolarSSL library* gekozen om als eerst te ontwikkelen en vrij te geven.

De verdere functionaliteit zal bepaald worden aan de hand van het ontwerp van de webapplicatie met de naam *Intercity*. Het ontwerp van deze applicatie zal resulteren in requirements die input zijn voor de verdere ontwikkeling van *PolarSSL for Ruby*. De requirements die uit het ontwerp van de voorbeeldcase *Intercity* blijken zullen gebruikt worden om de *software wrapper* uit te breiden. Als laatst zal de *software wrapper* geïntegreerd worden met de webapplicatie *Intercity*.

Werkwijze en methodes

De projectmanagementmethode die zal worden gebruikt is de methode *Kanban*. Deze methode maakt het mogelijk om losse features volledig op te leveren inclusief code, tests en documentatie. Tevens maakt deze methode het mogelijk om tijdens het project nieuwe op te stellen om deze vervolgens door te ontwikkelen.

Voor het ontwerp en de ontwikkeling van de software zal er gewerkt worden middels eXtreme Programming (XP). Deze softwareontwikkelmethodiek leent zich voor het inbedden in *Kanban*. Door het werken volgens

de regels van XP zal gegarandeerd worden dat de opgeleverde delen van de software altijd werken en uitbreidbaar zijn.

De stappen die voor de ontwikkeling van elke funtionaliteit binnen dit project gevolgd worden zijn:

1. Prototype / design
2. Development / testing
3. Documentation
4. Releasing

Hieronder volgt een korte beschrijving van welke activiteiten er in elke stap uitgevoerd worden:

Prototype / design

In deze stap wordt er op basis van de beschikbare informatie uit de *PolarSSL library* of beschikbare use cases een globaal ontwerp gemaakt om verdere ontwikkeling op te kunnen baseren.

Als het nodig is om vooronderzoek te doen in een bepaalde oplossingsrichting, omdat de implicaties niet van te voren voorzien kunnen worden, wordt er een *spike* uitgevoerd. In een *spike* wordt er een prototype ontwikkeld waar één of meerdere oplossingsrichting in worden bekeken om te beoordelen welke richting ingezet kan worden voor het ontwikkelen van de functionaliteit.

Development / testing

In deze stap wordt het globale ontwerp dat in de vorige stap is opgesteld vertaald naar een detailontwerp en geïmplementeerde code. Dit ontwerp komt tot stand door de functionaliteit middels *Test Driven Development (TDD)* te ontwikkelen. Daarbij wordt er alvorens met de implementatie van de code te beginnen eerst een mogelijke interface vastgelegd in een testscript. De stappen van dit testscript zullen per stuk geïmplementeerd worden tot werkende code. Op deze manier zal hier automatisch een ontwerp van klassen, methoden en parameters uit voortvloeien.

Na de implementatie van elke succesvol testscript zal er *refactoring* plaatsvinden. De activiteit *refactoring* houdt in dat het ontwerp en de code geïnspecteerd wordt om te bekijken of deze verbeterd kan worden.

De code die ontstaat tijdens de ontwikkeling wordt gecommit en beschikbaar gesteld op het *PolarSSL for Ruby* project op de website *GitHub.com*. Er zal een *continuous integration* service ingericht worden waar elke code-wijziging automatisch middels de geschreven testscripts wordt getest.

Documentation

Binnen deze stap worden er middels code-commentaar instructies toegevoegd aan de code die de publieke interface van de opgeleverde functionaliteit te beschrijven. Met een documentatie-generator wordt op basis van dit code-commentaar een webpagina gegenereerd waar programmeurs de klassen, methoden en parameters van de functionaliteit kunnen opzoeken.

Daarnaast wordt het bestand *README* van het project bijgewerkt. In deze README komen instructies te staan hoe de software geïnstalleerd kan worden en wordt er een *Getting started* voorbeeld opgenomen zodat nieuwe programmeurs direct aan de slag kunnen.

Releasing

In deze stap wordt de code en de documentatie uit de vorige stappen gebundeld en onder een versienummer publiek gemaakt. Dit gebeurt door op *GitHub* een *release* aan te maken.
De release met het betreffende versienummer wordt daarna als *Ruby Gem* gepackaged en op *RubyGems.org* gepubliceerd zodat andere programmeurs de gem in hun eigen applicaties kunnen integreren.

Middelen en tools

Code hosting - GitHub.com
Continuous integration service - Travis CI
Code generatie - Rdoc
Kanban planning - Trello.com

Globale planning

Mijlpaal	Activiteiten
Plan van Aanpak	• Methoden selecteren • Op te leveren producten inplannen • Fasering bepalen • Opdracht formuleren
Selectie wrapping-techniek	• Inventarisatie middelen om een C-library te wrappen • Proof of concept / prototype voor geselecteerde techniek programmeren
Ontwerp praktijkcase	• Opstellen requirements praktijkcase • Modeleren architectuur praktijkcase • Requirements voor wrapper opstellen
Ontwikkeling Ruby wrapper	• Selecteren requirements met opdrachtgever • Selecteren requirements voor praktijkcase • Onderzoek en opstellen guidelines van generiek Ruby library-conventies (API guidelines) • Modelleren en vertaling PolarSSL naar Ruby library • Ontwikkelen wrapper • Opzetten API documentatie • Opzetten open source project Ruby wrapper • Testen van de wrapper (testplan)
Integratie praktijkcase en wrapper	• Programmeren praktijkcase • Testen van integratie met Ruby wrapper

Bijlage 3: Testplan PolarSSL for Ruby

Inleiding

In dit document wordt kort beschreven hoe de *PolarSSL for Ruby* wrapper middels *Test Driven Development (TDD)* wordt ontwikkeld. Het doel van dit document is het verschaffen van een overzicht welke soorten tests er geschreven worden en met welke tools deze tests uitgevoerd worden.

Soorten tests

Binnen dit project worden er twee soorten testscripts ontwikkeld:

- Integration tests
- Unit tests

Voordat de eerste code van een functionaliteit geprogrammeerd wordt, wordt er eerst een **integration test** geschreven. Deze integration test wordt gezien als de specificatie waar naartoe gewerkt wordt middels *Test Driven Development*. De integration test bevat een specifiek scenario van een use case waarin de aanroep van meerdere klassen en methoden met elkaar tot een gebruikersresultaat moeten lijden.

De **unit tests** worden geschreven om specifieke methoden op een klasse te testen. Eén unit testscript verifieert de werking van één methode. Binnen een dergelijk testscript kunnen er meerdere invoer- en uitvoerscenario's getest worden.

Implementatie

Deze testscripts worden in de broncode van *PolarSSL for Ruby* opgenomen in de directory `test/`. Elke afzonderlijke klasse krijgt zijn eigen unit testscript en elke afzonderlijke functionaliteit zijn eigen integratietestscript.

Het *MiniTest* test framework zal gebruikt worden om de testscripts te implementeren. Dit framework bevat *assertion* functies om de waarden in een testscript te controleren. Daarnaast biedt het framework een commando aan waarmee alle tests automatisch uitgevoerd kunnen worden.

Als het volgende commando uitgevoerd wordt in de ontwikkelomgeving zullen alle testscripts automatisch uitgevoerd worden en zal er een testrapport op het scherm getoond worden.

```
rake test
```

Continuous integration middels Travis CI

De testscripts kunnen door de programmeur in de lokale ontwikkelomgeving uitgevoerd worden met het commando uit de vorige paragraaf.

Er wordt ook gebruik gemaakt van de *continuous integration* service TravisCI.com. Deze service zorgt ervoor dat alle testscripts automatisch in een testomgeving worden uitgevoerd zodra er een code-wijziging op de site GitHub.com geplaatst wordt. Deze dienst stuurt vervolgens een testrapport via e-mail. Door deze controle kan er nooit vergeten worden om de testscripts lokaal uit te voeren worden de tests in een onafhankelijke testomgeving uitgevoerd.

De rapporten van de testuitvoeren worden beschikbaar gesteld op https://travis-ci.org/michiels/polarssl-ruby.

Bijlage 4: API guidelines Ruby wrapper

In dit document worden de regels bijgehouden waarmee de functie-aanroepen en datatypen uit de procedurele *PolarSSL library* vertaald worden naar *PolarSSL for Ruby*. Op deze manier kan de consistentie van de indeling van de *Ruby wrapper* gewaarborgd blijven en kunnen nieuwe ontwikkelaars op basis deze regels nieuwe functionaliteit toevoegen aan de *wrapper*.

Struct met functies -> Klasse met methoden

De *PolarSSL* library is opgedeeld in modules. Binnen deze modulen worden variabelen met het datatype `struct` gebruikt om informatie tussen de functies in de module door te geven. Als er meerdere functiecalls beschikbaar zijn die een operatie uitvoeren op eenzelfde `struct` dan worden deze in de *wrapper* code vertaald naar een klasse met methoden.

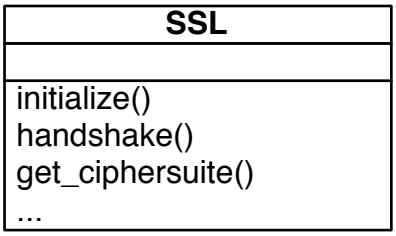
Om tot de klasse-naam te komen wordt de suffix `_context` van de struct naam afgehaald. `ssl_context` wordt klasse `PolarSSL::SSL`.

Voorbeeld

In de SSL wordt er gebruik gemaakt van een `ssl_context struct`. Er zijn meerdere functies met de prefix `ssl_` die operaties op deze struct uitvoeren:

```
ssl_context context;

ssl_init(&context);
ssl_handshake(&context)
ssl_get_ciphersuite(&context)
...
```

Deze struct en functies worden vertaald naar de klasse `PolarSSL::SSL` zoals in het volgende diagram te zien is:



Fouten afvangen met Exceptions

Als een functie in de *PolarSSL library* een fout teruggeeft wordt er een negatieve `int` waarde teruggegeven. Deze waardes worden in de *Ruby wrapper* afgevangen en vertaald naar het raisen van een exception:

```
    int ret = cipher_reset( rb_cipher->ctx, iv );

    if ( ret < 0 ) {
        rb_raise( e_BadInputData, "Either the cipher type, key or initialization vector was not
set." );
    }
```

Interne variabelen verbergen in de publieke interface

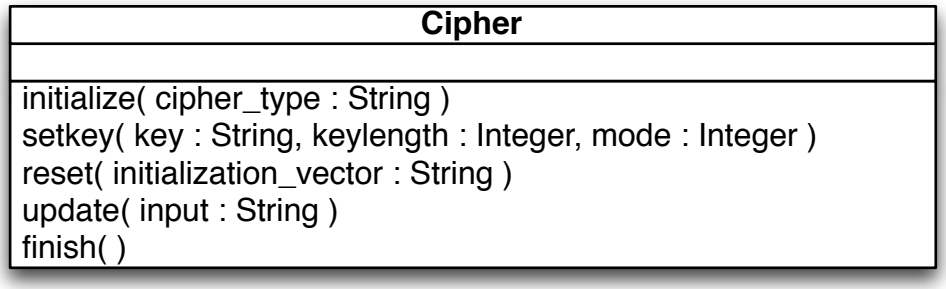
Als er aan een *PolarSSL* functie interne variabelen meegegeven moeten worden die niet relevant zijn voor de gebruiker dan worden deze in de publieke interface verborgen en intern in de *Ruby wrapper code* bijgehouden.

Voorbeeld

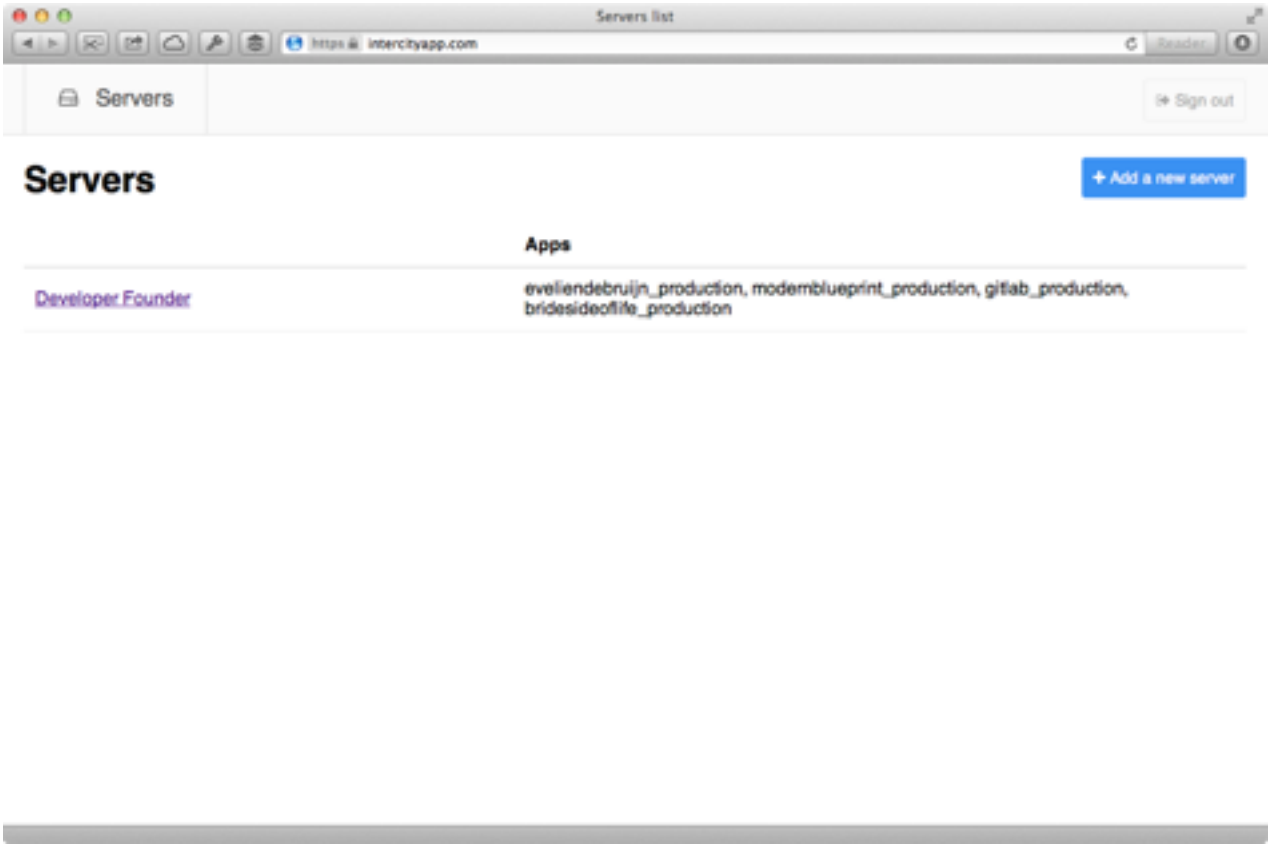
De functie `cipher_update()` heeft de volgende interface in *PolarSSL*:

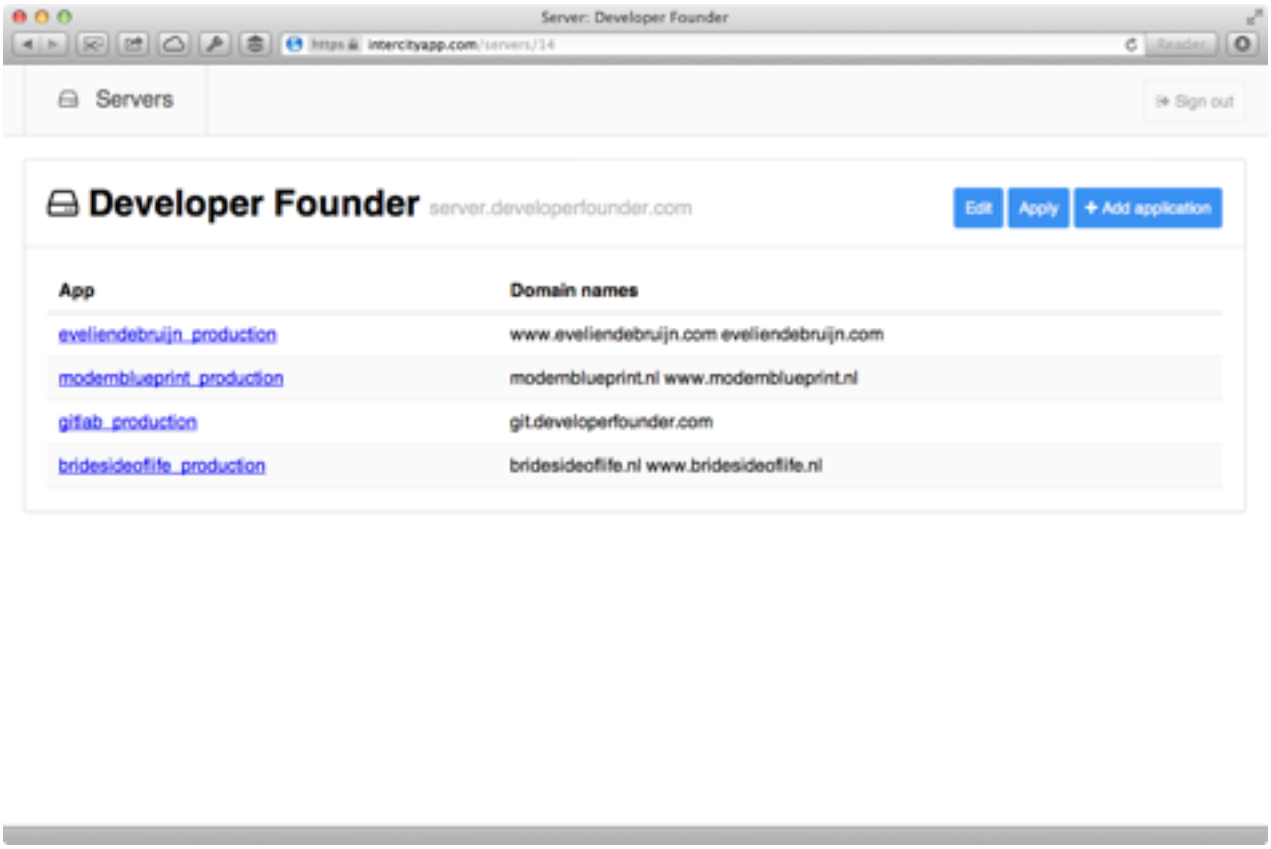
```
int cipher_update(
    cipher_context_t *ctx,
    const_unsighed char * input,
    size_t ilen,
    unsigned char * output,
    size_t * olen
);
```

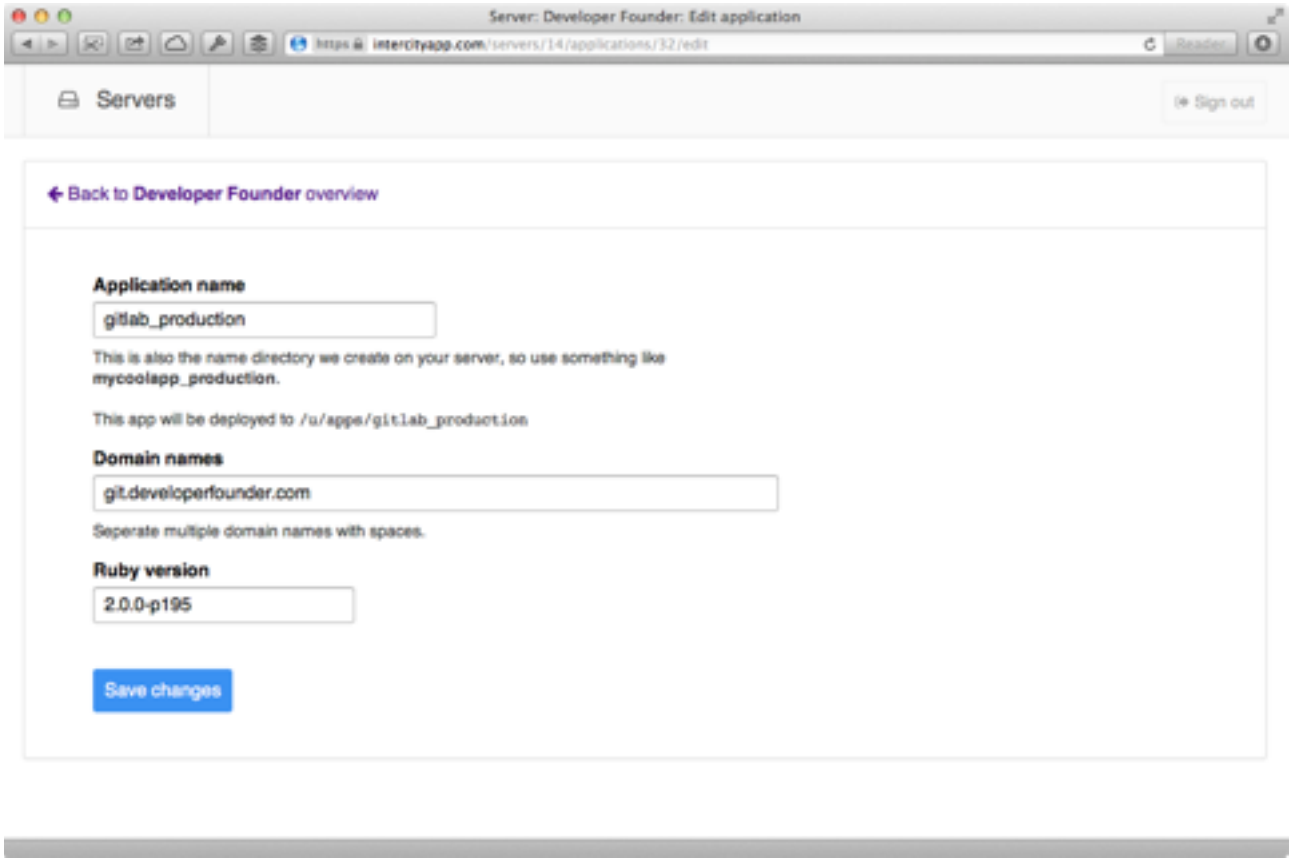
De argumenten `ilen, output, olen` kunnen op basis van de `input` variabele afgeleid worden of worden enkel intern gebruikt om informatie van de ene functiecall naar de volgende functiecall mee te geven. Deze variabelen worden in depublieke interface van de `PolarSSL::Cipher` klasse weggelaten:

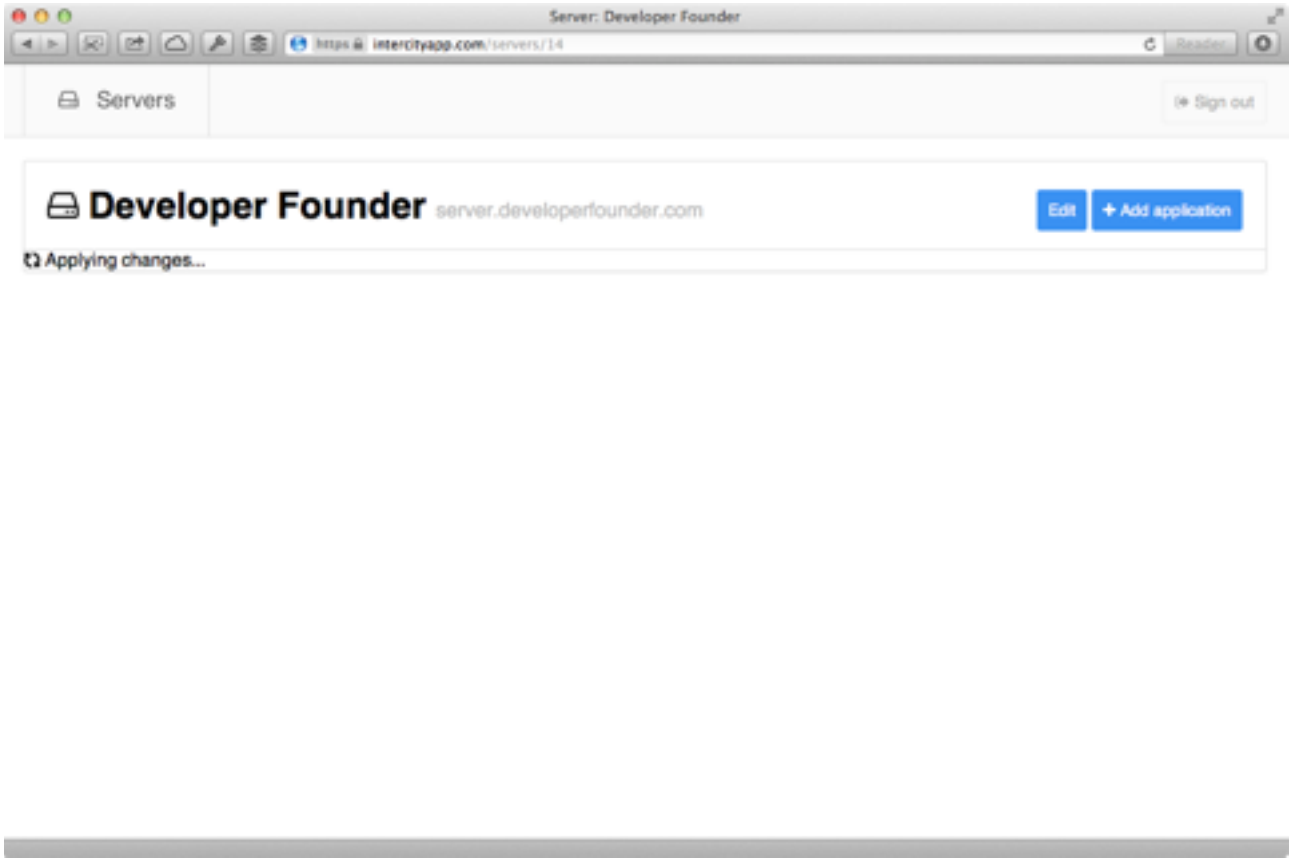


Bijlage 5: Screenshots Intercity






Server: Developer Founder: Edit application
https intercityapp.com/servers/14/applications/32/edit
Reader
Servers
Sign out
Back to Developer Founder overview
Application name
gitlab_production
This is also the name directory we create on your server, so use something like mycoolapp_production.
This app will be deployed to /u/apps/gitlab_production
Domain names
git.developerfounder.com
Seperate multiple domain names with spaces.
Ruby version
2.0.0-p195
Save changes


Server: Developer Founder
https intercityapp.com/servers/14
Reader
Servers
Sign out
Developer Founder
server.developerfounder.com
Edit
+ Add application
Applying changes...

Verklarende woordenlijst

(software) library - Softwarebibliotheek die bepaalde functionaliteit aanbiedt zodat deze gebruikt kan worden door andere programma's.

feature - Functioneel op te leveren deel van een stuk software.

encryption / *encrypten* - versleuteling (van gegevens)

decryption / *decrypten* - ontsluiteling (van gegevens)

plaintext - De gegevens in onversleutelde vorm

cyphertext - De gegevens in versleutelde vorm

deployen, deployer - Het publiceren van een nieuwe versie van een Ruby on Rails applicatie op een server.

Ruby wrapper - Software plugin die het mogelijk maakt functionaliteit uit een C library beschikbaar te maken voor Ruby programmeurs.

api - Application Programmable Interface. In de context van dit afstudeerproject vrij gedefineerd als de verzameling publiek aan te roepen classen en methoden van een software library.

PolarSSL library - De SSL en cryptography C-library die wordt onderhouden en verkocht door mijn opdrachtgever, *Offspark B.V* / *Paul Bakker*.

PolarSSL for Ruby - De te schrijven Ruby wrapper voor de *PolarSSL library*, het onderwerp van deze opdracht

OpenSSL library - Een software library die net als de PolarSSL library functionaliteit aanbiedt om communicatie te beveiligen.

Intercity - Hostingdashboard dat gebruikt wordt als praktijkcase voor *PolarSSL for Ruby*

wrappen - Het schrijven van Ruby extentiecode die functionaliteit uit de *PolarSSL library* beschikbaar maakt in *PolarSSL for Ruby*.

methode - Naam voor een functie op een klasse in *Ruby*.

refactoren - Code herschrijven om het voor de toekomst aanpasbaar en voor anderen leesbaar te maken, zonder iets aan de werking te veranderen.

compilen - Het vertalen van programmacode naar een (binair) uitvoerbaar bestandsformaat dat door een besturingssysteem uitgevoerd kan worden.

‘

dynamic of static gecompilede library - Twee manieren om software libraries te compilen.