Magnet™ Server Application Developer Guide

2.0

Revision A

Magnet Systems, Inc.

435 Tasso Street
Suite 100
Palo Alto, CA
94301

650–329–5904

info@magnet.com

Magnet

Contents

Magnet

1. About This Guide

The Magnet™ Server Application Developer Guide provides information and code samples for building applications on the Magnet Mobile Enterprise Server. It also provides instructions for uploading the generated project to a private cloud-based sandbox in the Magnet Developer Factory for testing.

Intended Audience

This guide is intended for developers who write Java-based server-side apps that interact with the Magnet Enterprise Server applications. This document assumes that you understand basic control flow of an application written in Java.

Document Organization

This guide includes these chapters:

- *Introducing Magnet Mobile App Server*, provides a high-level description of each component, its relationship, and interaction between the components.
- *Getting Started*, provides information that you need to gather before building your project. This information includes Maven environment, POM, Factory account set up, connected apps for the third-party services, and so forth.
- *Managing Your Project with Developer Factory*, provides instructions for creating, modifying, and deleting your project by way of Magnet Developer Factory.
- *Building a Service*, provides information and code samples for building a custom services in your application.
- *Working with Persistence*, provides information and code samples for reading and writing to the database.
- *Working with Transaction API*, provides information and code samples for making your services and persistence model enterprise ready.
- *Connecting to Third-Party Services*, provides information and instructions for connecting to third-party services, such as SalesForce, LinkedIn, and Facebook.

- *Compiling Your Project*, provides information and instructions for compiling your project.

Related Documents

The following table lists and describes other documents that are related to the Magnet products.

Document Title	Description
Magnet™ Android Developer Guide	Intended for Android app developers, this guide provides information and code samples for building enterprise native Android apps. It also provides instructions to upload the completed Android app to your private sandbox for testing.
Magnet™ iOS Developer Guide	Intended for iOS app developers, this guide provides information and code samples for building enterprise native iOS apps. It also provides instructions to upload the completed iOS app to your private sandbox for testing.
Magnet™ Mobile App Server Deployment Guide	Intended for IT administrators, this guide provides information and instructions for deploying the Magnet Mobile App Server to the Amazon cloud.
Magnet™ Mobile App Administrator Guide	Intended for IT administrators, this guide provides information and instructions for managing apps using the Magnet Mobile App Management Console.
Magnet™ Mobile for Android User Guide	Intended for users of Android devices, this guide provides information and instructions for installing the Magnet Mobile Server and using Apps@Work.
Magnet™ Mobile for iOS User Guide	Intended for users of iOS devices, this guide provides information and instructions for installing Setup@Work and using Apps@Work.

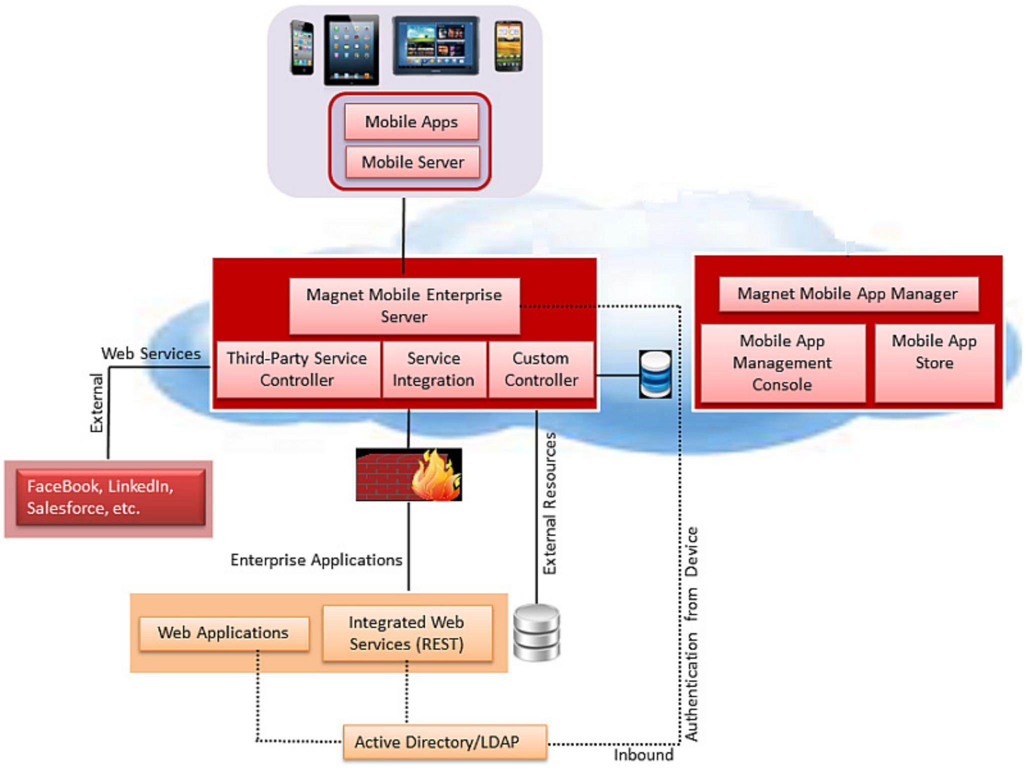
Magnet

2. Introducing Magnet Mobile App Server

The Magnet™ Mobile App Server provides basic constructs for creating and manipulating app objects spanning the server and mobile app contexts. The Magnet Mobile App Server:

- Enables data flow between users' devices and enterprise back-end systems, as well as monitoring usage of mobile apps.
- Supports performance, availability, scalability, and reporting.
- Ensures that security-sensitive data is encrypted locally on device storage.

The following diagram shows the Magnet Mobile App Server components and their relationships.



Mobile Apps
Mobile Server
Magnet Mobile Enterprise Server
Third-Party Service Controller
Service Integration
Custom Controller
Magnet Mobile App Manager
Mobile App Management Console
Mobile App Store
Web Services
External
FaceBook, LinkedIn, Salesforce, etc.
External Resources
Enterprise Applications
Web Applications
Integrated Web Services (REST)
Active Directory/LDAP
Authentication from Device
Inbound

Magnet Mobile Enterprise Server

Magnet Mobile Enterprise Server (MES) is a server that contains all business logic components in a single deployable JAR file. Each encapsulation of business logic is herein referred to as a Controller. Instances of the Magnet Mobile Enterprise Server are deployed in the cloud, and may contain a blend of off-the-shelf (OTS) and custom controllers that you write for your app.

Custom Controllers

Should you want to create your own custom controllers, you can download the source code for a reference server controller implementation. You can then modify this reference controller implementation to add your own business logic, then upload the modified project for your custom controller back into the Magnet Developer Factory so that this custom controller can be included into a build of the MES instance along with other controllers defined in your project. Custom controllers leverage services such as caching and off-line mode and persistence.

Caching and Off-line Message Delivery

To deal with unreliable connections between mobile devices and MES and to ensure reliable message delivery, Magnet Mobile App Server supports off-line operations that store and cache messages, and forwards information when the network connection is reestablished.

You can invoke these asynchronous services locally on the server or remotely from a mobile client. The invocations from the mobile client can occur asynchronously and reliably (reliability is an option). They are asynchronous in that the invocation is recorded for later playback to the server. They are reliable in that after an invocation has been submitted, it is guaranteed to be run once—and only once from the server perspective.

After entries are processed and results arrive back from the server, a listener that you optionally provide will be notified. Listeners are either transient or durable. A transient listener will be forgotten if the app JVM instance is cycled for any reason. Durable (non-transient) listeners will survive JVM restarts.

Persistence

Persistence is about the storage and retrieval of structured data. Magnet Mobile App Server provides support for entities and mapping entities to the data store. This creates, in effect, a “virtual entity data store” over any combination of MySQL, LDAP, and so forth, that can be used from within the programming interface Magnet provides.

Magnet Mobile App Server uses Java interfaces to define functions, methods, classes, and requests. Magnet Mobile App Server generates implementation from these classes built on annotations declared on interfaces, which allows for relationship, attributes, and so forth. Using user-friendly interfaces on entities, the Magnet Mobile App Server provides these features:

- Queries on entity with query composition
- CRUD (Create, Read, Update, Delete) operations on entities
- Complex queries such as paging and ordering

A developer can choose any technology (JDBC, Hibernate, etc.) to access a data store. However, implementing with Magnet persistence APIs is easier to use, and powerful over time. Every time an entity is used, unstructured data is collected. Over time, recommendations can be derived from that unstructured data. For example, when something is changed on an employee record, the implementation will track when that change is made and by whom. Of course, the recommendations will only be provided if you choose to incorporate those modules into your app. The entire methodology of building apps is modular - allowing you to blend the right combination of modules that make sense for your app.

Transactions

Magnet Mobile App Server can optionally use transactions to maintain the integrity of data.

Service Integration

Service integration can automatically transform Web applications from legacy enterprise application servers to the Magnet Mobile Enterprise Server. It exposes traditional SOAP-based Web Services hosted in the enterprise application servers to REST-based services for mobile access. Using the free on-line Developer Factory, you create a project with specific requirements, and download the generated APIs that abstract SOAP/REST services as controllers. Once generated, you can customize them as you see fit.

Third-Party Service Controllers

The third-party service controllers are sample controllers that allow you to connect to well-known social services, such as Facebook, LinkedIn, or Salesforce. These controllers have built-in support for OAuth, so you can securely retrieve contacts information, and share an update. The source code for these controllers is included in the project generated by the Developer Factory if you choose to include them in your project.

Magnet Mobile App Manager

The Magnet™ Mobile App Manager (MAM) is a run-time server that monitors the service status of Magnet MES and provides app management functionality by way of the Magnet Mobile App Management Console and Mobile App Store.

Mobile App Management Console

The Magnet MAM Console is a Web interface of the Magnet Mobile App Manager that enables IT administrators to centrally manage and administer apps on client devices. Access to the MAM Console is defined by role-based and group association, which is defined in LDAP. Using the MAM Console, IT administrators can:

- Enable employee BYOD.
- Remotely install, remove, upgrade, and track apps.
- Customize an enterprise private-label app store.
- Remotely activate / deactivate user accounts.
- Push required apps to authorized users.
- View logs to monitor events history.

Mobile App Store

The Mobile App Store is an app for installing and managing apps on a mobile device and is installed on iOS and Android devices. The Mobile App Store includes a suite of sample apps that are organized in categories. Via the MAM Console, IT administrators can brand the Mobile App Store with the enterprise private label. The App Store branding kit can be obtained on request.

Deployment

An executable JAR file contains all controllers required for a specific deployment. Instances of the Magnet Mobile Enterprise Server are deployed in the Amazon cloud —a virtual private cloud that is managed by customers.

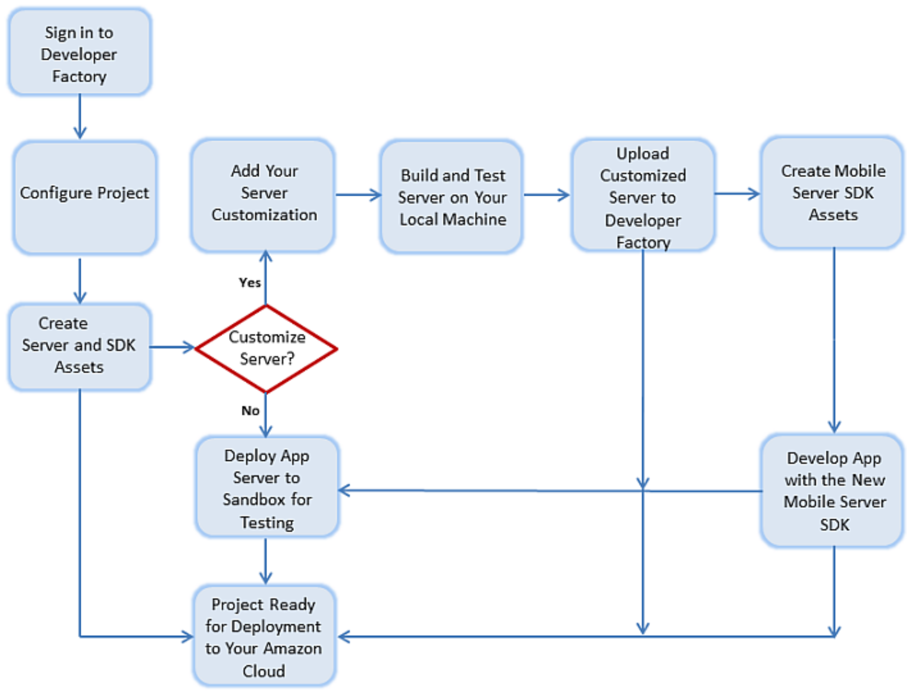
Developer Factory

The Developer Factory is a free on-line service that provides a secure and private place for you to dynamically construct controller interfaces specific to the back-end services required for use in your mobile apps. The Developer Factory empowers you to

- Create and manage your app projects.
- Generate customized controllers and runtime binaries for your projects.
- Obtain source code for all your customized project assets.
- Upload modified controller and entities source codes for custom app controllers to regenerate executable JAR files.
- Obtain binaries for additional components required to deploy your projects in the Amazon cloud. Automatically create other language packs of your controllers (for example, iOS/XCode).
- Test-drive your app projects by getting access to your own private sandbox environment in the cloud.

Flow Process

As shown in the following diagram, the process of building your project begins when you sign in to the Developer Factory.


Sign in to Developer Factory
Configure Project
Add Your Server Customization
Build and Test Server on Your Local Machine
Upload Customized Server to Developer Factory
Create Mobile Server SDK Assets
Create Server and SDK Assets
Customize Server?
Yes
No
Deploy App Server to Sandbox for Testing
Develop App with the New Mobile Server SDK
Project Ready for Deployment to Your Amazon Cloud

Magnet

3. Getting Started

The Magnet Mobile App Server uses Apache Maven, an open source tool used for building and assembling application.

- Java 1.7 or later for server-based components; Java 1.6 for Android-based applications.
- Maven 3.0.3 or later.

If you are using a recent version of MacOS, Maven may be already installed. For a free download and installation, visit

`http://maven.apache.org/download.html`

Maven uses a Project Object Model (POM) file to describe the software project being built, its dependencies on other external modules and components, an Internet-based repository to automatically retrieve OTS modules produced by Magnet or other open source contributors as needed, the build order, directories, and required plug-ins. The POM file contains everything needed to describe a project. This file is automatically created when creating a Maven project from the Developer Factory.

Setting Up Your Repository Environment

You can set up the repository using one of the following methods:

- create a settings.xml file
- declare in the POM file

Creating a settings.xml File

Place the *settings.xml* file in the **.m2** directory. An example of the settings.xml file is shown.

```
<settings>
    <profiles>
        <profile>
            <id>magnet</id>
            <repositories>
                <repository>
                    <id>maven_magnet</id>
                    <name>Public Magnet Maven Repository</name>
                    <url>http://developer.magnet.com/nexus/content/
groups/public/</url>
                </repository>
            </repositories>
        </profile>
    </profiles>
    <activeProfiles>
        <activeProfile>magnet</activeProfile>
    </activeProfiles>
</settings>
```

Declaring in the POM File

You can declare the repository for your project by adding the following information to your POM file.

```
<project ...>
    ...
    <repositories>
        ...
        <repository>
            <id>maven_magnet</id>
            <name>Public Magnet Maven Repository</name>
            <url>http://developer.magnet.com/nexus/content/
groups/public/</url>
        </repository>
    </repositories>
    ...
</project>
```

Visit http://maven.apache.org/ or http://maven.apache.org/guides/getting-started/ for getting additional information about Maven.

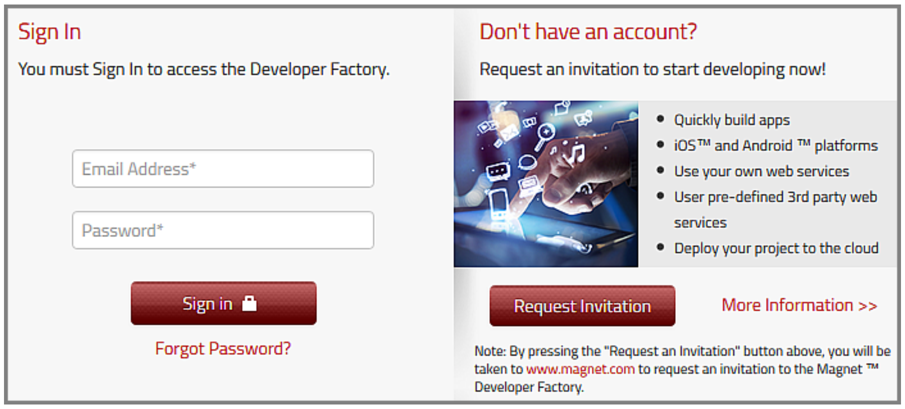
Setting up Your Account in Developer Factory

The Developer Factory is a free on-line service that provides a secure and private place for you to dynamically construct controller interfaces specific to the back-end services required for use in your mobile apps and on your server instance(s).

To request a Developer Factory account:

1. Launch a browser and enter the following URL:

```
http://factory.magnet.com
```



Sign In

You must Sign In to access the Developer Factory.

Email Address*

Password*

Sign in

Forgot Password?

Don't have an account?

Request an invitation to start developing now!

- Quickly build apps
- iOS™ and Android ™ platforms
- Use your own web services
- User pre-defined 3rd party web services
- Deploy your project to the cloud

Request Invitation

More Information >>

Note: By pressing the "Request an Invitation" button above, you will be taken to www.magnet.com to request an invitation to the Magnet ™ Developer Factory.

2. Click **Request Invitation** to start the process.

Choosing Magnet Mobile Enterprise Server Location

- The IP of the Magnet Mobile Enterprise Server must be accessible from outside the network.
- This IP must be static.

Creating Connected App on SalesForce Developer Console

You need to set up the connected app only when you are including SalesForce in your application. When the connected app is created, both consumer key and consumer secret are generated, and are used to authenticate your application to SalesForce. Make a note of them as you will need them for setting up the Magnet Mobile Enterprise Server as well as the SOAP API endpoint URL specific to your Salesforce connected app.

You need a Salesforce developer account to create a connected app (visit http://developer.force.com/ for information).

The Callback URL points to your Magnet Enterprise Server OAuth Servlet, for example, https://ec2-184-73-100-147.compute-1.amazonaws.com:8081/MagnetOAuthServlet.

Creating Connected App for LinkedIn

You need to create a connected app only when you are including LinkedIn in your application. When the connected app is created, a client ID and a client secret are generated, and are used to authenticate your application to LinkedIn. Make a note of them as you will need them for setting up your Magnet Mobile Enterprise Server as well as the API endpoint URL specific to your LinkedIn connected app.

You need a LinkedIn developer account to create a connected app (visit http://developer.linkedin.com/ for information).

Creating Connected App for Facebook

You need to create a connected app only when you are including Facebook in your application. When the connected app is created, a client ID and a client secret are generated, and are used to authenticate your application to Facebook. Make a note of them as you will need them for setting up your Magnet Mobile Enterprise Server as well as the API endpoint URL specific to your Facebook connected app.

You need a FaceBook developer account to create a connected app (visit https://developers.facebook.com/docs/appcenter/).

Magnet

4. Managing Your Project with Developer Factory

The Developer Factory provides an intuitive environment enabling you to easily build and manage your project. As an owner of a project that you create, you can:

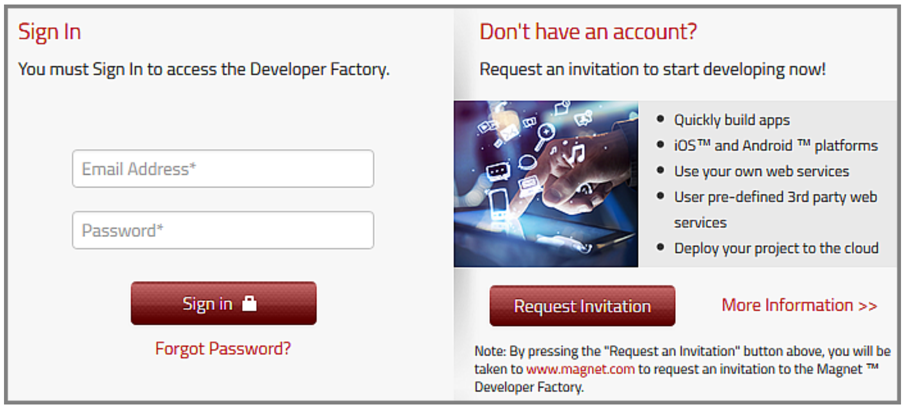
- Create a project (page *14*)
- Deploy your project for testing or download it for customization (page *21*)
- Modify your project (page *22*)
- Upload a customized project (page *24*)
- Remove your project (page *25*)
- Invite other developers (page *27*)

Creating a Project

The Magnet Developer Factory provides wizard that walks you through the creation of a project.

1 Launch a browser and enter the following URL:

```
http://factory.magnet.com
```



Sign In

You must Sign In to access the Developer Factory.

Email Address*

Password*

Sign in

Forgot Password?

Don't have an account?

Request an invitation to start developing now!

- Quickly build apps
- iOS™ and Android ™ platforms
- Use your own web services
- User pre-defined 3rd party web services
- Deploy your project to the cloud

Request Invitation

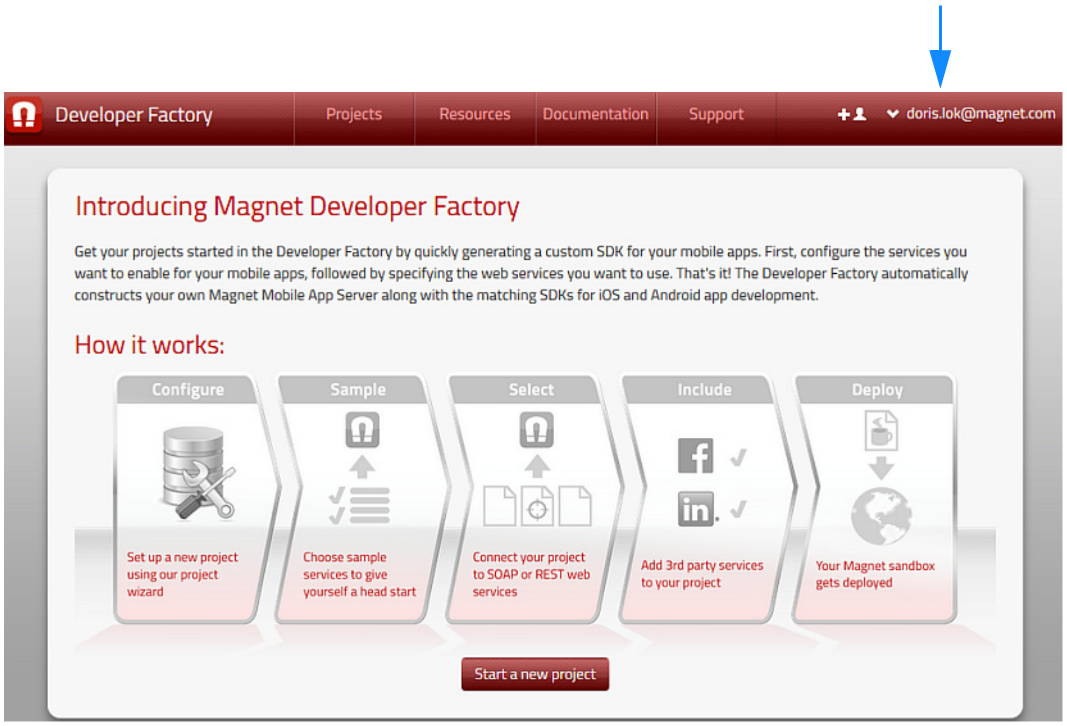
More Information >>

Note: By pressing the "Request an Invitation" button above, you will be taken to www.magnet.com to request an invitation to the Magnet ™ Developer Factory.

2 Enter your user credentials information in the appropriate fields:

3 Click **Sign in**.

The page with your user name displayed opens, as shown in the following example.



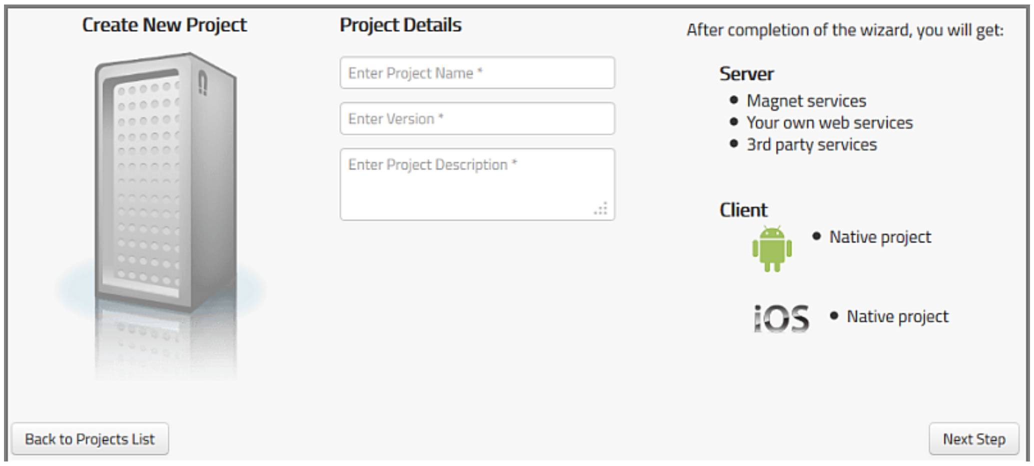
Introducing Magnet Developer Factory

Get your projects started in the Developer Factory by quickly generating a custom SDK for your mobile apps. First, configure the services you want to enable for your mobile apps, followed by specifying the web services you want to use. That's it! The Developer Factory automatically constructs your own Magnet Mobile App Server along with the matching SDKs for iOS and Android app development.

How it works:

Configure
Set up a new project using our project wizard
Sample
Choose sample services to give yourself a head start
Select
Connect your project to SOAP or REST web services
Include
Add 3rd party services to your project
Deploy
Your Magnet sandbox gets deployed
Start a new project

4 Click **Start a new project.**



After completion of the wizard, you will get:

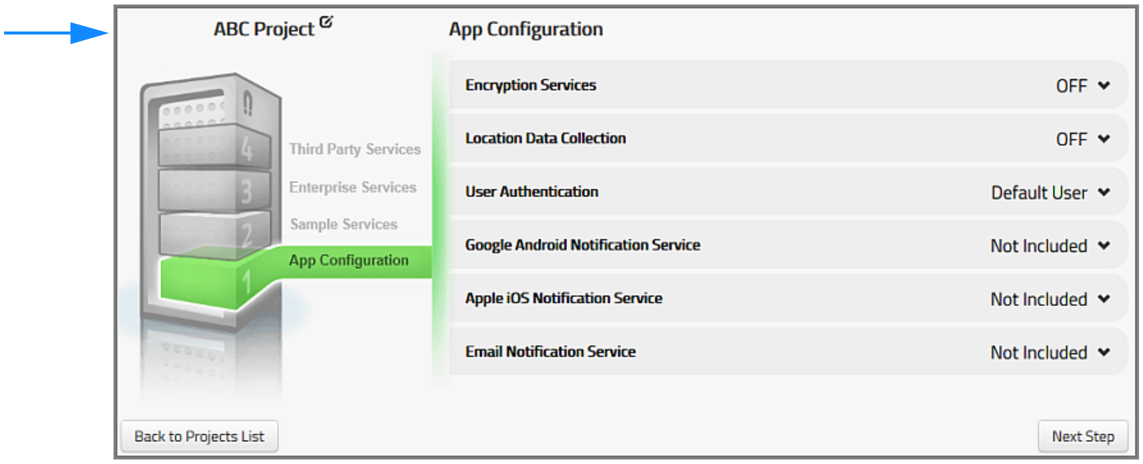
Server
- Magnet services
- Your own web services
- 3rd party services

Client
- Native project
- Native project

5 Enter the name of your project, version number, and a brief description in the appropriate fields.

6 Click **Next Step**.

The name of the project you created is displayed as shown in the following example.



Encryption Services	OFF
Location Data Collection	OFF
User Authentication	Default User
Google Android Notification Service	Not Included
Apple iOS Notification Service	Not Included
Email Notification Service	Not Included

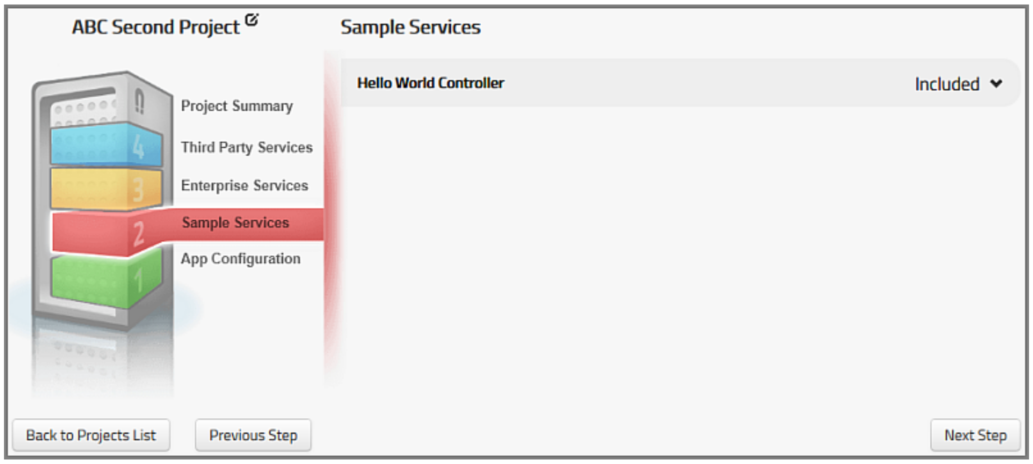
7 Click to select the following configuration information:

- *Encryption Services*. This is used to encrypt security sensitive information in your configuration as well as on your mobile devices.
- *Location data collection*: This enables geo-tagged coordinates to be transported each time a controller is invoked on your server, allowing your server to know the location of the mobile app user.

- *User Authentication*: You can select the appropriate data store from where you want to read user credentials to authenticate for accessing server from the mobile clients.
  - — Select Default User if you want the Developer Factory to provide a user name and password for you that will get you access to a deployed sandbox.
  - — Select the appropriate option if you want to use your own tools and environment to control access to the sandbox. User name and password will be managed by one of these three systems.

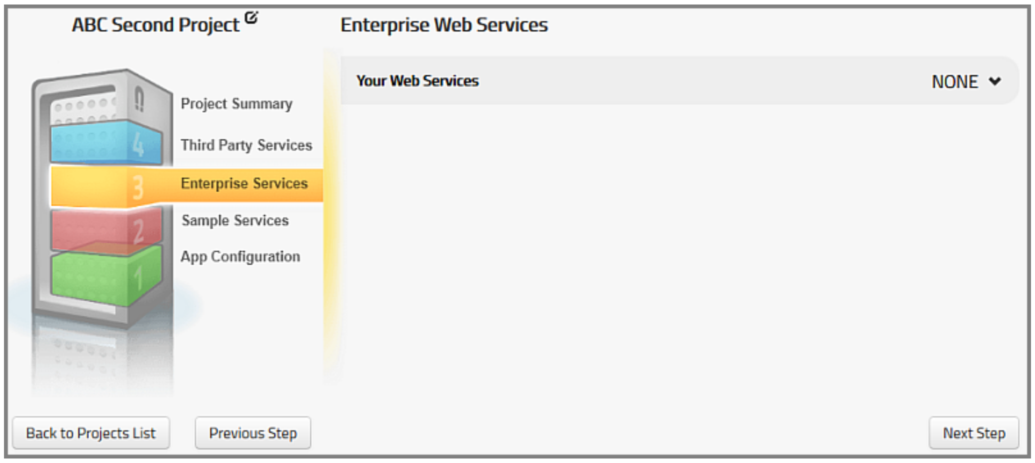
8. Click **Include** to enable these services so you can utilize those notification capabilities for your apps.
   - Google Android Notification Service
   - Apple iOS Notification Service
   - Email Notification Service
9. Click **Next Step** to select the sample services.
10. Include the Hello World Controller if you want to get a template to start developing your own custom controllers. However, if you do not want to add your own custom controllers, then click **Don't Include**.

The Hello World Controller provide a simple UI that can be run on an emulator or on a phone and connect to the deployed sandbox (or any server where you may have installed our generated server JAR).

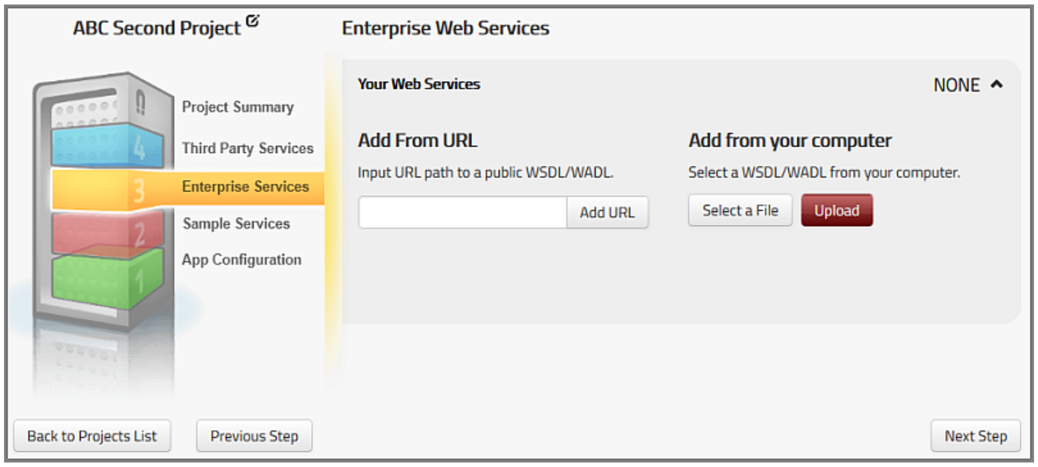


Hello World Controller — Included

11 Click **Next Step** to add your Web services.



12 Click **NONE** to display the Web Services options.



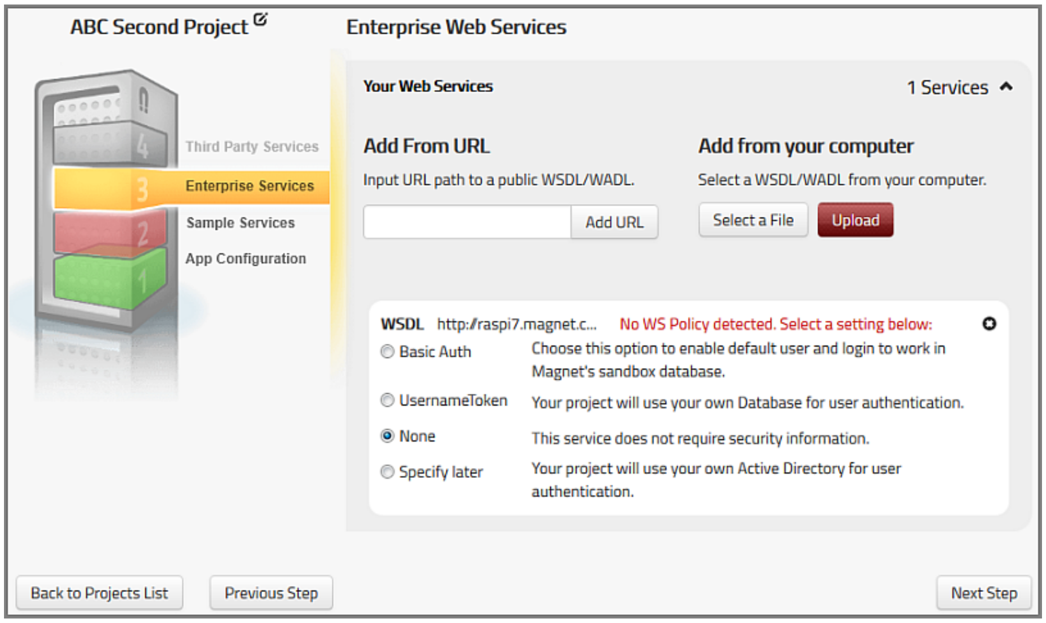
13 Enter the URL to add your WSDL or WADL file, click **Add URL**.

— Or —

Click **Select a File** to add your WSDL or WADL file from your computer, click **Upload.**

14 Configure security settings for your app to properly communicate with this Web service.

The security configuration window will not appear if security configuration is included in your WSDL file.



Enterprise Web Services

Your Web Services — 1 Services

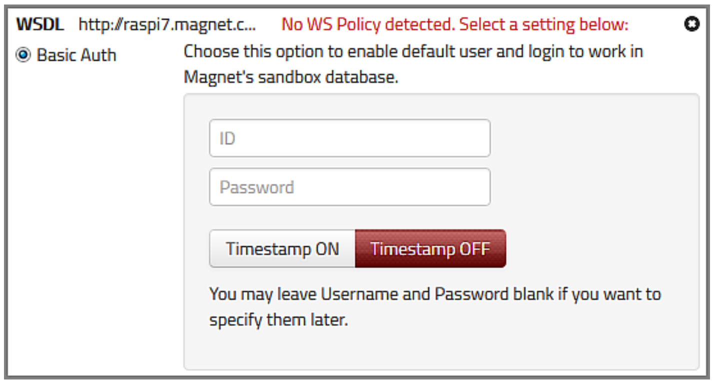
Add From URL
Input URL path to a public WSDL/WADL.
Add URL

Add from your computer
Select a WSDL/WADL from your computer.
Select a File | Upload

WSDL http://raspi7.magnet.c... No WS Policy detected. Select a setting below:

Basic Auth	Choose this option to enable default user and login to work in Magnet's sandbox database.
UsernameToken	Your project will use your own Database for user authentication.
[✓ None]	This service does not require security information.
Specify later	Your project will use your own Active Directory for user authentication.

- When Basic Auth is selected, the following window appears:



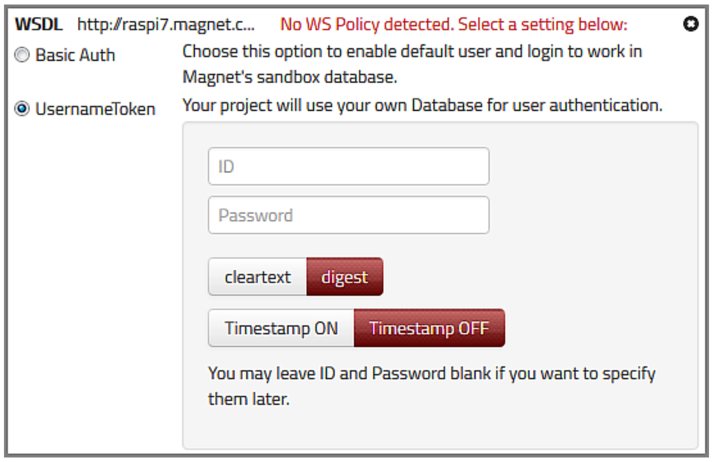
WSDL http://raspi7.magnet.c... No WS Policy detected. Select a setting below:

[✓ Basic Auth] Choose this option to enable default user and login to work in Magnet's sandbox database.

ID
Password
Timestamp ON | Timestamp OFF

You may leave Username and Password blank if you want to specify them later.

- When UsernameToken is selected, the following window appears:



WSDL http://raspi7.magnet.c... No WS Policy detected. Select a setting below:

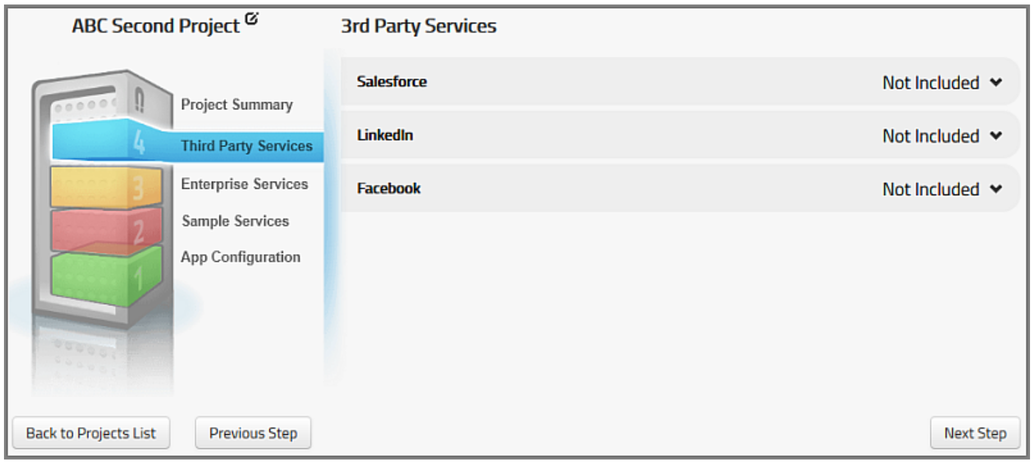
Basic Auth — Choose this option to enable default user and login to work in Magnet's sandbox database.

[✓ UsernameToken] Your project will use your own Database for user authentication.

ID
Password
cleartext | digest
Timestamp ON | Timestamp OFF

You may leave ID and Password blank if you want to specify them later.

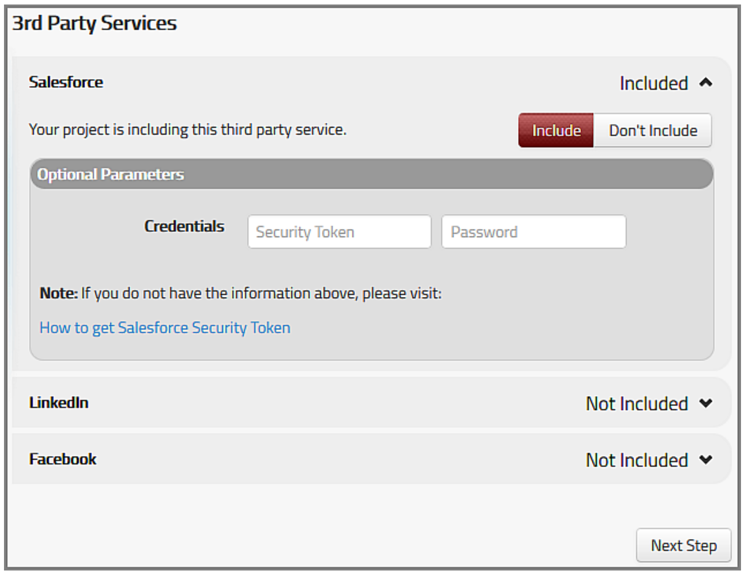
15 Click **Next Step** to display the *3rd Party Services* page.



3rd Party Services

Salesforce	Not Included
LinkedIn	Not Included
Facebook	Not Included

16 Select the third-party services that you want to include in your application, click **Include**.



3rd Party Services

Salesforce — Included

Your project is including this third party service.

Optional Parameters

Credentials: Security Token, Password

Note: If you do not have the information above, please visit:

How to get Salesforce Security Token

LinkedIn — Not Included

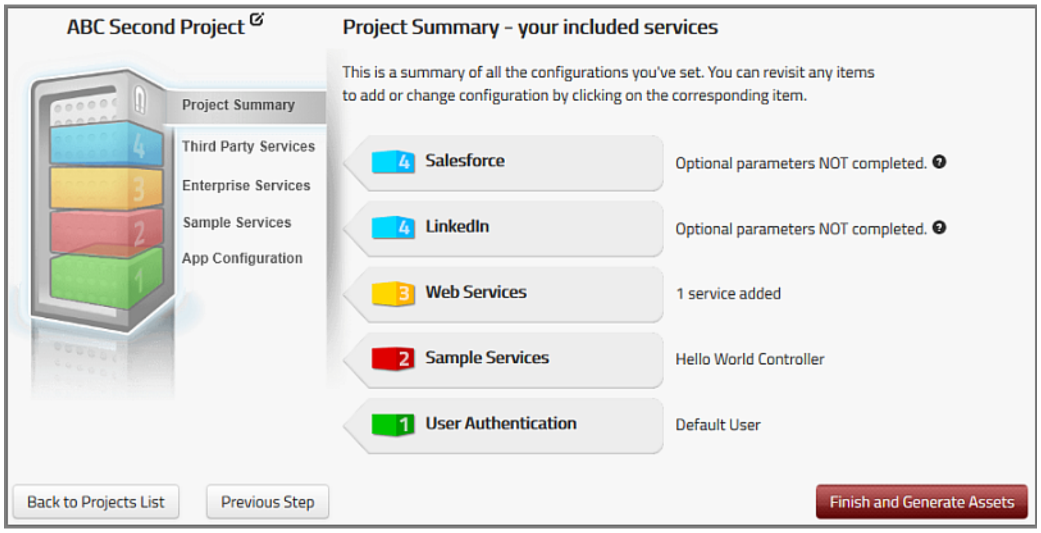
Facebook — Not Included

17 Optionally enter the credentials and password (you received those from Connected Apps) in each of the services you include.

You can configure these settings later in your generated project.

18 Click **Next Step**.

Your project summary is displayed as shown in the following example.



ABC Second Project

Project Summary – your included services

This is a summary of all the configurations you've set. You can revisit any items to add or change configuration by clicking on the corresponding item.

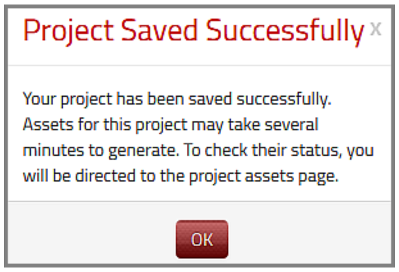
4 Salesforce	Optional parameters NOT completed.
4 LinkedIn	Optional parameters NOT completed.
3 Web Services	1 service added
2 Sample Services	Hello World Controller
1 User Authentication	Default User

The color-coded buttons shown on the Project Summary side are now interactive. Clicking them takes you directly to the intended page. For example, clicking User Authentication opens the App Configuration step.

19 Review your project.

- To make changes, click **Previous Step** to the step that you need changing.
- Click **Finish and Generate Assets** if no further changes are needed.

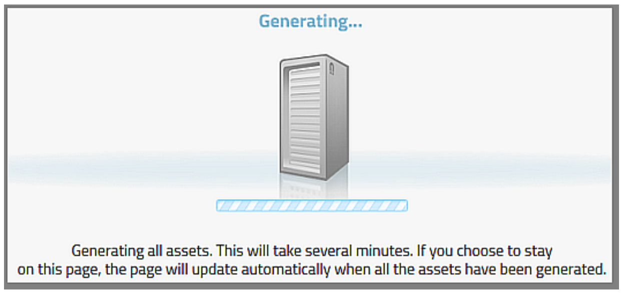
A dialog box appears informing of the project status.



Project Saved Successfully

Your project has been saved successfully. Assets for this project may take several minutes to generate. To check their status, you will be directed to the project assets page.

20 Click **OK** to start the asset generation process.



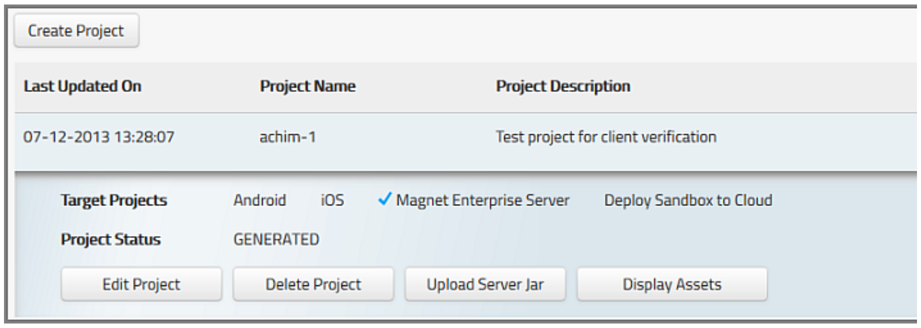
Generating...

Generating all assets. This will take several minutes. If you choose to stay on this page, the page will update automatically when all the assets have been generated.

Displaying the Generated Assets

After your project is generated, you can download it to your local machine for customization or deploy it to the sandbox for testing.

1 Click the **Projects** tab, and select the intended project.



Create Project

Last Updated On	Project Name	Project Description
07-12-2013 13:28:07	achim-1	Test project for client verification

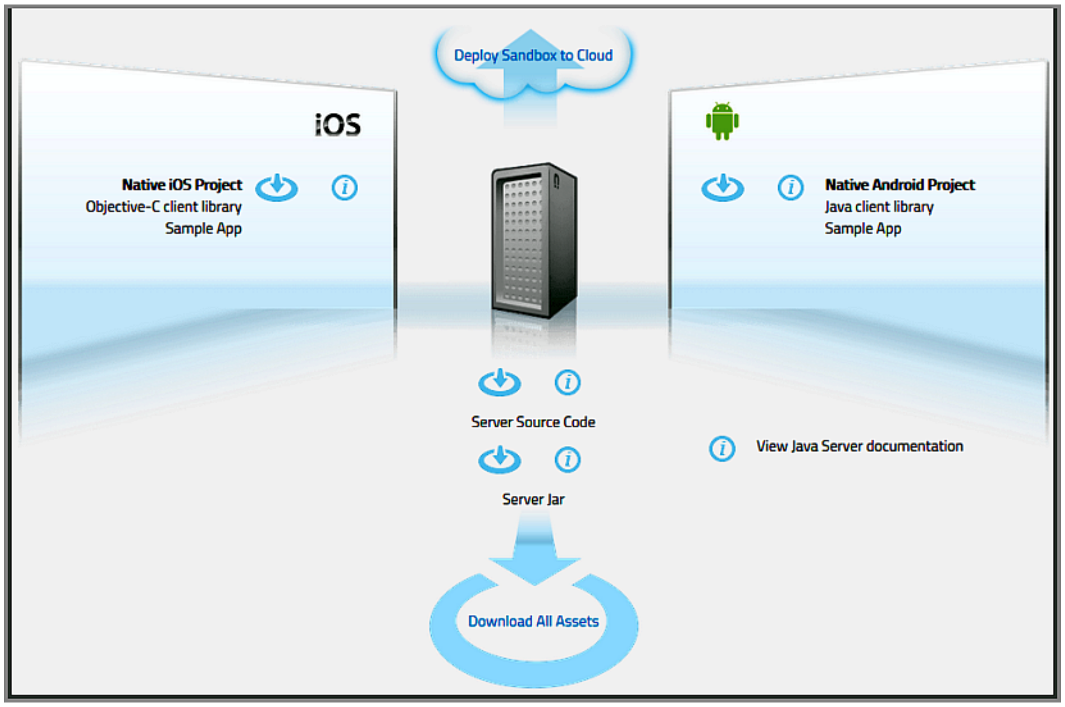
Target Projects: Android iOS ✓ Magnet Enterprise Server Deploy Sandbox to Cloud

Project Status: GENERATED

Edit Project | Delete Project | Upload Server Jar | Display Assets

2 Click **Display Assets**.

You are informed if the project has not been generated. Click **Generate Assets**, then repeat step 2.



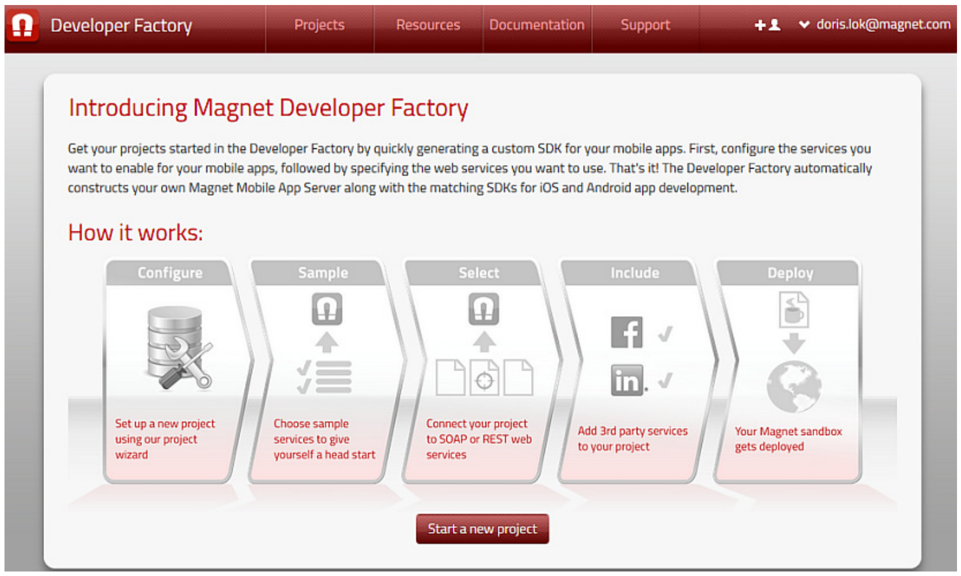
3 Click **Deploy Sandbox to Cloud**, or select the project you want to download, or click **Download All Assets** to download all assets.

Each server project is compressed into a zip file.

Modifying a Project

Do not change the maven groupId, artifactId, and version of your project if you plan to regenerate the mobile assets out of a modified project.

1 Sign in to your Developer Factory account.



Introducing Magnet Developer Factory

Get your projects started in the Developer Factory by quickly generating a custom SDK for your mobile apps. First, configure the services you want to enable for your mobile apps, followed by specifying the web services you want to use. That's it! The Developer Factory automatically constructs your own Magnet Mobile App Server along with the matching SDKs for iOS and Android app development.

How it works:

Set up a new project using our project wizard

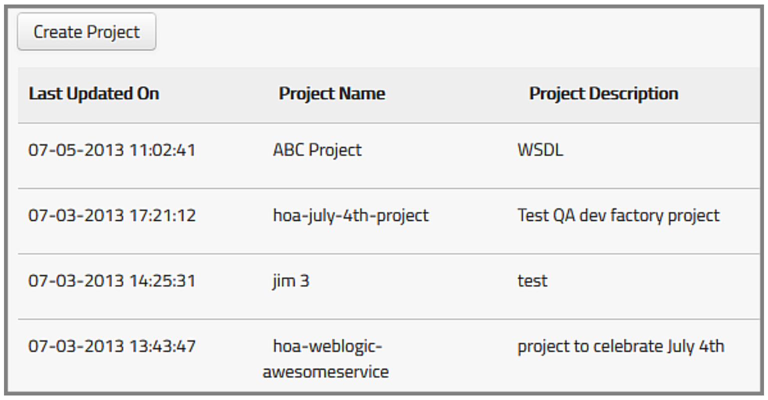
Choose sample services to give yourself a head start

Connect your project to SOAP or REST web services

Add 3rd party services to your project

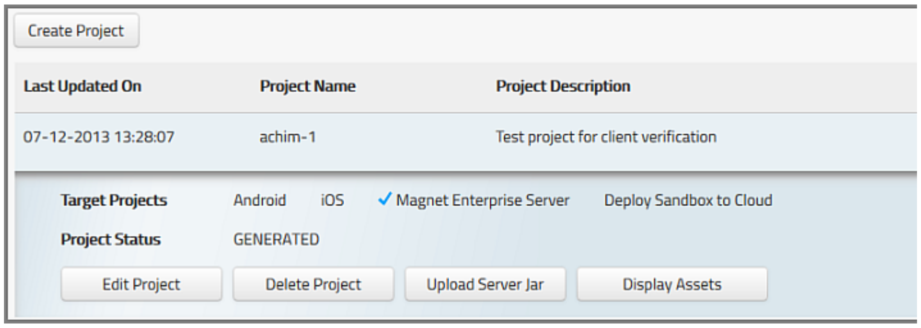
Your Magnet sandbox gets deployed

2 Click the **Projects** tab to display all projects that you have created, as shown in the following example.



Last Updated On	Project Name	Project Description
07-05-2013 11:02:41	ABC Project	WSDL
07-03-2013 17:21:12	hoa-july-4th-project	Test QA dev factory project
07-03-2013 14:25:31	jim 3	test
07-03-2013 13:43:47	hoa-weblogic-awesomeservice	project to celebrate July 4th

3 Select the project you want to modify.



Create Project

Last Updated On	Project Name	Project Description
07-12-2013 13:28:07	achim-1	Test project for client verification

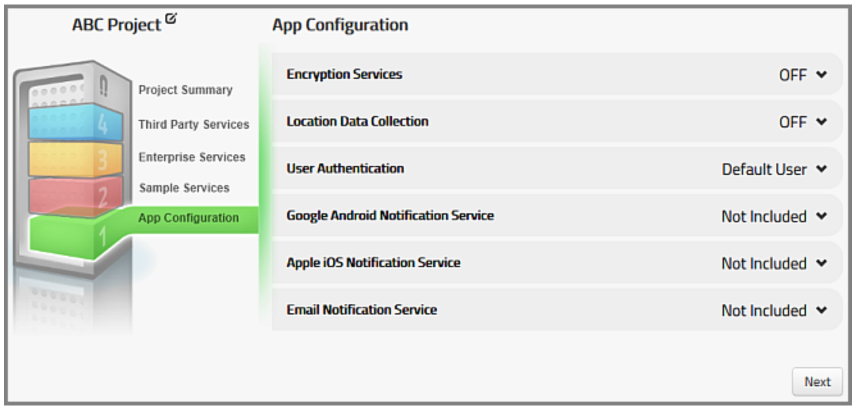
Target Projects: Android, iOS, ✓ Magnet Enterprise Server, Deploy Sandbox to Cloud

Project Status: GENERATED

4 Click **Edit Project**.

A page similar to the following example appears.

All color-coded buttons are now interactive. Clicking them takes you directly to the intended page.



App Configuration

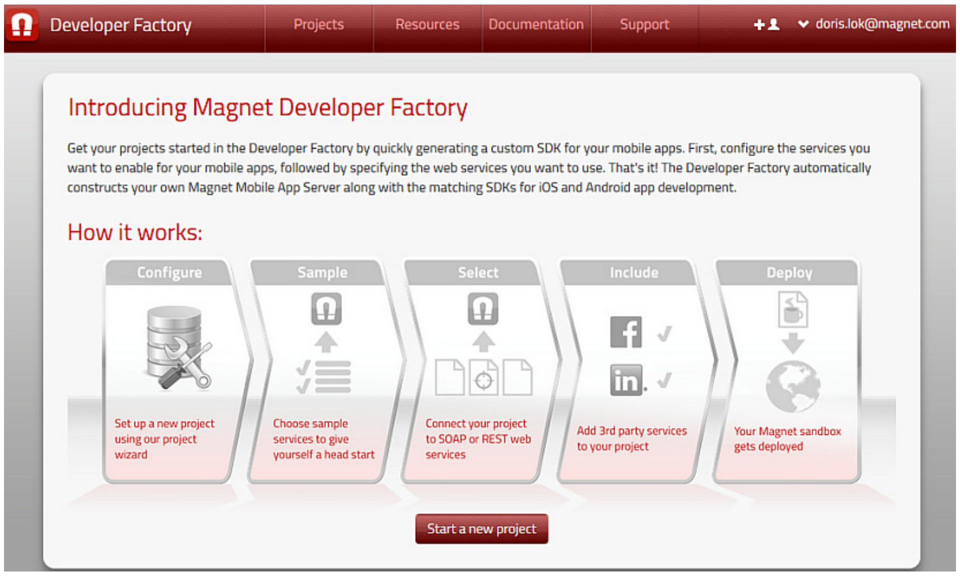
Encryption Services	OFF
Location Data Collection	OFF
User Authentication	Default User
Google Android Notification Service	Not Included
Apple iOS Notification Service	Not Included
Email Notification Service	Not Included

5 Click the intended color-coded services layers and modify any information (refer to ***Creating a Project*** for detailed description).

Uploading a Project

When your project customization is complete, you need to upload the project from your local machine to the Developer Factory, and regenerate the assets.

1 Sign in to your Developer Factory account.



Introducing Magnet Developer Factory

Get your projects started in the Developer Factory by quickly generating a custom SDK for your mobile apps. First, configure the services you want to enable for your mobile apps, followed by specifying the web services you want to use. That's it! The Developer Factory automatically constructs your own Magnet Mobile App Server along with the matching SDKs for iOS and Android app development.

How it works:

Set up a new project using our project wizard

Choose sample services to give yourself a head start

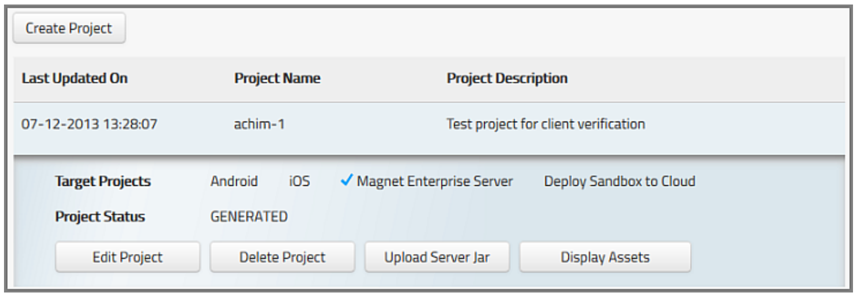
Connect your project to SOAP or REST web services

Add 3rd party services to your project

Your Magnet sandbox gets deployed

2 Click the **Projects** tab to display all projects that you have created.

3 Select the project you want to upload.

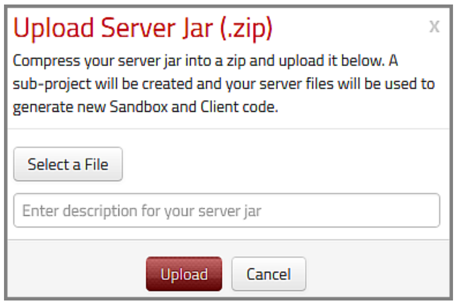


Last Updated On	Project Name	Project Description
07-12-2013 13:28:07	achim-1	Test project for client verification

Target Projects: Android iOS ✓ Magnet Enterprise Server Deploy Sandbox to Cloud

Project Status: GENERATED

4 Click **Upload Server Jar**.



Upload Server Jar (.zip)

Compress your server jar into a zip and upload it below. A sub-project will be created and your server files will be used to generate new Sandbox and Client code.

5 Click **Select a File**, and navigate to the *aws-Server/target* directory where the server JAR zip is stored.

Refer to ***Chapter 9, Compiling Your Project***, for the directory structure of the project.

6 Enter an optional description for the server JAR.

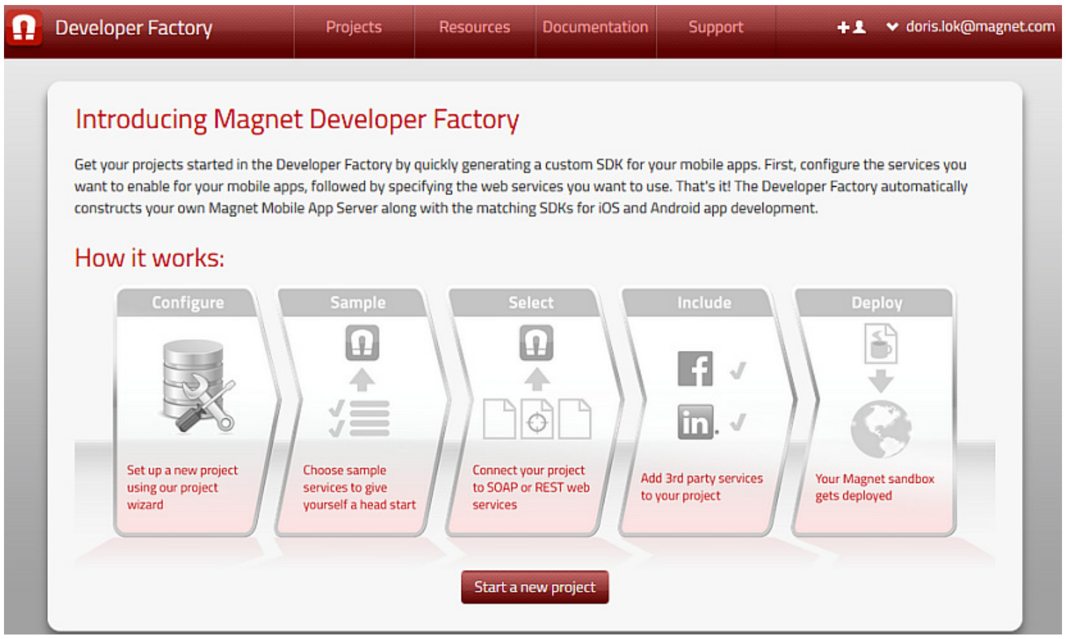
7 Click **Upload**.

Removing a Project

If you close your Developer Factory account, all projects that you owned will be automatically removed, and you will be removed from all projects to which you had been invited.

This section provides instructions to remove a project.

1 Sign in to your Developer Factory account.

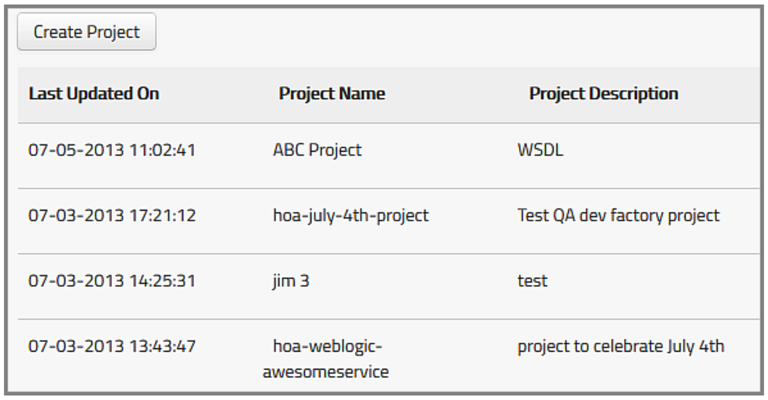

Developer Factory
Projects
Resources
Documentation
Support
doris.lok@magnet.com
Configure
Sample
Select
Include
Deploy
Set up a new project using our project wizard
Choose sample services to give yourself a head start
Connect your project to SOAP or REST web services
Add 3rd party services to your project
Your Magnet sandbox gets deployed
Start a new project

Introducing Magnet Developer Factory

Get your projects started in the Developer Factory by quickly generating a custom SDK for your mobile apps. First, configure the services you want to enable for your mobile apps, followed by specifying the web services you want to use. That's it! The Developer Factory automatically constructs your own Magnet Mobile App Server along with the matching SDKs for iOS and Android app development.

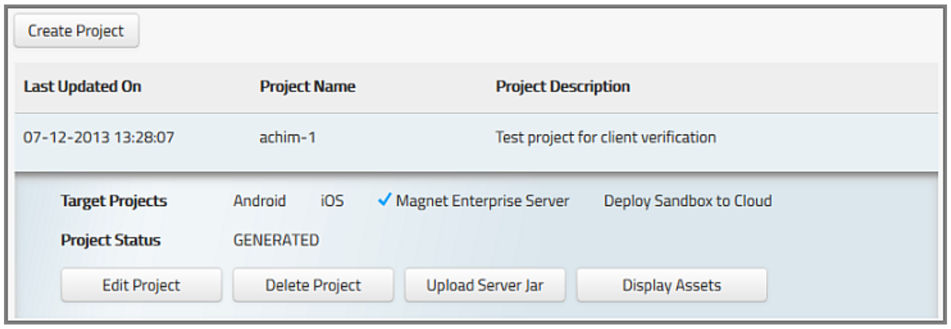
How it works:

2 Click the **Projects** tab to display all projects that you have created, as shown in the following example.



Last Updated On	Project Name	Project Description
07-05-2013 11:02:41	ABC Project	WSDL
07-03-2013 17:21:12	hoa-july-4th-project	Test QA dev factory project
07-03-2013 14:25:31	jim 3	test
07-03-2013 13:43:47	hoa-weblogic-awesomeservice	project to celebrate July 4th

3 Select the project you want to remove.



Last Updated On	Project Name	Project Description
07-12-2013 13:28:07	achim-1	Test project for client verification

Target Projects Android iOS ✓ Magnet Enterprise Server Deploy Sandbox to Cloud

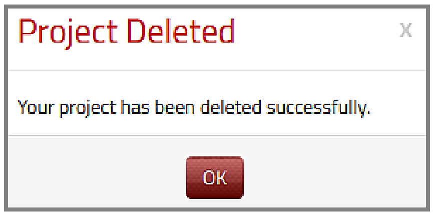
Project Status GENERATED

4 Click **Delete Project**.

A dialog box appears asking for confirmation.

5 Click **Continue**.

A dialog box appears showing the operation status.



Project Deleted

Your project has been deleted successfully.

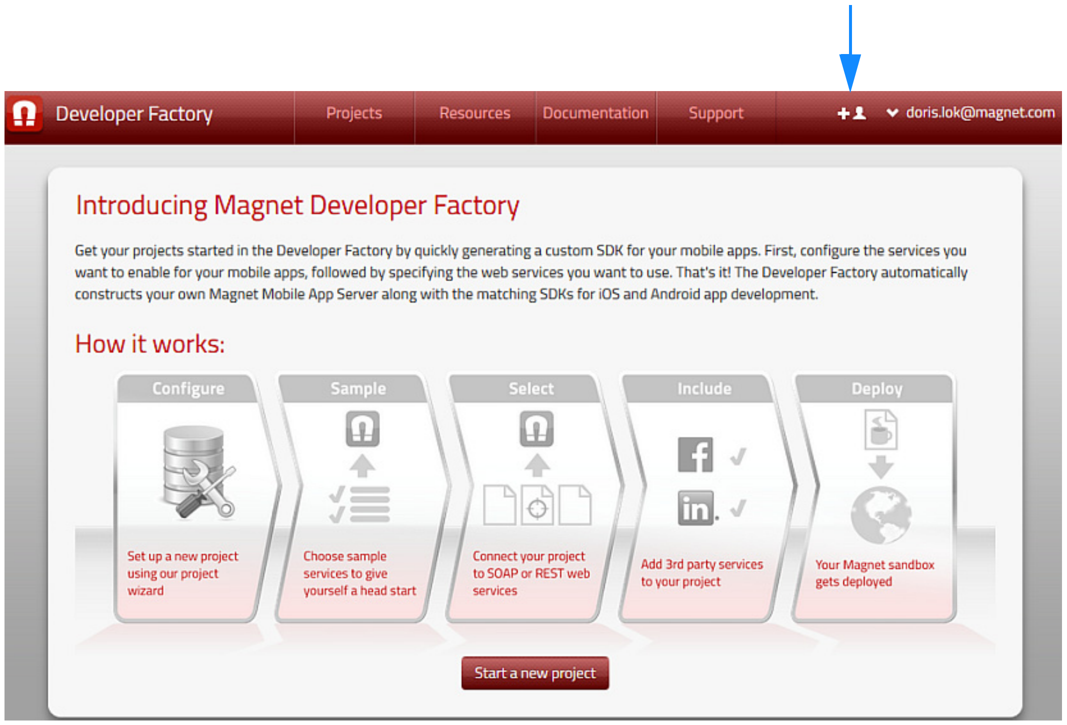
6 Click **OK**.

The project is immediately removed from the project page.

Inviting Other Developers

You can invite other developers to the Magnet Developer Factory to create their own accounts. If your invitees are not registered with Developer Factory, they need to first register for an account.

1 Sign in to your Developer Factory account.

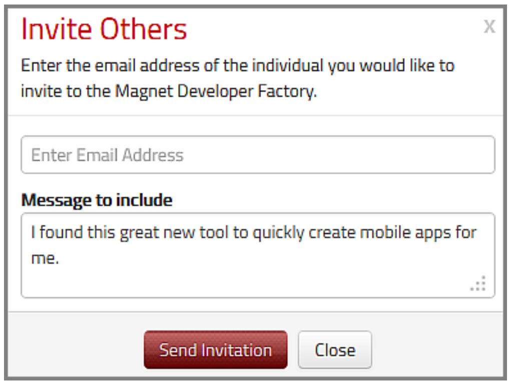

Developer Factory
Projects
Resources
Documentation
Support
+👤
doris.lok@magnet.com
Configure
Sample
Select
Include
Deploy
Set up a new project using our project wizard
Choose sample services to give yourself a head start
Connect your project to SOAP or REST web services
Add 3rd party services to your project
Your Magnet sandbox gets deployed
Start a new project

Introducing Magnet Developer Factory

Get your projects started in the Developer Factory by quickly generating a custom SDK for your mobile apps. First, configure the services you want to enable for your mobile apps, followed by specifying the web services you want to use. That's it! The Developer Factory automatically constructs your own Magnet Mobile App Server along with the matching SDKs for iOS and Android app development.

How it works:

2 Click +👤 to display the invitation, as shown in the following:



Invite Others

Enter the email address of the individual you would like to invite to the Magnet Developer Factory.

Message to include

I found this great new tool to quickly create mobile apps for me.

3 Enter the e-mail address of the person you want to invite, and optionally delete the existing message from the **Message to include** box, and enter your own message that you want to include with your e-mail.

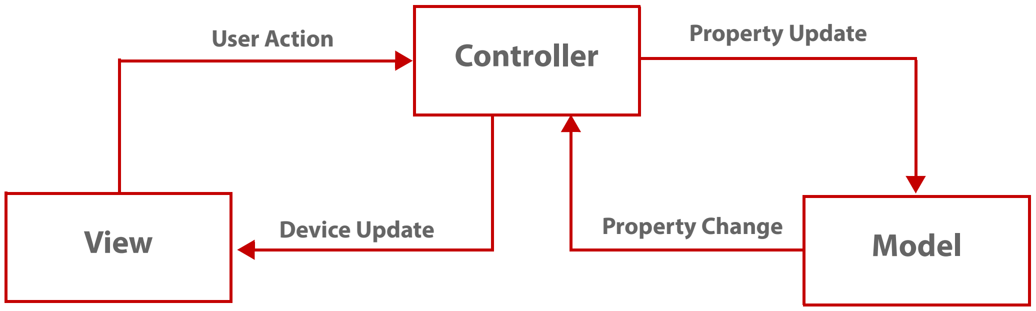
4 Click **Send Invitation**.

You are informed of the operation status. Your recipient will receive an e-mail notification that includes a link to register an account.

Magnet

5. Building a Service

The Magnet Mobile Enterprise Server exposes the Model–View–Controller (MVC) pattern for building apps, allowing developers to encapsulate business logic in controllers and persist data through entities.



The Model object represents the data of the application and the business rules used to manipulate and process that data. Much of the data that is part of the persistent state of the application resides in the model objects after the data is loaded into the application. Because model objects represent knowledge and expertise related to a specific problem domain, they can be reused in similar problem domains.

Magnet Mobile Enterprise Server uses the notion of “Entity” as the Model object. The Magnet approach to modeling entities is based on a fundamentally RESTul approach. Each entity has a unique URI. These identifiers are used to perform all traditional CRUD (Create, Retrieve, Update, and Delete) operations and the semantics are easily-mapped onto REST.

The View object is an application that users can see. The View object corresponds to elements of the user interface such as text, check-box items, and so forth. A major purpose of the View object is to display data from the application’s model objects and to enable the editing of that data.

The Controller object is the business logic that can perform setup and coordinating tasks for an application and manage the life cycles of other objects. In Magnet Mobile Enterprise Server, controllers are Java classes that access various services configured in your application and make them remotable. One of the most common functions of controllers is to coordinate changes to one or more of the entities in the Model object, from Java-based controllers plus inclusion of other language pack generation plugins, libraries can be created for your controllers that can be included and used in other platform runtime environments such as iOS or Javascript.

Magnet Mobile Enterprise Server makes extensive use of Java annotations. The following table lists and describes different annotations that are commonly used in services that are declared in Magnet Mobile Enterprise Server.

Annotations	Description
@Contract	Declare an injectable service.
@RemoteAccessible	Make the service interface remotely accessible over REST. Currently this is only applicable for Controller type services.
@Inject	Inject a service.
@Optional	Make the injected service optional.
@Singleton	Implement a singleton service.
@ConfigBean	Initialize a service that requires use-based configuration.

Declaring a Service with @Contract

When you write a service, you advertise its interfaces with the @Contract annotation. A Contract is declared with annotations rather than external descriptors. The advantage is that it's no longer necessary to constantly ensure that your code is synchronized with external files. The following shows use of the @Contract annotation to advertise a service (Adder).

```
@Contract
public interface Adder {
int add(int first, int second);
}
```

Making a Service Remote Accessible with @RemoteAccessible

Controllers can be accessible either locally or remotely. When a controller is local it can only be invoked from within your server-side application. When a controller is declared as @RemoteAccessible it is available to various network endpoints that are provisioned based on the Web containers. Absent this annotation, a controller is local.

```
@RemoteAccessible
@Contract
public interface Adder extends Controller {
int add(int first, int second);
}
```

Injecting a Service with @Inject

A class declares the services upon which it depends (usually by providing the interface contracts of those services). This enables one service to be injected with another without having to explicitly look it up. The primary benefit is that each part of the system cleanly defines its dependencies and the services that it provides.

The following example shows how one service is injected with another.

```
@Singleton
public class SimpleTupleAdder implements TupleAdder {

    @Inject
    private Adder adder;

    @Override
    public int add(int first, int second, int third) {
        return adder.add( adder.add(first, second), third);
    }
}
```

The Magnet Mobile App Server supports JSR-330 — the industry standard for injection in Java. This applies to server and Android mobile applications alike. Magnet extends the basic model to provide a richer set of options that is not available in JSR-330, for example, optional injection points.

Making the Injected @Optional

@Optional is typically used for configurable services that may be disabled or otherwise is conditionally present at runtime. If @Optional is specified and the service is disabled, or the module containing the service implementation is not present, then the reference will be null.

```
    @Optional
    @Inject
    private Adder adder;

    @Override
    public Integer add(int first, int second, int third) {
    return (adder == null) ? null : adder.add(
adder.add(first, second), third);
}
```

Implementing a Service with @Singleton

The following example shows the implementation of a singleton Adder service.

```
@Singleton
public class SimpleAdder implements Adder {
    @Override
    public int add(int first, int second) {
        return first + second;
    }
}
```

Initializing a Service with @ConfigBean

This @ConfigBean annotation is used mainly to control initialization of services, which require user-provided attributes assembled into configuration beans injected into the constructor of dependent services. The following example shows @ConfigBean used to configure for the Jetty Web container service on the server side. These types of services are often referred to as config-driven services, and while available on the mobile side, is mostly used on server side.

```
@ConfiguredBy(value = JettyConfigBean.class,
    defaultConfig = JettyService.DefaultConfigBean.class,
    validator = JettyService.DefaultConfigValidator.class)
@NamedAlias("jetty")
public class JettyService extends AbstractWebContainer
implements PostConstruct, PreDestroy, JettyStatisticsMXBean {
...
    private final JettyConfigBean config;
...
    @Inject
    public JettyService(JettyConfigBean config) {
        this.config = config;
    }
…
}
```

Making Services Writable

ServicesWritable is an extension of *Services* that supports dynamic runtime registration for service components.

```
@Contract
public interface ServicesWriteable
extends Services
```

The following table lists and describes *ServicesWritable* methods and a page reference to each method.

Methods	Description	Page
create()	Create a services by class name, and then injects all of the injection points.	page *34*
createAndInitialize()	Create the service by class, satisfies the injection points, and then handles the PostConstruct and PostStarted conditionally based on kernel state.	page *35*
destroy()	Terminate lifecycle by conditionally calling PreDestroy and Closeable.	page *35*
dump()	Empty the services registry state to the provided print writer (if possible).	page *35*
getRegisteredService()	Obtain a handle to a releasable service reference.	page *35*

Methods	Description	Page
initialize()	Set a newly created instance and manually inject all injection points, followed by calling PostConstruct and PostStarted conditionally based on kernel state.	page *36*
isReady()	Return true if this services registry instance is in a ready state.	page *36*
register()	Register a service or class as a service and return its registration reference.	page *36*
register(Object serviceOrClass, Collection<String> extraContracts, Boolean passManagement)	Register a service or class as a service and return its registration reference.	page *36*
register(Object serviceOrClass, List<String> extraAliases, Collection<String> extraContracts, boolean force, Boolean passManagement)	Register a service or class as a service and return its registration reference.	page *36*
register(Object serviceOrClass, String serviceName, List<String> serviceAliases, Set<String> contracts, Set<String> qualifiers, boolean dynamic, Integer serviceRanking, boolean force, Boolean passManagement)	Register a service or class as a service and return its registration reference.	page *37*
register(Object serviceOrClass, String serviceName, Set<String> contracts, Set<String> qualifiers, boolean dynamic, boolean force)	Register a service or class as a service and return its registration reference.	page *37*
registerDynamic()	Register a service and return its registration reference.	page *37*

create()

Create a service by a class name, and inject it to all injection points. This method also supports constructor injection (according to JSR-330 if applicable).

After creating a service, you can call ***initialize()*** on page ***36*** to initialize it.

```
create(Class<T> serviceClass)
```

createAndInitialize()

Create a service by class, satisfy the injection points, and handle the PostConstruct and PostStarted conditionally based on the kernel state.

This method does **not** register a service into the services registration. This method should not be called on a service that is being managed by the platform.

```
createAndInitialize(Class<T> serviceClass)
```

destroy()

Terminate a lifecycle by conditionally calling PreDestroy and Closeable.

```
void destroy(T service)
```

dump()

Empty the services registry state to the provided print writer (if possible).

```
void dump(PrintWriter out)
```

getRegisteredService()

Obtain a handle to a releasable service reference. Calling release will cause the backing service to go away (i.e., calling PostConstruct, Close, etc.) and eventually become garbage collected.

This method is currently supported only on dynamic services.

```
getRegisteredService(Reference<?> ref)
```

initialize()

Set a newly created instance and manually injects all injection points, followed by calling PostConstruct and PostStarted conditionally based on the kernel state.

This method does **not** register a service into the services registration however. This method should not be called on a service that is being managed by the platform.

```
initialize(T service)
```

isReady()

Return true if this services registry instance is in a ready state.

```
boolean isReady()
```

register()

Register a service or class as a service, returning its registration reference

```
register(Object serviceOrClass)
```

register()

Register a service or class as a service, returning its registration reference.

```
 register(Object serviceOrClass,
          Collection<String> extraContracts,
          Boolean passManagement)
```

register()

Register a service or class as a service, returning its registration reference.

```
register(Object serviceOrClass,
         List<String> extraAliases,
         Collection<String> extraContracts,
         boolean force,
         Boolean passManagement
```

register()

Register a service or class as a service, returning its registration reference.

```
register(Object serviceOrClass,
         String serviceName,
         List<String> serviceAliases,
         Set<String> contracts,
         Set<String> qualifiers,
         boolean dynamic,
         Integer serviceRanking,
         boolean force,
         Boolean passManagement)
```

register()

Register a service or class as a service, returning its registration reference.

```
 register(Object serviceOrClass,
          String serviceName,
          Set<String> contracts,
          Set<String> qualifiers,
          boolean dynamic,
          boolean force)
```

registerDynamic()

Register a service, returning its registration reference.

```
registerDynamic(Object service)
```

Creating New Resource

ResourceNode

```
public interface ResourceNode {
AttributeContainer getAttributeContainer();
}
```

A marker interface for an instance that represents an aggregate of attributes. Implementing this interface as a base interface or a mixin provides access to a lightweight AttributeContainer that includes a schema that can be used to bind the object to many alternative persistent stores or to marshal it to various communication protocols.

All Controller(s) that need complex data structures as parameters or return values should extend from ResourceNode and/or ResourceNodeWriteable and use EntityCreator for creation or cloning.

As a developer, you should not provide implementation classes for these interfaces; instead, let the platform generate the class implementation(s) for these (this is the role of the magnet-platform-model-apt module that should be placed in the compile classpath for annotation processing).

ResourceNodeWriteable

```
public interface ResourceNodeWriteable extends ResourceNode {
/**
* An attribute Container that can be used to mutate the entity if available. */

AttributeContainerWriteable getAttributeContainerWriteable(); }
```

ResourceNodeBuilder

Provides a means to use a fluent builder style to build a ResourceNode.

```
ResourceNodeBuilder<U extends ResourceNode,T extends
ResourceNodeBuilder<U,T>>
```

ResourceNodeCreator

ResourceNodeCreator is a service used to create new ResourceNode and ResourceNodeWriteable instances or to clone those instances.

```
@Contract
public interface ResourceNodeCreator
```

The following table lists and describes *ResourceNodeCreator* methods and a page reference to each method.

Methods	Description	Page
copyNode	Create a clone of the specified node by copying all its attributes.	page ***40***
copyNode(ResourceNode toCopy, Collection<String> attributeNames)	Duplicate the specified entity by creating a new entity with all of the same attributes.	page ***40***
createNode	Create a new instance of the ResourceNode derived class.	page ***40***
createNodeBuilder	Creates a fluent builder around a target ResourceNode type.	page ***41***
createNodeFromNode	Create a new Resource and define the MagnetUri to be the one supplied.	page ***41***
createResourceProjection	Create a new ResourceProjection by copying the values and schema definition of the specified set of properties.	page ***41***
getAttributeSchema(Class<? extends ResourceNode> nodeClass)	Use reflection to build the AttributeSchema for the specified ResourceNode derived class.	page ***42***
getAttributeSchema(Class<? extends ResourceNode> nodeClass, Collection<String> attributeSet	Create a schema that is a subset of the available properties.	page ***42***
getNodeTypeForClass	Return the default short name for an ResourceNode or Resource from it class using the class name or the @Named annotation, as appropriate.	page ***42***

copyNode

Duplicate a node that includes all its attributes.

```
copyNode(ResourceNode toCopy)
```

Parameters

toCopy indicates the node to be duplicated.

Return

None

copyNode()

Copy a node and create a new entity that includes the same attributes. The value of MagnetUri is not set so that it can be persisted anew.

```
copyNode(ResourceNode toCopy,
        Collection<String> attributeNames)
```

Parameters

- *toCopy* indicates the node to be duplicated.
- *attributeNames*

Return

None

createNode

Create a new instance of the ResourceNode derived class.

```
createNode(Class<T> clazz)
```

Parameters

clazz indicates the class.

Return

None

createNodeBuilder

Create a fluent builder around a target ResourceNode type.

```
createNodeBuilder(Class<U> clazz)
```

Parameters

clazz indicates the class.

Return

None

createNodeFromNode

Create a new Resource and define the MagnetUri to be the one supplied.

```
createNodeFromNode(Class<? extends ResourceNode> nodeClazz,
                   ResourceNode toCopy)
```

Parameters

- *nodeClazz* indicates the type of the entity to create.
- *toCopy* indicates a Resource from which to copy the existing attributes and relationships.

Return

None

createResourceProjection

Create w new ResourceProjection by copying the values and schema definition of the specified set of properties.

```
createResourceProjection(ResourceNode node,
                         Collection<String> attributes)
```

Parameters

- *node* indicates the source node.
- *attributes* indicates the set of attribute names to project from the source node.

Return

None

getAttributeSchema()

Use reflection to build the AttributeSchema for the specified ResourceNode derived class.

```
getAttributeSchema(Class<? extends ResourceNode> nodeClass)
```

getAttributeSchema()

Create a schema that is a subset of the available properties.

```
getAttributeSchema(Class<? extends ResourceNode> nodeClass,
                   Collection<String> attributeSet)
```

getNodeTypeForClass

Get the default short name for an ResourceNode or Resource from it class using the class name or the @Named annotation as appropriate.

```
getNodeTypeForClass(Class<? extends ResourceNode> nodeClass)
```

Parameters

nodeClass indicates the name of the class.

Return

The class name.

Interface Reference<T>

A simple reference, *extending javax.inject.Provider* to support class type accessibility.

```
public interface Reference<T>
extends Provider<T>
```

Provider and *Reference* are used extensively in the kernel of the platform. It has a number of uses:

- A part of service lookup. When a service component is looked up out of the services registry the returned service(s) are not yet constructed or activated until the first time the service(s) are referenced. This is called lazy initialization. Once a service is activated, it typically stays active for Singleton type services.
- *References* serve as a proxy to underlying *Dynamic* services, where those service(s) might "come and go" throughout the lifetime of the application. For example, Android's Activity instances are activated by the Google Android kernel — they come and go basically at any time. When these singletons are activated, however, one would like to maintain a proxy reference to those activity instances.
- Factory providers. Sometimes *References* and *Providers* act as factories for callers.

Reference<T> includes one method, getType.

getType

The class type of instances that will be provided by the Provider.get() method. Calling this method will not activate any lazy service. Return null if the class type is not known or may be different from call to call.

```
Class<?> getType()
```

Data Types

Magnet exposes RESTful interfaces to remote clients, but can use SOAP to talk to external Web services. Data types are used as a standardized way to define, send, receive, and interpret basic data types in the SOAP messages exchanged between client applications and the external Web services.

Magnet supports all Java primitive types: byte, short, int, long, float, double, char, boolean. The following data types are supported in REST messages in addition to Java primitives:

- java.lang.String
- byte[]
- enum
- java.net.URI
- java.util.Date
- java.math.BigDecimal
- java.math.BigInteger
- com.magnet.api.ResourceUri
- com.magnet.attributes.api.Data (and its various subclasses)
- com.magnet.model.api.Node (including all its subclasses)
- java.util.List (including its various subclasses)
- com.magnet.common.api.Reference (including its various subclasses)

Java-XML Binding

WSDL-based services encode data as XML types. The following table lists the mapping from WSDL-services into Magnet Java types.

XML Type	Magnet Java Type
xsd:base64binary	com.magnet.attributes.api.Data
xsd:anySimpleType (for xsd:element of this type)	com.magnet.model.api.Node
xsd:anySimpleType (for xsd:attribute of this type)	N/A
xsd:boolean	boolean
xsd:byte	byte
xsd:date	java.util.Date
xsd:dateTime	java.util.Date
xsd:double	double
xsd:duration **Note**: Not supported in the current release.	N/A
xsd:float	float
xsd:g	java.util.Date
xsd:hexBinary	com.magnet.attributes.api.Data
xsd:int	int
xsd:integer	java.math.BigInteger
xsd:long	long
xsd:NOTATION	com.magnet.connect.controller.client.api.QNameNode
xsd:QNameNode	com.magnet.connect.controller.client.api.QNameNode
xsd:short	short
xsd:string	java.lang.String
xsd:time	java.util.Date
xsd:unsignedByte	short
xsd:unsignedInt	long
xsd:unsignedShort	int
xsd:unsignedLong	java.math.BigDecimal
Data Structures	
xsd:list	java.util.List
Enum Types	
xs:enumeration	enum

XML Type	Magnet Java Type
Nillable	
xsd:element with attribute xsd:nillable=”true”	com.magnet.common.api.Reference<T>
WSDL Types	SOAPFault is not supported. wsdl:fault is mapped to Exception types that are thrown by the method they are associated with.

6. Working with Persistence

Persistence is about storage and retrieval of structured data. Magnet Mobile Enterprise Server (MES) provides support for entities and mapping entities to the data store. This creates, in effect, a “virtual entity data store” over any combination of MySQL, LDAP, and so forth, that can be used from within the programming interface Magnet provides.

Magnet Mobile Enterprise Server uses “interfaces” to define functions, methods, classes, and requests. MES generates classes build on annotations on interfaces, which allows for relationship, attributes. Using user-friendly interfaces on entities, MES provides

- Queries on entity with query composition
- CRUD (Create, Read, Update, Delete) operations on entities
- Complex queries such as paging and ordering

You can choose any technology (JDBC, Hibernate, etc.) to access a data store. However, implementing with Magnet persistence APIs is easier to use, and powerful over time. Every time when an entity is used, unstructured data is collected as a result of using that API. Over time, recommendation can be optionally derived from that unstructured data. For example, when something is changed on an employee record, the implementation would track when that change was made and by whom.

The Magnet persistence APIs simplify Java persistence and provide an object-relational mapping approach that allows you to declaratively define how to map Java objects to database tables.

Using Annotations

Magnet Mobile Enterprise Server uses annotations to support persistence. An annotation is a simple, expressive means of decorating Java source code with metadata that is compiled into the corresponding Java class files for interpretation at runtime.

The following annotations are used at an interface level:

- @Entity (page ***49***)
- @ValueType (page ***50***)

The following annotations are used at the method level:

- @Property (page ***52***)
- @Id (page ***54***)
- @Unique (page ***55***)
- @Relationship (page ***56***)

MES persistence supports strong typing interface for querying and manipulations of entities. Unlike JPA, which lacks strong typing and string queries, MES persistence enables developer to discover errors at compilation time.

Defining @Entity

In order to persist and manipulate entities, MES processes the annotated interfaces to generate the classes used for persisting, manipulating, and querying these entities. The generated class for each entity includes the support for equals() and hash code() based on the identity properties of the entity.

The set of Primitive types include String, Integer (int), Short (short), Long (long), Float (float), Double (double), BigDecimal, Boolean, and Date (java.util.Date).

@Entity is an interface that correlates to a row in a table by means of a unique identifier. Because of this unique identifier, entities exist independently. You can reference an individual entity by its ID.

- @Entity is used to indicate its persistence.
- Entities must be identifiable; you can create and retrieve an entity using its ID.
- You can have composite ID (having multiple IDs) within an entity; however, an entity must contain at least one field annotated by @Id.

@Entity has attributes as described in the following table.

Attribute	Description
name	This optional attribute specifies the name for the entity. By default, the name is the simple class name (camel-cased, non-packaged name).
collection	This optional attribute specifies the name for the collection of which the entity instances are members. By default, the collection name is the plural form of the entity name.
baseType	This optional class is the base implementation type of the entity. Specify the base when you want to augment the generated class with your own methods.

Example

The following example shows an entity **FulfillmentOrder** is created with a default collection.

```
@Entity(name="FulfillmentOrder", collection="FulfillmentOrders",
baseType=OrderBase.class)
public interface Order {
...
}

public class OrderBase implements Order {
...
}
```

Defining @ValueType

Value type is used to identify a collection of properties, which may include small groups of related variables. Value objects are changeable. Changing one property of a value object essentially destroys the old object and creates a new one.

Value type can have inheritance (extend) but can not be abstract, which means a value type cannot be instantiated and does not have any instances of its own.

Value types only exist in terms of their parent entities and have no separate identity. Moreover, value types are always owned by their containing entity. Once you delete the entity, you also delete all attributes associated to that entity.

- @Entity and @ValueType are mutually exclusive.
- @ValueType cannot have any method annotated with @Id when it has a base type specified via @CodeGen annotation.

Refer to ***Creating an Entity Using ObjectBuilder*** on page ***63*** for object construction using proxies.

@ValueType has one optional attribute as described in the following table.

Attribute	Description
baseType	This optional class is the base implementation type of the entity.

Example

```
@ValueType(baseType=AddressBase.class)
public interface Address {

    @Property(maxLength = 50)
    String getAddress1();

    void setAddress1(String value);

    @Property(nullable = true)
    String getAddress2();

    void setAddress2(String value);

    @Property
    String getCity();

    void setCity(String value);

    @Property
    String getState();

    void setState(String value);

    @Property
    int getZip();

    @Property(nullable = true)
    Integer getZip4();
}

public class AddressBase implements Address {

...
}
```

Defining @Property

Persistent properties are indicated using @Property. You can specify @Property on a mapped attribute within an entity. Return type of property may be:

- Java primitive types
- user-defined valued types
- collection of primitive types or user-defined valued types

All methods annotated with @Property must start with "get" or "set" with the exception for those returning the boolean type, which should start with "is."

Release 2.0 supports the basic collection (bag). It supports Collection and Array but does not support set and list.

@Property has optional attributes as described in the following table.

Attribute	Type	Description
generated	boolean	■ true indicates that the values of the field are generated. ■ **false** (default) indicates otherwise.
nullable	boolean	■ true indicates that the values of the field are null. ■ **false** (default) indicates otherwise.
readOnly	boolean	■ true indicates a read-only field. ■ **false** (default) indicates otherwise.
maxLength	Int	This attribute specifies the maximum length of the field. The default is **-1**. **Note**: This attribute is applicable only to the String field.
maxValue	String	This attribute specifies the maximum value of the field. The default is an empty string. **Note**: This attribute is applicable only to number fields.
minValue	String	This attribute specifies the minimum value of the field. The default is an empty string. **Note**: This attribute is applicable only to number fields.

Attribute	Type	Description
defaultValue	String	This attribute specifies the default value of the field. The default is an empty string.
possibleValues	Array	This attribute specifies the possible values of the field. The default is **0**.

Example

```
@Entity
public interface Employee {

    @Id

    @Property(generated=true, readOnly=true)
    int getEmployeeId();

    @Property(maxLength=20, nullable = false)
    String getLastName();

    @Property(maxLength=10, nullable = false)
    String getFirstName();

    @Property(possibleValues={"Mr.", "Mrs.", "Ms."},
defaultValue="Mr.",nullable = true)
    String getTitle();

    @Property(nullable = true)
    Date getBirthDate();

    @Property(maxValue = 1000)
    int getSignOnBonus();

    ...
}
```

Defining @Id

The @Id annotation is used to identify one of more properties to an entity. The ID is mapped to the primary key.

All methods annotated with @Id must start with "get" or "set" with the exception for those returning the boolean type, which starts with "is."

In the current release, @Id properties cannot be nullable and cannot be of a ValueType.

@Id has an optional attribute as described in the following table.

Attribute	Type	Description
generated	boolean	■ true indicates that the ID is generated. ■ **false** (default) indicates otherwise.

Example

```
@Entity
public interface Student {

    @Id(generated=true)
    String getStudentId();

    ...
}
```

Defining @Unique

The @Unique annotation is translated to a unique constraint. @Unique can be nullable. You can have composite Unique (having multiple fields); in this case, name the Unique for combining multiple fields.

All methods annotated with @Unique must start with "get" or "set" with the exception for those returning the boolean type, which starts with "is."

Do not use @Unique to annotate a subset of properties annotated with @Id.

@Unique has an attribute as described in the following table.

Attribute	Type	Description
value	String	This attribute specifies the name of the unique constraint.

Example

The following example shows Department (Entity interface) is identified by properties companyId and name, but it has an unique constraint defined with two properties (companyId, departmentName).

```
@Entity
public interface Department {

    @Property(generated=true,maxLength=20)

    @Id

    String getName();

    @Unique(name="uniqueDept")
    @Id
    String getCompanyId();

    @Unique(name="uniqueDept")
    String getDepartmentName();
}
```

Defining @Relationship

The @Relationship annotation indicates the relationship of the property to the entity. The differences between relationship types include how they handle data deletion and record ownership. You can include multiple relationships within an entity. When using this technique, you must name the relationship, and the name must be unique.

In relationship, there are owner and participants. The notion of ownership is similar to parent-to child relationship. When a parent is removed, all children are removed. Similarly, when the owner is removed, so are the participants.

The relationship can be one-to-one, one-to-many, and many-to-many. You can declare relationship one of three ways: via properties, via a single property, or via a package.

All methods annotated with @Relationship must start with "get" or "set." Boolean return type is not allowed in relationship methods. The return types of the methods annotated with @Relationship must be non-primitive, declared type.

@Relationship has optional attributes as described in the following table.

Attribute	Type	Description
name	String	This attribute specifies the name of the relationship. By default, the relationship name is the concatenation of the names of participants, listed in alphabetical order, and each name is separated by a hyphen "–."
owner	boolean	■ true indicates that the current entity is the owner of the relationship. ■ **false** (default) indicates otherwise.
embedded	boolean	■ true indicates that the relationship is embedded in the current entity. ■ **false** (default) indicates otherwise. **Note**: When this attribute is set to true, deleting the entity also triggers the deletion of the relationship.
Participant	String	This attribute indicates the name of the participant corresponding to the current entity. By default, the participant name is the entity name.

One-to-One Relationship Example

This is a parent-child relationship in which the parent controls certain behaviors of the child.

- When a parent record is deleted, its related child records are also deleted.
- A child cannot be the record owner, and its record is automatically set to the owner of its associated parent record.
- The child record inherits the sharing and settings of its parent record.

The example shows a one-to-one relationship between Product and Supplier that are defined by *Product.getMyOnlySupplier()* and *Supplier.getMyOnlyProduct()*. The relationship is embedded (*true*) in the entity Product (*Product.getMyOnlySupplier()*), and the owner of the relationship is Supplier (*Supplier.getMyOnlyProduct()*). The deletion of a supplier triggers the deletion of the related product.

```
@Entity
public interface Product {

@Id
int getProductId();
...

@Relationship(embedded=true)
Supplier getMyonlySupplier();
void setMyOnlypplier(Supplier supplier);

}

@Entity
public interface Supplier {
@Id
int getSupplierId();
...

@Relationship(owner = true)
Product getMyOnlyProduct();
...
}
```

One-to-Many Relationship Example

A one-to-many relationship allows one record to be linked to multiple records from another object. The following example shows a one-to-many relationship between Category and Product defined by *Category.getProducts()* and *Product.getCategory()*. “RelativeCollection” is used to map to associated entities.

```
@Entity
public interface Product {

@Id
int getProductId();
...

@Relationship
RelativeCollection<Product, Category> getCategory();
}

@Entity
public interface Category {

@Id
int getCategoryId();
...

@Relationship
RelativeCollection<Category, Product> getProducts();
}
```

Many-to-Many Relationship Example

A many-to-many relationship allows each record of one object to be linked to multiple records from another object and vice versa. The following example shows a many-to-many relationship between Product and Order by Product.getOrders() and Order.getProducts(). "RelativeCollection" is used to map to associated entities.

```
@Entity
public interface Product {

@Id
int getProductId();
...

@Relationship(owner=true)
RelativeCollection<Product, Order> getOrders();}

@Entity
public interface Order {

@Id
int getOrderId();
...

@Relationship
RelativeCollection<Order, Product> getProducts();
...
}
```

Defining Context

At runtime, MES supports context that is used to perform data manipulation operations on the user-defined entities and access to different collections of those entities. EntityContextDefinition is the entry point for performing persistent operations. It is injected into a module as shown in the following example:

```
@Inject
private EntityContextDefinition definition;
```

Example -- MES Controller Using Persistence

```
...
@Singleton
public class ProductControllerImpl implements ProductController {

    private Random random = new Random();
    @Inject
    private EntityContextDefinition contextDefinition;
    private Product buildProduct(fulfillment.Product p) {
        ObjectBuilder<Product> builder = ObjectBuilder.get(Product.class);
        Product product = builder.getProxy();
        product = builder.p(product.getName()).set(p.getProductName())
            .p(product.getId()).set(p.getProductId()).create();
    return product;

    }
...
```

Example -- Retrieving a Product Using Its Identity

```
...
    private fulfillment.Product getProductCore(EntityContext context, String
productName) {
    EntityCollection<fulfillment.Product> coll =
context.getCollection(fulfillment.Product.class);
    QueryBuilder b = new QueryBuilder();
    fulfillment.Product proxy = b.source(coll);
    Queryable<fulfillment.Product> q =
b.from(proxy).where().p(proxy.getProductName()).eq(productName).select(proxy).
build();
    for (fulfillment.Product p : q) {
        return p;
    }
    return null;
}
...
```

Example -- Implementation of the Controller Method to Get a Product Using Its Identity

```
...
    @Override
    public Product getProduct(String productName) {
    EntityContext context = contextDefinition.createContext();
    fulfillment.Product p = getProductCore(context, productName);
    if (p != null) {
        return buildProduct(p);
    }
    return null;
}
...
```

Example -- Deleting a Product Using Its Identity (Product Name)

```
...
    @Override
    public void deleteProduct(String productName) {
        EntityContext context = contextDefinition.createContext();
        fulfillment.Product p = getProductCore(context, productName);
        if (p != null) {
            EntityCollection<fulfillment.Product> coll =
context.getCollection(fulfillment.Product.class);
            coll.remove(p);
    }
}
...
```

Example -- Creating a Product Entity and Using Money of a Value Type

```
...
    @Override
    public int createProduct(String productName) {
        EntityContext context = contextDefinition.createContext();
        EntityCollection<fulfillment.Product> coll =
context.getCollection(fulfillment.Product.class);
        ObjectBuilder<Money> m = ObjectBuilder.get(Money.class);
        Money n = m.getProxy();
        n = m.p(n.getCurrency()).set((short) 10)
            .p(n.getAmount()).set(BigDecimal.valueOf(10)).create();
        ObjectBuilder<fulfillment.Product> b =
ObjectBuilder.get(fulfillment.Product.class);
        fulfillment.Product p = b.getProxy();
        ObjectBuilder<fulfillment.Category> ca =
ObjectBuilder.get(fulfillment.Category.class);
        Category cat = ca.getProxy();
        cat =
ca.p(cat.getCategoryId()).set(2).p(cat.getCategoryName()).set("categoryName")
            .p(cat.getDescription()).set("Category description").create();
        ObjectBuilder<Supplier> su = ObjectBuilder.get(Supplier.class);
        Supplier sup = su.getProxy();
        sup =
su.p(sup.getSupplierId()).set(10).p(sup.getContactName()).set("contactName").c
reate();
    p = b.p(p.getProductId()).set(random.nextInt())
    .p(p.getReorderLevel()).set((short) 10)
    .p(p.getUnitOnOrder()).set((short) 30)
    .p(p.getUnitsInStock()).set((short) 40)
    .p(p.getQuantityPerUnit()).set("2Ltrs")
    .p(p.getCategory()).set("sample")
    .p(p.isDiscontinued()).set(false)
    .p(p.getProductName()).set(productName)
    .p(p.getUnitPrice()).set(n).create();
    coll.add(p);
    return p.getProductId();
}
```

Example -- Using Query to Get the List of all Products

```
...
    @Override
    public List<Product> getProducts() {
        EntityContext context = contextDefinition.createContext();
        EntityCollection<fulfillment.Product> coll =
context.getCollection(fulfillment.Product.class);
        List<Product> products = new ArrayList<Product>();
        for (fulfillment.Product p : coll) {
        products.add(buildProduct(p));
        }
        return products;
    }
    @Override
    public String getVersion() {
        return PlatformProvider.MAGNET_VERSION_1_0_0;
    }
}
```

Creating an Entity Using ObjectBuilder

1 Get an instance of the target entity:

```
ObjectBuilder<Product> builder =
ObjectBuilder.get(Product.class);
```

2 Set the proxy:

```
Product product = builder.getProxy();
```

3 Set the value of the entity properties.

```
builder.p(product.getName()).set(p.getProductName())

.p(product.getId()).set(p.getProductId());
```

4 Create an instance by calling create on the builder.

```
builder.create();
```

Declaring an Entity Collection Interface

The **EntityCollection**<T> interface includes methods for managing a collection.

```
public interface EntityCollection<T> extends Queryable<T>
```

The following table lists and describes the supported methods and a page reference to each of the method.

Methods	Description	Page
add	Add an entity to the collection.	page *64*
attachAsUpdated	Update an entity.	page *65*
getById	Return an entity with a specific Id properties.	page *65*
getContext	Return an entity context associated with the current collection.	page *65*
getDefinition	Return an entity definition associated with the current collection.	page *66*
remove	Remove an entity from the collection.	page *66*
removeById	Remove an entity from the collection.	page *66*

add

Add an entity to the collection, resulting inserting it into the underlying data store.

```
boolean add(T entity);
```

Parameter

- *entity* indicates the entity you want to add to the collection.

Return

boolean.

- Yes indicates that the entity is added.
- No indicates otherwise.

attachAsUpdated

Update an entity.

```
boolean attachAsUpdated(T entity);
```

Parameter

- *entity* indicates the entity you want to update.

Return

Not applicable.

getById

Return an entity with a specific Id properties.

```
T getById(Object... idParts);
```

Parameter

- *idParts* indicates the value of the Id properties associated with the entity you want to return.

Return

The entity associated with the Id properties you specified.

getContext

Return an entity context associated with the current collection.

```
EntityContext getContext();
```

Parameter

- None.

Return

getDefinition

Return an entity definition associated with the current collection.

```
EntityDefinition<T> getDefinition();
```

Parameter

None

Return

The EntityDefinition that includes the metadata (properties, relationships, and base entity) associated with this entity instance.

remove

Remove the whole entity from the collection. The entity is deleted from the persistence data store (MySql) and any of its dependent artifacts, which include collection, relationships, and owned entities.

```
boolean remove(Object entity);
```

Parameter

- *entity* indicates the entity you want to remove.

Return

- Yes indicates that the entity is removed.
- No indicates otherwise.

removeById

Remove the entity from the collection. The entity is deleted from the persistence data store (MySql) and all its dependent artifacts, which include collection, relationships, and owned entities.

```
boolean removeById(Object... idParts);
```

Parameter

- *idParts* indicates the value of the Id properties associated with the entity you want to remove.

Return

- Yes indicates that the entity is removed.
- No indicates otherwise.

Deleting Entity

Each entity defined belongs to a collection. To delete an entity, simply remove it from its collection, as shown in the following example.

```
// Fetch the entity to delete
    fulfillment.Product p = getProductCore(context, productName);
    if (p != null) {

// Get the collection from the context
    EntityCollection<fulfillment.Product> coll =
context.getCollection(fulfillment.Product.class);

// Remove the product from the collection which results in deleting it from
// the persistence store.
    coll.remove(p);
    }
```

Defining Interface Methods

EntityCollection

```
public interface EntityCollection<T>
extends com.magnet.persistence.queryable.Queryable<T>
```

The following table lists and describes EntityCollection methods and a page reference to each method.

Methods	Description	Page
add()	Add an entity to the collection.	page *68*
attachAsUpdated()	Add an entity to the current context and update it.	page *68*
getById()	Return an entity with the specified ID properties.	page *68*
getContext()	Return an entity context associated with the current collection.	page *69*
getDefinition()	Return an entity definition associated with the current collection.	page *69*
remove()	Remove the whole entity from the collection.	page *69*
removeById()	Remove an entity associated with the ID specified.	page *70*

add()

Add an entity to the collection, resulting the entity being inserted into the underlying data store.

```
boolean add(T entity)
```

Parameters

entity indicates the type of entity to be added to the collection.

Return

- true indicates that entity is added.
- false indicates otherwise.

attachAsUpdated()

Add an entity to the current context and update it.

```
boolean attachAsUpdated(T entity)
```

Parameters

entity indicates the type of entity to be updated.

Return

- true indicates that the entity is added.
- false indicates otherwise.

getById()

Return an entity with the specified ID properties.

```
T getById(Object... idParts)
```

Parameters

idParts indicates the value of the ID properties to be returned.

Return

The entity with the specific ID properties.

getContext()

Return an entity context associated with the current collection.

```
EntityContext getContext()
```

Parameters

None

Return

An entity context.

getDefinition()

Return an entity definition associated with the current collection. The entity definition is the object associated with the entity.

```
com.magnet.persistence.metadata.EntityDefinition<T>
getDefinition()
```

Parameters

None

Return

An entity definition associated with the current collection.

remove()

Remove the whole entity from the collection. This method deletes the entity from the underlying data store as well as all of its dependent artifacts, including collection, relationships, and owned entities.

To remove a specific entity, call the removeById() on page *70*.

```
boolean remove(Object entity)
```

Parameters

Entity indicates the entity to be removed.

Return

- true indicates that the entity is removed.
- false indicates otherwise.

removeById()

Remove an entity associated with the ID specified. This method deletes the entity from the underlying data store as well as all of its dependent artifacts, including collection, relationships, and owned entities.

```
boolean removeById(Object... idParts)
```

Parameters

idParts indicates the value of the ID properties to be removed.

Return

- true indicates that the entity is removed.
- false indicates otherwise.

EntityContext

EntityContext is an interface used for manipulating persistence entities, which includes methods for fetching an entity using it ID properties.

```
public interface EntityContext
```

The following table lists and describes EntityContext methods and a page reference to each method.

Methods	Description	Page
getById	Return an entity with the specified identity.	page *71*
getCollection	Return collection.	page *71*
getCollection	Return collection.	page *71*
getCollection	Return collection.	page *72*
getDefinition()	Return an entity definition associated with the current collection.	page *72*

getById

Return an entity with the specified identity.

```
<T> T getById(Identity<T> identity)
```

Parameters

Identity indicates the entity to be returned.

Return

The entity, or null if it does not exist.

getCollection

```
<T> EntityCollection<T> getCollection(Class<T> entityType,
                                      String name)
```

Parameters

Return

The runtime representation of a collection as well as for manipulating it.

getCollection

```
<T> EntityCollection<T> getCollection(Class<T> entityType)
```

Parameters

Return

The runtime representation of a collection as well as for manipulating it.

getCollection

```
<T> EntityCollection<T>
getCollection(com.magnet.persistence.metadata.EntityDefinition<T> definition)
```

Parameters

Return

Collection given the metadata of the entity as defined by the EntityDefinition.

getDefinition()

Return an entity context definition associated with the current collection.

```
EntityContextDefinition getDefinition()
```

Parameters

None

Return

An entity definition.

RelativeCollection

RelativeCollection is an interface that provides many sides of a relationship, including a single side of one-to-many and both sides of many-to-many relationships.

```
public interface RelativeCollection<TOwner,TRelated>
extends com.magnet.persistence.queryable.Queryable<TRelated>
```

The following table lists and describes RelativeCollection methods and a page reference to each method.

Methods	Description	Page
add	Add an entity.	page *73*
getOwner()	Return the owner of the relationship.	page *73*
getOwnerId()	Return the identity of the owner.	page *73*
getOwnerParticipant()	Return owner participant.	page *74*

Methods	Description	Page
getTargetParticipants()	Return target participant.	page *74*
remove()	Remove an entity from its related collection.	page *74*

add

Augment the set of related entity with an additional entity.

```
boolean add(TRelated relatedEntity)
```

Parameters

relatedEntity is the target entity of this relationship.

Return

- true indicates that entity is added.
- false indicates otherwise.

getOwner()

Return the owner of the relationship.

```
TOwner getOwner()
```

Parameters

None

Return

The type of owner.

getOwnerId()

Return the identity of the owner.

```
Identity<TOwner> getOwnerId()
```

Parameters

None

Return

The identity of the owner.

getOwnerParticipant()

```
com.magnet.persistence.metadata.RelationshipParticipantDefinition
getOwnerParticipant()
```

Parameters

None

Return

getTargetParticipants()

```
com.magnet.persistence.metadata.RelationshipParticipantDefinitionSet
getTargetParticipants()
```

Parameters

None

Return

remove()

Remove an entity from its related collection.

```
boolean remove(Object relatedEntity)
```

Parameters

relatedEntity is the target entity of this relationship.

Return

- true indicates that entity is removed.
- false indicates otherwise.

Creating Custom Controllers

This section describes how you add business logic components to your application. The following example shows codes on defining a base controller.

```
@Contract
@RemoteAccessible
public interface ProductController extends Controller {

Product getProduct(@Named("productName") String productName);

void deleteProduct(@Named("productName") String productName);

int createProduct(@Named("productName") String productName);

List<Product> getProducts();

}
```

Compiling Codes

Magnet provides several tools that are used at code compilation.

- Annotation Processing Tool (APT) is used to process the annotations to produce metadata. The output includes an XML file representing the metadata.
- Magnet persistence plugin is used to aggregate the metadata when there are multiple modules as dependencies. The single output XML file includes the aggregate of multiple metadata XML files from the dependencies.
- DDL generator is used to generate DDL and to validate an existing database schema against the DDL.

For general Java compiler errors, refer to Oracle Web page for description.

Setting Up the Annotation Processing Tool

The Java Annotation Processing Tool allows external products to integrate easily with the Java compiler to augment the capabilities of the Java compiler by interpreting meta data that you incorporate into their source code.

Simply adding the APT module to your project during compilation causes the required artifacts to be generated into your target classes directory. The compiler detects the presence of the annotation processor and causes it to be invoked when the Magnet annotations are applied. An output XML file for logical metadata is automatically placed under the *target/classes/META-INF/persistence* directory. The same XML file is also packaged in the JAR file.

1. Open your project POM.XML file.
2. Add the dependency to the POM file as shown in the following example.

```
<dependency>
    <groupId>com.magnet.persistence</groupId>
    <artifactId>magnet-platform-persistence-model-metadata-apt</artifactId>
    <version>2.0</version>
</dependency>
```

Adding Magnet Persistence Plugin

Magnet persistence plugin is used to aggregate logical metadata XML files found in the JARs of the dependencies. For each dependency, the plugin looks for XML files under META-INF/persistence in the JAR. It makes temporary copies of the XML files, parses the XML files, checks for all referenced classes, and finally produces a new XML file, named *persistence-metadata.xml*, under the META-INF/persistence directory.

It is typical that plugin is used in combination with the APT module and/or modules that use APT module. That following example shows the addition of plugin to the POM file.

```
<build>
    <plugins>
        <plugin>
            <groupId>com.magnet.persistence</groupId>
            <artifactId>magnet-persistence-maven-plugin</artifactId>
            version>${project.version}</version>
            <executions>
                <execution>
                  <phase>
                     process-classes
                  </phase>
                  <goals>
                     <goal>
                       metadata
                     </goal>
                  </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

Running DDL Generator

DDL generator is a tool to generate DDL for a specific database from the metadata.

It is typical that plugin is used in combination with the APT module and/or modules that use APT module. The following is an example of running DDL generator by calling its main method, using Maven exec plugin.

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>exec-maven-plugin</artifactId>
  <version>1.1.1</version>
   <executions>
    <execution>
      <phase>test</phase>
      <goals>
      <goal>java</goal>
      </goals>
      <configuration>
      <mainClass>com.magnet.persistence.DDLGenerator</mainClass>
      <arguments>
        <argument>-cp</argument>
        <argument>${project.basedir}/target/persistence-tests-metadata-0.9.0-
SNAPSHOT.jar</argument>
        <argument>-metadata</argument>
        <argument>${project.basedir}/target/classes/META-INF/persistence/
persistence-tests-metadata.xml</argument>
        <argument>-output</argument>
        <argument>target/mysql/ddl.sql</argument>
        <argument>-database</argument>
        <argument>MySql</argument>
        <argument>-schema</argument>
        <argument>Magnet</argument>
      </arguments>
      </configuration>
    </execution>
   </executions>
</plugin>
```

Building your Database

After the entity is processed, a Data Definition Language (DDL) file is generated that includes a table to store the defined EntityDefinition.

EntityWithPrimitiveTypes

```
@Entity
public interface EntityWithPrimitiveTypes {

    @Id
    int getId();

    @Property
        boolean getBoolean();
    @Property
        char getCharacter();
    @Property
        byte getByte();
    @Property
        short getShort();
    @Property
        int getInteger();
    @Property
        long getLong();
    @Property
        float getFloat();
    @Property
        double getDouble();
    @Property
        BigDecimal getBigDecimal();

    @Property(nullable = true)
        Boolean getBooleanNullable();
    @Property(nullable = true)
        Character getCharacterNullable();
    @Property(nullable = true)
        Byte getByteNullable();
    @Property(nullable = true)
        Short getShortNullable();
    @Property(nullable = true)
        Integer getIntegerNullable();
    @Property(nullable = true)
        Long getLongNullable();
    @Property(nullable = true)
        Float getFloatNullable();
    @Property(nullable = true)
        Double getDoubleNullable();

    @Property
        Date getDate();
    @Property
        String getString();
}
```

The default mapping from the Java primitive types into MySQL are described in the following example:

Java Primitive Type	MySQL
@Id	
id	id INT NOT NULL
@Property	
BooleanNotNull	booleanNotNull BOOLEAN NOT NULL
CharacterNotNull	characterNotNull CHAR NOT NULL
ByteNotNull	byteNotNull TINYINT NOT NULL
ShortNotNull	shortNotNull SMALLINT NOT NULL
int getIntegerNotNull	integerNotNull INT NOT NULL
LongNotNull	longNotNull BIGINT NOT NULL
FloatNotNull	floatNotNull FLOAT NOT NULL
DoubleNotNull	doubleNotNull DOUBLE NOT NULL
BigDecimalNotNull	bigDecimalNotNull DECIMAL NOT NULL
@Property(nullable = true)	
BooleanNullable	booleanNullable BOOLEAN
CharacterNullable	characterNullable CHAR
ByteNullable	byteNullable TINYINT
ShortNullable	shortNullable SMALLINT
IntegerNullable	integerNullable INT
LongNullable	longNullable BIGINT
FloatNullable	floatNullable FLOAT
DoubleNullable	doubleNullable DOUBLE
@Property	
DateNotNull	dateNotNull BIGINT NOT NULL
StringNotNull	stringNotNull VARCHAR (255) NOT NULL

SQL File

The SQL file includes the following for each table:

```
DROP TABLE IF EXISTS `EntityWithPrimitiveTypes`;

CREATE TABLE `EntityWithPrimitiveTypes` ( id INT NOT NULL,
shortNullable SMALLINT , shortNotNull SMALLINT NOT NULL, dateNotNull
BIGINT NOT NULL, floatNullable FLOAT , stringNotNull VARCHAR (255) NOT
NULL, longNotNull BIGINT NOT NULL, bigDecimalNotNull DECIMAL NOT NULL,
booleanNullable BOOLEAN , doubleNotNull DOUBLE NOT NULL,
characterNotNull VARCHAR (255) NOT NULL, byteNotNull TINYINT NOT NULL,
booleanNotNull BOOLEAN NOT NULL, integerNotNull INT NOT NULL,
floatNotNull FLOAT NOT NULL, byteNullable TINYINT , doubleNullable
DOUBLE , longNullable BIGINT , characterNullable VARCHAR (255) ,
integerNullable INT );

ALTER TABLE `EntityWithPrimitiveTypes` ADD PRIMARY KEY ( id);
```

Defining Query

Magnet Mobile Enterprise Server provides a user-friendly query builder. The query builder enables you to build the target query using proxy. The QueryBuilder is instantiated using a default constructor without any arguments.

As shown in the following example, the first step to compose a query is to get the EntityCollection associated with the target EntityDefinition, using the getCollection method on the EntityContextDefinition (context). The API for building a query provides a strongly typed for defining the main three clauses.

- The **from** clause uses proxy for the data source; this is obtained from the source method on the QueryBuilder instance.
- The **where** method defines the predicate on the source properties used.
- The **select** method defines the projection list.

```
EntityCollection<fulfillment.Product> coll =
context.getCollection(fulfillment.Product.class);

QueryBuilder b = new QueryBuilder();
fulfillment.Product proxy = b.source(coll);
Queryable<fulfillment.Product> q =
b.from(proxy).where().p(proxy.getProductName()).eq(productName)
.select(proxy).build();
for (fulfillment.Product p : q) {
return p;
}
```

Queryable Interface

Queryable is a parameterized interface that captures the composed query.

```
public interface Queryable<T> extends Iterable<T> {

Type<T> getType();

Expression getExpression();

QueryProvider getProvider();

}
```

Configuring your Service

You can specify values in configuration properties (cproperties) files to set the default entity context and default data provider. These cproperties files are stored in the *config.dir* directory.

DefaultEntityContext-1.cproperties

You specify the default entity context values in the *DefaultEntityContext-1.cproperties* file. The following example shows the default configuration settings. The following table lists and describes valid values that can be specified in the *DefaultEntityContext-1.cproperties* file.

```
metadataFileLocation=META-INF/persistence/magnet-platform-
persistence-demos-fulfillment.xml
validatePhysicalStore=no_validation
```

Parameter	Value
metadataFileLocation	The relative path to the metadata XML file. The default value is **META-INF/persistence/ persistence-metadata.xml**.
validatePhysicalStore	The DDL validation level of the physical store upon server startup. Possible values are: ■ no_validation ■ table_only ■ column_only ■ column_type

DefaultDataProvider-1.cproperties

You specify the default data provider values in the *DefaultDataProvider-1.cproperties* file. The current release supports MySql as an only data provider; therefore, parameters in this configuration are related to the MySql JDBC connection, connection pool, and transaction management.

The following example shows the default configuration settings. The following table lists and describes valid values that can be specified in the *DefaultDataProvider-1.cproperties* file.

```
dmlUserName=
dmlUserPassword=
driver=com.mysql.jdbc.Driver
url=
enabled=
```

Parameter	Value
dmlUserName	The user name for DML operation.
dmlUserPassword	The password for the above user.
driver	The JDBC driver. The only driver currently supported is **com.mysql.jdbc.Driver**.
url	The URL for the MySql database. Typically, it is **jdbc:mysql://localhost:3306/ testDatabase?createDatabaseIfNotExist=true**
enabled	■ true to enable the configuration. ■ false to disable the configuration.

Magnet

7. Working with Transaction API

Magnet Mobile App Server provides the ability to handle transactions to maintain the integrity of data. A single transaction consists of one or more independent units of work, each reading and/or writing information to a data store. When this happens, all processing leaves the data store in a consistent state.

The Magnet transactions are JTA-compliant. They provide an “all-or-nothing” proposition, stating that each work-unit performed in a data source must either complete in its entirety (committed) or have no effect whatsoever (rollback).

Annotations

An annotation is a simple, expressive means of decorating Java source code with metadata. Using annotation, you can specify how transactions are managed with the following annotation on a Controller interface or any of its methods to override the default transaction attribute.

```
com.magnet.transaction.annotation.Transactional
```

- If no Transactional.Attribute enum value is specified with the Transactional annotation, the default Transactional.Attribute enum value is Transactional.Attribute.REQUIRED, unless overridden by way of configuration.
- If no Transactional annotation is specified on a Controller interface, the default Transactional.Annotation enum value for methods of that interface are Transactional.Attribute.REQUIRED, unless the default transaction attribute is overridden via configuration.
- If no Transactional annotation is specified on a Controller method, the default Transactional.Attribute enum value is that of the Controller interface.

The optional value specified is of type Transactional.Attribute, which is an enum. The following table lists and describes these enum values.

Enum Values	Description
MANDATORY	Indicates that a transaction is mandatory. ■ If no transaction is in progress when the method is invoked, **TransactionRequiredException** will be thrown. ■ If a transaction is in progress when the method is invoked, that transaction will be used for the method.
NEVER	Indicates that a transaction is never supported. ■ A transaction must not be in progress when the method is invoked. If a transaction is in progress, **TransactionActiveException** will be thrown. ■ If no transaction is in progress, the method proceeds without a transaction.
NOT_SUPPORTED	Indicates that a transaction is not supported. ■ If a transaction is in progress when the method is invoked, the transaction is suspended, and the method proceeds without a transaction. The transaction is resumed after the method returns. The called method has no affect on the resumed transaction. ■ If no transaction is in progress, the method proceeds without a transaction.
REQUIRED	Indicates that a transaction is required. ■ If no transaction is in progress, a transaction will be started for the method. ■ If a transaction is in progress, that transaction will be used. ■ When no Transactional.Attribute enum value is specified, the default enum value is REQUIRED, unless overridden by way of configuration.
REQUIRES_NEW	Indicates that a new transaction is required. ■ If no transaction is in progress, a transaction will be started for the method. ■ If a transaction is already in progress, that transaction will be suspended, and a new transaction will be started for the method. Once the method is completed and the new transaction is ended, the suspended transaction, if any, will be resumed. The called method has no affect on the resumed transaction.
SUPPORTS	Indicates that a transaction is supported. ■ If a transaction is in progress, it will be used. ■ If no transaction is in progress, the method proceeds without a transaction, which means data integrity is not guaranteed.

Exceptions

Exceptions listed below are defined in the following package:

```
com.magnet.transaction.exception
```

TransactionActiveException

This exception is thrown when Transactional Attribute is NEVER, but a transaction is active.

For example, a Controller interface method is invoked with transaction attribute REQUIRED, and in turn invokes a Controller interface method with transaction attribute NEVER. In this scenario, the latter method invocation will result in this exception.

TransactionRequiredException

This exception is thrown when Transactional Attribute is MANDATORY, but no transaction is active.

For example, a Controller interface method is invoked with transaction attribute NOT_SUPPORTED, and in turn invokes a Controller interface method with transaction attribute MANDATORY. The latter method invocation will result in this exception.

TransactionRolledBackException

This exception is thrown when the transaction processing the request has been rolled back or marked for rollback-only.

For example, a Controller interface method is invoked for which a transaction is started. That method invokes another Controller interface method that throws an unchecked exception (a subclass of RuntimeException), which causes the transaction to be marked for rollback-only. The first method catches the thrown unchecked exception and then invokes another Controller interface method for which the transaction is used. The last method call will cause this exception to be thrown. If the first method catches the TransactionRolledBackException and then attempts to return to the client, TransactionRolledBackException is thrown back to the client because the transaction was marked for rollback-only.

TransactionInvalidStatusException

This exception is thrown when the transaction status is invalid.

For example, an attempt was made to start a transaction when the current javax.transaction.Status value was other than NO_TRANSACTION, ACTIVE, or MARKED_ROLLBACK.

This exception should not occur under normal programming circumstances.

Container-managed Transaction

When both ControllerTransactionInterceptor and AtomikosTransactionManager are **enabled** (default), transactions are container-managed. Refer to ***Transaction Configuration Files*** on page ***89*** for description of how to configure the ControllerTransactionInterceptor and the AtomikosTransactionManager.

Within the context of the Magnet Mobile Enterprise Server, container-managed transactions are JTA transactions that are demarcated by the Magnet Mobile Enterprise Server on the boundaries of a Controller interface methods. Both AtomikosTransactionManager and ControllerTransactionInterceptor must be enabled for container-managed transactions to be in effect.

Release 2.0 does not support client-managed transactions, which means that the client application cannot begin a transaction, invoke multiple Controller interface methods that take part in that transaction, and end that transaction, either with commit or rollback. By default, when a client invokes a Controller interface method, a JTA transaction is started automatically for that method, and all work done by that method and any other methods that it calls is committed as a single transaction, assuming no exceptions are thrown, or all checked exceptions are handled. A checked exception is any exception that is not an extension of RuntimeException.

When a Controller interface method is decorated with the @Transactional annotation (refer to ***Annotations*** on page ***85***), the transactional attribute for that method is taken from that annotation.

If a Controller interface method is not decorated with the @Transactional annotation, the transactional attribute of the method is taken from the @Transactional annotation on the Controller interface. If the Controller interface is not decorated with the @Transactional annotation, the transactional attribute of the method is Transactional.Attribute.REQUIRED (unless this default is overridden in ControllerTransactionInterceptor-1.cproperties).

The transaction manager intercepts calls made on Controller interfaces. It does not intercept calls made directly on Controller implementations. Let's look at this scenario:

> Controller1 has two methods, method1 and method2, and is implemented by Controller1Impl. Controller1 is decorated with "@Transactional(Transactional.Attribute.REQUIRES_NEW)" and neither method1 nor method2 are decorated with the Transactional annotation.

If Controller1Impl.method1() invokes Controller1Impl.method2() directly, the transaction manager is bypassed when method2() is invoked, and a new transaction is not started for the call to Controller1Impl.method2().

If Controller1Impl.method1() invokes Controller1Impl.method2() via an injected Controller1 interface, the transaction manager is not bypassed, and a new transaction is started for the call to Controller1Impl.method2().

Transaction Configuration Files

You can specify values in configuration properties (cproperties) files to set policies that affect how transactions run. These cproperties files are stored in the *config.dir* directory. The following table lists and describes these cproperties files and a page reference to each of the cproperties.

cproperties Files	Description	Page
ControllerTransactionInterceptor	Used to disable the ControllerTransactionInterceptor (the transaction manager interceptor), and to change the default Transactional.Attribute value	page ***90***
AtomikosTransactionManager	Used to disable the Atomikos transaction manager, or to set Atomikos-specific properties for the Atomikos transaction manager.	page ***91***
AtomikosJdbcConnectionPool	Used to set Atomikos-specific connection pool properties.	page ***92***
DefaultJdbcConnectionPool	Used to set BasicDataSource connection pooling properties when Atomikos is not used.	page ***94***
MySqlJdbcPersister	Used to define the data store URL and credentials.	page ***98***

In addition to the cproperties files, add the following lines to the logging.properties file (stored in the *config.dir* directory) to silence log messages from Atomikos:

```
com.atomikos.level = WARNING
com.atomikos.handlers = java.util.logging.FileHandler, java.util.logging.ConsoleHandler
```

ControllerTransactionInterceptor-1.cproperties

You can specify values in this .cproperties file to disable the ControllerTransactionInterceptor (the transaction manager interceptor), and to change the default Transactional.Attribute value. This interceptor is used to perform the container-managed transactions. If disabled, container-managed transactions are not performed.

The following example shows the default configuration settings. The following table lists and describes valid values that can be specified in the *ControllerTransactionInterceptor-1.cproperties* file.

```
enabled flag=true
defaultTransactionalAttribute=REQUIRED
```

Property	Value
enabled	■ **true** (default) to enable the Controller Transaction Interceptor. ■ false to disable the Controller Transaction Interceptor. **Note**: AtomikosTransactionManager is the JTA transaction manager used by Magnet. If the AtomikosTransactionManager and the ControllerTransactionInterceptor are disabled, container-managed transactions are not performed.
defaultTransactionalAttribute	Valid values include: ■ MANDATORY indicates that transaction is mandatory. ■ NEVER indicates that transaction is never supported ■ NOT_SUPPORTED indicates that transaction is not supported ■ **REQUIRED** (default) indicates that transaction is required. ■ REQUIRES_NEW indicates that a new transaction is required ■ SUPPORTS indicates that transaction is supported **Note**: Refer to the table page *86* for detailed description for enum values.

AtomikosTransactionManager-1.cproperties

You can specify values to disable the Atomikos transaction manager, or to set Atomikos-specific properties for the Atomikos transaction manager.

Release 2.0 supports using the AtomikosTransactionManager for a single (non-XA) JDBC data source. It does not support managing transactions across multiple (XA) JDBC data sources.

The following example shows the default configuration settings. The following table lists and describes valid values that can be specified in the *AtomikosTransactionManager-1.cproperties* file.

```
enabled=true
tmUniqueName=[machine’s IP address]
defaultJtaTimeout=30000
maxActiveTransactions=500
enableLogging=false
logBaseName=tmlog
logBaseDir=data.dir
checkpointInterval=500
outputDir=data.dir
consoleLogLevel=WARNING
```

Property	Description
enabled	■ **true** (default) to enable the Atomikos Transaction Manager. ■ false to disable the Atomikos Transaction Manager. **Note**: If both the AtomikosTransactionManager and the ControllerTransactionInterceptor are enabled, transactions are container-managed; otherwise, container-managed transactions are not performed.
tmUniqueName	Specifies the transaction manager's unique name. The default is the **machine's IP address**. A time-stamp is appended to this unique name.
defaultJtaTimeout	Specifies in milliseconds the default timeout for JTA transactions. The default is **30000 ms**.
maxActiveTransactions	Specifies the maximum number of active transactions. The default is **500**. Use -1 for unlimited number of active transactions.

Property	Description
enableLogging	■ true to enable the transaction log. ■ **false** (default) to disable the transaction log. **Note**: Set to disable in a testing environment. Do not disable the transaction log on production where data integrity is essential.
logBaseName	Specifies the transactions log file base name. The default is **tmlog**.
logBaseDir	Specifies the directory in which the log files should be stored. The default is **data.dir**. **Note**: Be sure that logBaseDir directory exists.
checkpointInterval	Specifies the number of logs written between checkpoints. The default is **500**.

AtomikosJdbcConnectionPool-1.cproperties

You can specify values to set Atomikos-specific connection pool properties. This connection pool is used if Atomikos transaction manager is enabled.

The following example shows the default configuration settings. The following table lists and describes valid values that can be specified in the *AtomikosJdbcConnectionPool-1.cproperties* file.

```
enabled=true
url=jdbc:mysql://
transactionIsolation=READ_COMMITTED
borrowConnectionTimeout=
loginTimeout=
maintenanceInterval=
maxIdleTime=
maxPoolSize=100
minPoolSize=2
reapTimeout=
uniqueResourceName=MagnetDB
```

Property	Description
enabled	■ **true** (default) to enable the Atomikos JDBC connection pool. ■ false to disable the Atomikos JDBC connection pool.
url	Specifies the database URL that matches the URL listed in MySqlJdbcPersister-X.cproperties. Let's look at the following definitions, for example: AtomikosJdbcConnectionPool-1.cproperties url=jdbc:mysql:// MySqlJdbcPersister-X.cproperties url=jdbc:mysql\://localhost\:3306/ $APP_DB_NAME? In this example, the URL defined in MySqlJdbcPersister-X.cproperties will be used.
transactionIsolation	Specifies the isolation level that applies to the transaction. Valid values include: ■ **READ_COMMITTED** (default) indicates that dirty reads are prevented; non-repeatable reads, and phantom reads can occur. ■ READ_UNCOMMITTED indicates that dirty reads, non-repeatable reads, and phantom reads can occur. ■ REPEATABLE_READ indicates that dirty reads and non-repeatable reads are prevented; phantom reads can occur. ■ READ_SERIALIZABLE indicates that dirty reads, non-repeatable reads, and phantom reads are prevented. **Note**: Refer to the javadocs for *java.sql.Connection* for more information on transaction isolation.
borrowConnectionTimeout	This optional property specifies the maximum amount of time in seconds the pool will block waiting for a connection to become available in the pool when it is empty.
loginTimeout	This optional property specifies the maximum time in seconds that this data source can wait while attempting to connect to a database.
maintenanceInterval	This optional property specifies the maintenance interval for the pool maintenance thread.

Property	Description
maxIdleTime	This optional property specifies the maximum length of time in seconds that a connection can stay in the pool before being eligible for being closed during pool shrinking.
maxPoolSize	Specifies the maximum number of connections The default is **100**.
minPoolSize	Specifies the minimum number of pooled connections. The default is **2**.
reapTimeout	Specifies the length of time in seconds that the connection pool will allow a connection to be borrowed before claiming it back.
uniqueResourceName	Specifies an arbitrary value that identifies this data source. The default is **MagnetDB**.

DefaultJdbcConnectionPool-1.cproperties

You can specify values to set BasicDataSource connection pooling properties when Atomikos is not used. By default when Atomikos not used, Apache Commons DBCP BasicDataSource is used for connection pooling.

The following example shows the default configuration settings. The following table lists and describes valid values that can be specified in the *DefaultJdbcConnectionPool-1.cproperties* file.

```
enabled=true
url=jdbc:mysql://
transactionIsolation=READ_COMMITTED
defaultCatalog=
initialSize=
logAbandoned=
maxActive=-1
maxIdle=
maxOpenPreparedStatements=-1
maxWait=
minEvictableIdleTimeMillis=
minIdle=
numTestsPerEvictionRun=
removeAbandoned=
removeAbandonedTimeout=
testOnBorrow=false
testOnReturn=false
testWhileIdle=true
timeBetweenEvictionRunsMillis=30000
validationQueryTimeout=
poolPreparedStatements=true
```

Property	Description
enabled	■ **true** (default) to enable the BasicDataSource JDBC connection pool. ■ false to disable the BasicDataSource JDBC connection pool.
url	Specifies the database URL that matches the URL listed in MySqlJdbcPersister-X.cproperties. Let's look at the following definitions, for example: defaultAtomikosJdbcConnectionPool-1.cproperties url=jdbc:mysql:// MySqlJdbcPersister-X.cproperties url=jdbc:mysql\://localhost\:3306/$APP_DB_NAME? In this example, the URL defined in MySqlJdbcPersister-X.cproperties will be used.

Property	Description
transactionIsolation	Specifies the isolation level that applies to the transaction. Valid values include: ■ **READ_COMMITTED** (default) indicates that dirty reads are prevented; non-repeatable reads, and phantom reads can occur. ■ READ_UNCOMMITTED indicates that dirty reads, non-repeatable reads, and phantom reads can occur. ■ REPEATABLE_READ indicates that dirty reads and non-repeatable reads are prevented; phantom reads can occur. ■ READ_SERIALIZABLE indicates that dirty reads, non-repeatable reads, and phantom reads are prevented. **Note**: Refer to the javadocs for *java.sql.Connection* for more information on transaction isolation.
defaultCatalog	This optional property specifies the default "catalog" of connections created by this pool.
initialSize	This optional property specifies the initial number of connections that are created when the pool is started.
logAbandoned	This optional property specifies a flag. ■ true indicates to log stack traces for application code that abandoned a statement or connection. ■ false indicates otherwise.
maxActive	Specifies the maximum number of active connections that can be allocated from this pool at the same time. The default is **-1**, meaning there is no limitation on the number of active connections.
maxIdle	This optional property specifies the maximum number of open statements that can be allocated from the statement pool at the same time.
maxOpenPreparedStatements	Specifies the maximum number of open statements that can be allocated from the statement pool at the same time. The default is **-1**, meaning there is no limitation on the number of open statements.

Property	Description
maxWait	This optional property specifies the maximum length of time in milliseconds that the pool will wait for a connection to be returned before throwing an exception.
minEvictableIdleTimeMillis	Specifies the minimum length of time an object may sit idle in the pool before it is eligible for eviction by the idle object evictor (if any).
minIdle	This optional property specifies the minimum number of idle connections in the pool.
numTestsPerEvictionRun	This optional property specifies the number of objects to examine during each run of the idle object evictor thread (if any).
removeAbandoned	This optional property specifies a flag. ■ true indicates to remove abandoned connections if they exceed the removeAbandonedTimout. ■ false indicates otherwise.
removeAbandonedTimeout	This optional property specifies a timeout in seconds before an abandoned connection can be removed.
testOnBorrow	■ true indicates that objects will be validated before being borrowed from the pool. ■ **false** (default) indicates otherwise.
testOnReturn	■ true indicates that objects will be validated before being returned to the pool. ■ **false** (default) indicates otherwise.
testWhileIdle	■ **true** (default) indicates that objects will be validated by the idle object evictor (if any). ■ false indicates otherwise.
timeBetweenEvictionRunsMillis	Specifies the number of milliseconds to sleep between runs of the idle object evictor thread. The default is **30000** ms.
validationQueryTimeout	This optional property specifies a timeout in seconds before connection validation queries fail.
poolPreparedStatements	■ **True** (default) indicates to pool statements (true) ■ false indicates otherwise.

MySqlJdbcPersister-1.cproperties

This configuration file defines the data store URL and credentials. For example,

```
url=jdbc:mysql://localhost/jdbcTest1?createDatabaseIfNotExist=true
user=jdbcTestU
password=jdbcTestP
transactionsSupported=true
```

Property	Description
url	Specifies the path to MySQL data store.
user	The name of user for accessing the data store.
password	The password associated with the user name.
transactionsSupported	■ true indicates transactions are supported. ■ false indicates otherwise.

Magnet

8. Connecting to Third-Party Services

The Magnet App Server currently provides three social controllers that you can include in your project to enable access to third-party services. These controllers include Salesforce, LinkedIn, and Facebook.

These controllers are provided if you have included them when you first created your project in the Developer Factory. You can add third-party services to your application if they were not included earlier (refer to ***Modifying a Project*** on page *22* for information). The generated project contains source code for these controllers, which you can modify for your environment.

Using a Salesforce Controller

The Salesforce controller is generated from the Salesforce's enterprise.wsdl. It exposes the Salesforce SOAP API as controller methods. The SOAP API allows performing various operations on your Salesforce Connected Apps, such as create, retrieve, update, or delete records. Go to Salesforce Web site for Connected Apps information (http://wiki.developerforce.com/page/Connected_Apps).

Enabling SSL

A server running the Salesforce controller must be configured to be accessed via SSL. You must enable SSL on your local servers that you are using for testing purposes. In a production environment and when your server is behind a load balancer, SSL can be disabled.

Modifying cproperties Files

The Salesforce controller is built with OAuth support for user authentication and secure HTTP (HTTPS) connection. When you create a project using the Developer Factory, all configurations are generated for you based on the input you provide. When you need to change these settings (such as endpoint address), they are specified in the configuration cproperties files. These cproperties files are stored in the *config.dir* directory.

SoapPort-sforce.cproperties

The endpoint address is specific to the remote connected app that you have created. You need to change the endpoint address in the *SoapPort-sforce.cproperties* file to override the controller endpoint uri defined in Salesforce WSDL.

The following example shows default settings of the *SoapPort-sforce.cproperties* file.

```
target = com.magnet.connect.sforce.api.SforceController
endpointAddress = <salesforce application endpoint address>
```

Parameter	Value
target	This is automatically populated by the Developer Factory. **Note**: Do not change this value.
endpointAddress	This is the endpoint address specific to the remote app.

To get your connected app endpoint address:

1 Query Salesforce to get the OAuth access token.

For example,

```
curl -X POST -H 'X-PrettyPrint:1' -d "grant_type=password" -d
"client_id=4CLI2EnT6id" -d "client_secret=0123456789" -d"
username=user@example.com" -d "password=passw0rd" https://
login.salesforce.com/services/oauth2/token
```

The access_token and id are displayed, as shown in the following example.

```
{
  "id" : "https://login.salesforce.com/id/00Di0000000JDcYEAW/
005i0000000m5zFAAQ",
  "issued_at" : "1369244452159",
  "instance_url" : "https://na15.salesforce.com",
  "signature" : "JaAGkaAxxZVUFYsuoK5XpKu2guNd69RGZ6G4npygHzw=",
  "access_token" :
"00Di0000000JDcY!AR4AQPDdBna8ihJiT0ODCuntIJyg3GNCKed.SlKZ4a.lZKUl.5fjswRSnM3G
rfbS0ChFfb0RPC.9M5Tn0qALkdvfqRHBpKAt"
}
```

2 Get the endpoint address with the id and access token obtained from step 1, as shown in the following example.

```
curl -i -H 'X-PrettyPrint:1' -H 'Authorization: Bearer
00Di0000000JDcY!AR4AQPDdBna8ihJiT0ODCuntIJyg3GNCKed.SlKZ4a.lZKUl.5fjswRSnM3Gr
fbS0ChFfb0RPC.9M5Tn0qALkdvfqRHBpKAt
' -L https://login.salesforce.com/id/00Di0000000JDcYEAW/005i0000000m5zFAAQ
```

3 Click the enterprise URL.

The following example shows the endpoint address.

```
https://na15-api.salesforce.com/services/Soap/c/{version}/00Di0000000JDcY
```

4 Change version to **27.0**, the version supported by the Salesforce controller, as shown in the following example.

```
https://na15-api.salesforce.com/services/Soap/c/27.0/00Di0000000JDcY
```

This is the endpoint address that you set in the *SoapPort-sforce.cproperties* file.

SFDCOAuthConfig-1.cproperties

You can modify clientId, clientSecret, and redirectUri parameters in the *SFDCOAuthConfig-1.cproperties* file. Other parameters contain default values that should not be changed.

The following example shows default settings of the *SFDCOAuthConfig-1.cproperties* file.

```
enabled=true
accessTokenUri=https://login.salesforce.com/services/oauth2/token
authorizationUri=https://login.salesforce.com/services/oauth2/authorize
clientId=
clientSecret=
redirectUri=
scope=full
```

Parameter	Value
enabled	When true (default), the Salesforce controller is able to authenticate against the Salesforce connected app. **Note**: Do not change this value.
accessTokenUri	This is automatically populated by the Developer Factory. **Note**: Do not change this value.
authorizationUri	This is automatically populated by the Developer Factory. **Note**: Do not change this value.
clientId	The key that was generated from your Connected App.
clientSecret	The Secret Key that was generated from your Connected App.
redirectUri	The Callback URL that was defined for remote access when you created your Connected App, and is automatically populated by the Developer Factory. Salesforce requires that the redirectUri be a HTTPS URI.
scope	Specifies what data your application can access. By default it is set to "full." For scope values, visit http://help.salesforce.com/apex/HTViewHelpDoc?id=remoteaccess_oauth_scopes.htm

Mapping the Saleforce Controller API

For every SOAP API type, the Saleforce controller exposes an associated type appended with the "Node" suffix. You can easily map the Salesforce native Java type to Magnet types using this rule of thumb.

For SOAP API documentation and mapping information, visit:

```
http://www.salesforce.com/us/developer/docs/api/Content/
sforce_api_calls_list.htm
```

Using a LinkedIn Controller

The linkedIn controller enables obtaining your LinkedIn connections and sharing updates from a mobile device. The linkedIn controller interacts with linkedIn using the LinkedIn REST API. The linkedIn controller is built as a part of the Magnet App Server, and its source code is available for you to implement the LinkedIn service on the mobile devices. The linkedIn controller is implemented with OAuth2.0 for authentication and security.

```
@Contract
@RemoteAccessible
public interface LinkedInController
extends com.magnet.controller.spi.Controller
```

Methods	Description	Page
searchConnections	Search contacts of the logged in user.	page *10 3*

searchConnections

Search contacts of the logged in user. If any search fields are null, then they are not used as constraints. If more than one field is specified, all fields will act as constraints (AND not OR).

```
com.magnet.common.api.IterableResource<LinkedInUser> searchConnections(String
keywords,
   String firstName,
   String lastName)
```

Parameters

The following parameters are optional.

- *keyword* indicates a word in any field to be searched.
- *firstName* indicates the contact's first name to be searched.
- *lastName* indicates the contact's last name to be searched.

Both first and last names must be an exact match; however, they are not case-sensitive.

Return

An IterableResource over matching contacts based on the optional parameters.

Example

```
@Contract
@RemoteAccessible
public interface LinkedInController extends Controller {
    @Path("/linkedin/contacts")
    @GET
    IterableResource<LinkedInUser> getConnections();

    @Path("/linkedin/query")
    @GET
    IterableResource<LinkedInUser> searchConnections(@Optional
@QueryParam("keywords") String keywords,
                             @Optional @QueryParam("firstName") String firstName,
                             @Optional @QueryParam("lastName") String lastName);
}
```

Modifying the cproperties File

The linkedIn controller is built with OAuth support for user authentication. When you create a project using the Developer Factory, all configurations are generated for you based on the input you provide. Other security settings (such as server endpoints) are specified in the *LNKOAuthConfig-1.cproperties* file, which is stored in the *config.dir* directory.

LNKOAuthConfig-1.cproperties

You can modify clientId, clientSecret, and redirectUri parameters in the *LNKOAuthConfig-1.cproperties* file. Other parameters contain default values that should not be changed. The following example shows settings of the *LNKCOAuthConfig-1.cproperties* file.

```
enabled=true
clientId=
clientSecret=
redirectUri=https://ec2-184-73-100-147.compute-1.amazonaws.com:8090/
MagnetOAuthServlet
```

Secure HTTP is not required with the LinkedIn controller. Magnet recommends using HTTPS as default as it is required by the Salesforce controller. A server running both controllers cannot support both HTTP and HTTPS.

Parameter	Value
enabled	When true (default), the LinkedIn controller is able to authenticate against the LinkedIn connected app. **Note**: Do not change this value.
clientId	The API key that was generated from your Connected App.
clientSecret	The Linkedin App Secret Key that was generated from your Connected App.
redirectUri	The Callback URL that was defined for remote access when you created your Connected App, and is automatically populated by the Developer Factory. **Note**: Do not change this value.

Using a Facebook Controller

The Facebook controller enables searching for friends and sharing updates from a mobile device. The Facebook controller interacts with Facebook using the Facebook REST API. The Facebook controller is built as a part of the Magnet App Server, and its source code is available for you to implement the Facebook service on the mobile devices. The Facebook controller is implemented with OAuth2.0 for authentication and security.

```
@Contract
@RemoteAccessible
public interface FacebookController
extends com.magnet.controller.spi.Controller
```

The following table lists and describes the Facebook Controller methods and a page reference to each of the method.

Methods	Description	Page
searchFriends	Search friends of the logged in user.	page *106*
setStatus	Set the status for the logged in user.	page *106*

searchFriends

Search friends of the logged in user. If any search fields are null, then they are not used as constraints. If more than one field is specified, all fields will act as constraints (AND not OR).

```
com.magnet.common.api.IterableResource<FacebookUser> searchFriends(String keyword,
                                                                   String firstName,
                                                                   String lastName)
```

Parameters

- *keyword* indicates a word in any field to be searched.
- *firstName* indicates the friend's first name to be searched.
- *lastName* indicates the friend's last name to be searched.

Both first and last names must be an exact match; however, they are not case-sensitive.

Return

An IterableResource over the user's contacts.

setStatus

Set the status for the logged in user.

```
String setStatus(String status)
```

Parameters

status indicate the status line.

Return

The ID of the status.

Setting up cproperties Files

The Facebook controller is built with OAuth support for user authentication and secure HTTP (HTTPS) connection. When you create a project using the Developer Factory, all configurations are generated for you based on the input you provide. Other security settings (such as server endpoints) are specified in the *FBOAuthConfig-1.cproperties* file, which is stored in the *config.dir* directory.

FBOAuthConfig-1.cproperties

You can modify clientId, clientSecret, and redirectUri parameters in the *FBOAuthConfig-1.cproperties* file. Other parameters contain default values that should not be changed. The following example shows settings of the *FBOAuthConfig-1.cproperties* file.

```
enabled=true
clientId=
clientSecret=
redirectUri=https://ec2-184-73-100-147.compute-1.amazonaws.com:8082/
MagnetOAuthServlet
```

Parameter	Value
enabled	When true (default), the Facebook controller is able to authenticate against the Facebook connected app. **Note**: Do not change this value.
clientId	The key that was generated from your Connected App.
clientSecret	The Secret Key that was generated from your Connected App.
redirectUri	The Callback URL that was defined for remote access when you created your Connected App, and is automatically populated by the Developer Factory. **Note**: Do not change this value.

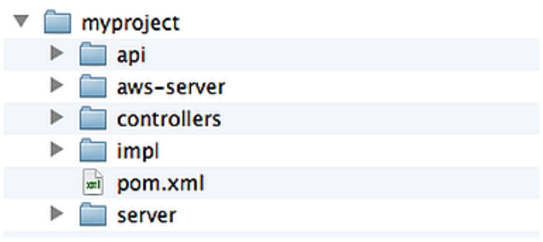
9. Compiling Your Project

After you have downloaded your project from the Developer Factory to your local environment, you can modify your project, compile it, then upload the new project to the Developer Factory for new assets generation or to the Amazon cloud for deployment.

A Maven project is structured using different modules. Each module includes files that are related to your project. The separation of modules allows artifacts (that are to be loaded on mobile devices or on server) to be easily segregated. (Visit http://maven.apache.org/ for information.)

Understanding the Project Structure

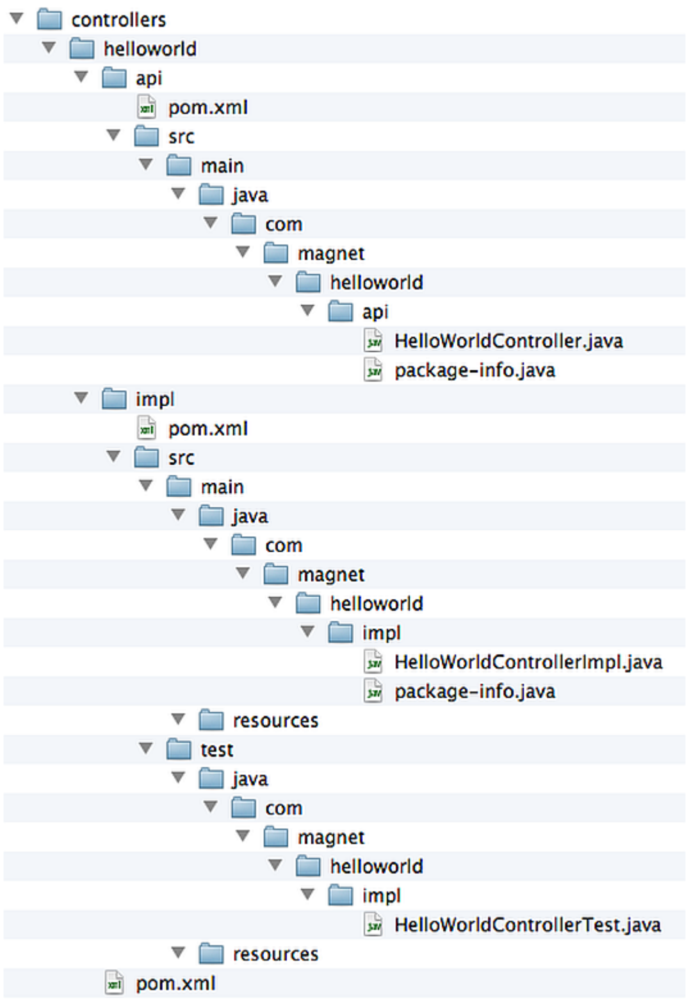
The parent project is defined in the top-level pom.xml, which contains the build configuration for the entire project, the version of the dependencies, and the plugin configuration. Here is a project layout:


```
myproject
  api
  aws-server
  controllers
  impl
  pom.xml
  server
```

controllers

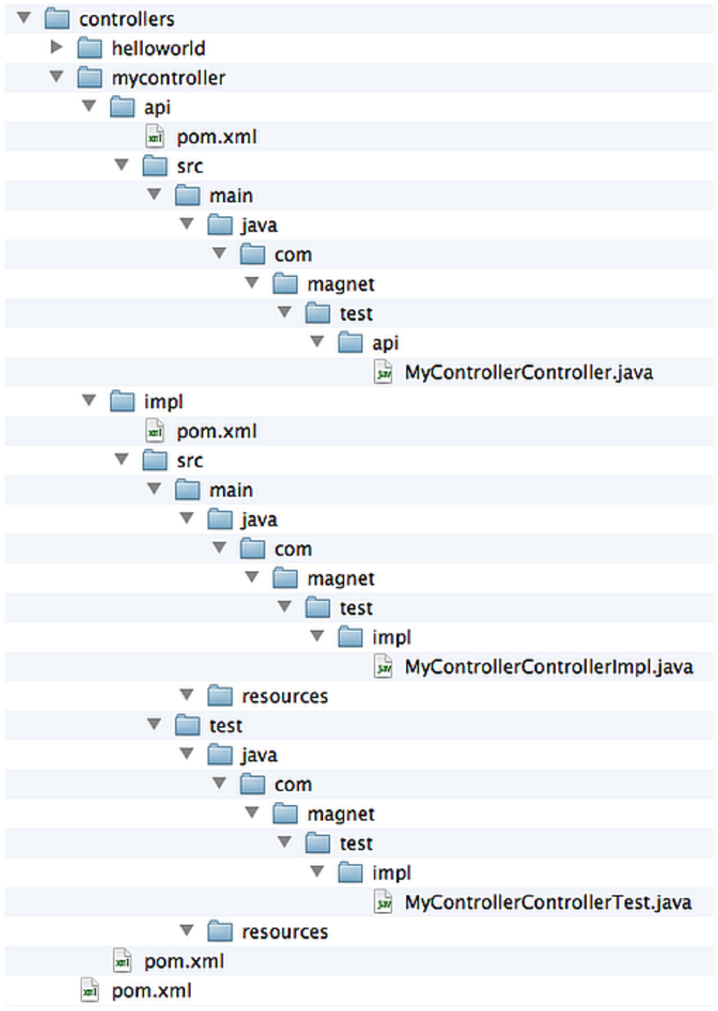
controllers is the parent module for all controllers modules, including API and impl as submodules, as shown in the following diagram. The Controller implementation conventionally runs on the server side, and clients use the API interface to proxy to it over a RESTful protocol.

This is the parent module where the Developer Factory generates the out-of-the-box controllers that you chose to include in the initial project. As shown in the following diagram, a helloworld controller was selected when the project was created in the Developer Factory.



```
controllers
  helloworld
    api
      pom.xml
      src
        main
          java
            com
              magnet
                helloworld
                  api
                    HelloWorldController.java
                    package-info.java
    impl
      pom.xml
      src
        main
          java
            com
              magnet
                helloworld
                  impl
                    HelloWorldControllerImpl.java
                    package-info.java
          resources
        test
          java
            com
              magnet
                helloworld
                  impl
                    HelloWorldControllerTest.java
          resources
    pom.xml
```

When adding new controllers, Magnet recommends to add new controllers under the controllers directory, and follow the same structure as the helloworld controller as shown in the following diagram, where you separate api and implementation in two different submodules. This way, the build will produce two distinct modules for the controller api and implementation. This allows the api to be deployed separately from the implementation, as it is the case on mobile devices.

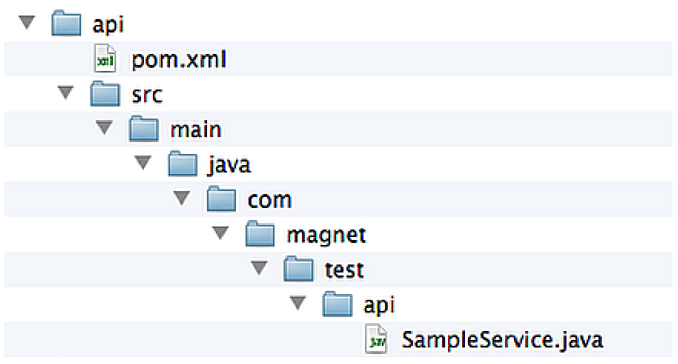


```
▼ controllers
    ▶ helloworld
    ▼ mycontroller
        ▼ api
            pom.xml
            ▼ src
                ▼ main
                    ▼ java
                        ▼ com
                            ▼ magnet
                                ▼ test
                                    ▼ api
                                        MyControllerController.java
        ▼ impl
            pom.xml
            ▼ src
                ▼ main
                    ▼ java
                        ▼ com
                            ▼ magnet
                                ▼ test
                                    ▼ impl
                                        MyControllerControllerImpl.java
                    ▼ resources
                ▼ test
                    ▼ java
                        ▼ com
                            ▼ magnet
                                ▼ test
                                    ▼ impl
                                        MyControllerControllerTest.java
                    ▼ resources
        pom.xml
    pom.xml
```

After a new controller is created, ensure that both api and impl controller modules are added as Maven compile dependencies in the server/pom.xml, and that api controller module is added in the aws-server/pom.xml as provided dependency (refer to ***aws-server*** on page ***114*** for information).

api

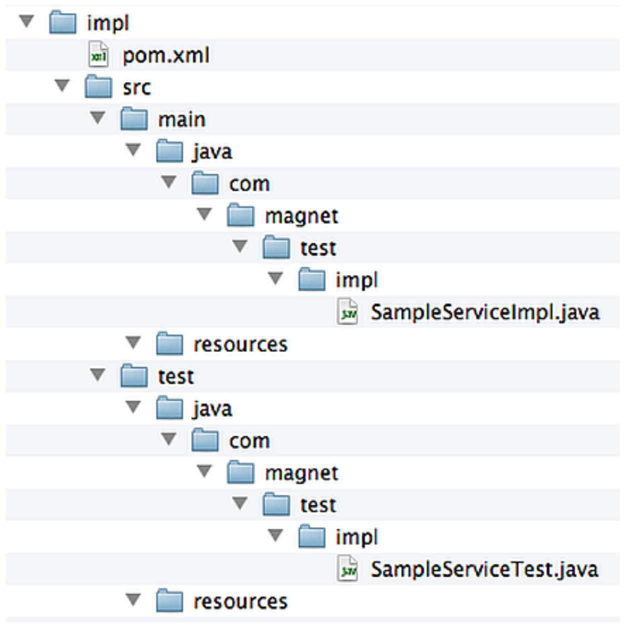
The **api** module contains non-controller custom interfaces (such as services) of your project.The api is the primary package your IDE project depends on and compiles against.



```
▼ api
      pom.xml
  ▼ src
    ▼ main
      ▼ java
        ▼ com
          ▼ magnet
            ▼ test
              ▼ api
                  SampleService.java
```

impl

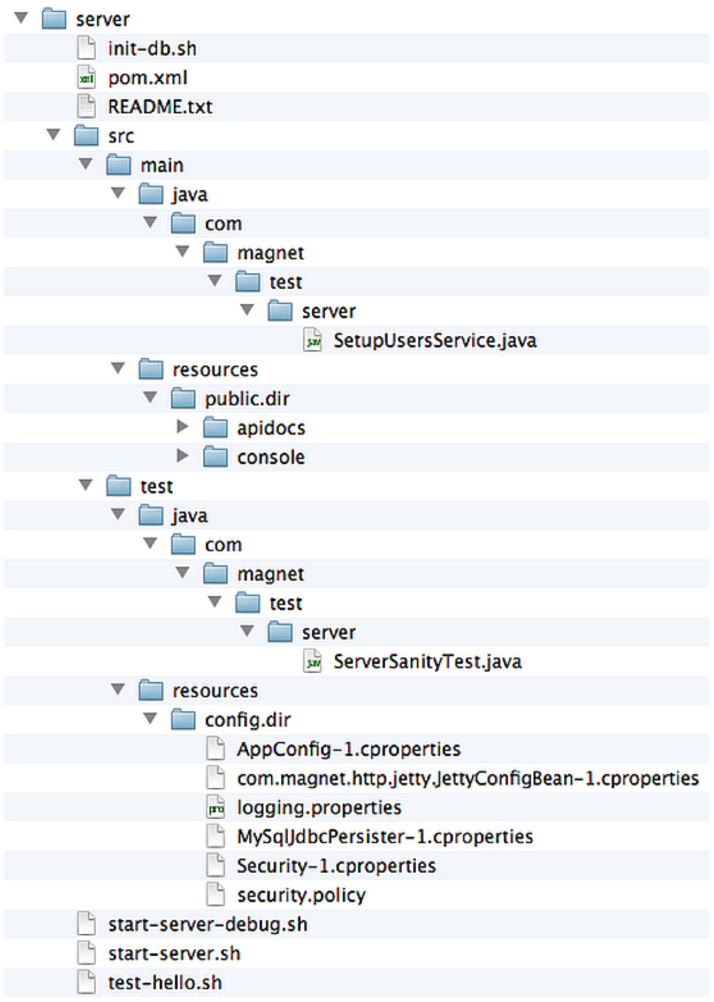
The **impl** module contains api implementation and unit tests for your custom implementation. By default, the api and impl submodules contain a sample service interface, implementation and unit test.



```
▼ impl
      pom.xml
  ▼ src
    ▼ main
      ▼ java
        ▼ com
          ▼ magnet
            ▼ test
              ▼ impl
                  SampleServiceImpl.java
      ▼ resources
    ▼ test
      ▼ java
        ▼ com
          ▼ magnet
            ▼ test
              ▼ impl
                  SampleServiceTest.java
      ▼ resources
```

server

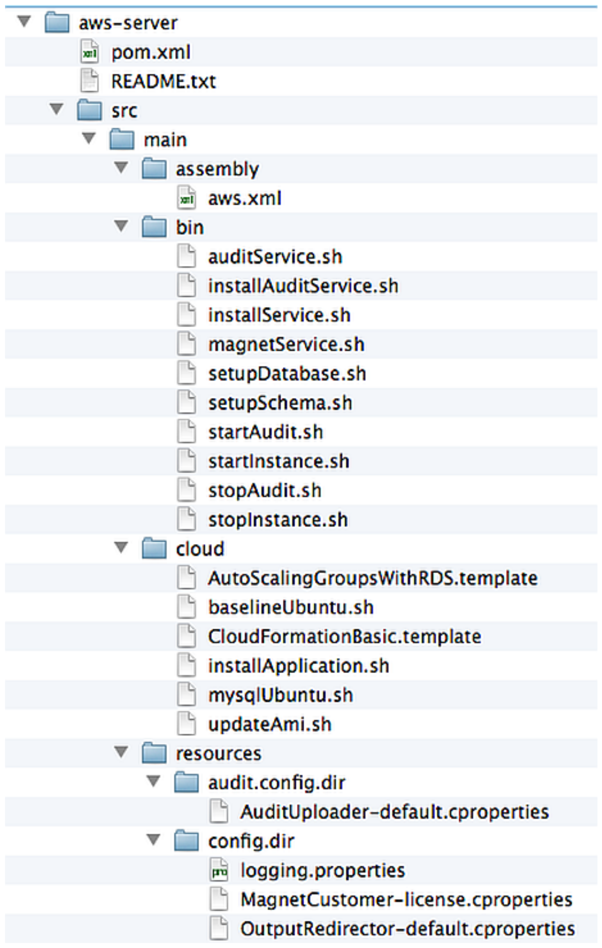
The **server** module produces a shaded jar that combines all server artifacts (api and implementation modules) in a single standalone jar. As shown in the following diagram, the server module contains the server configuration, scripts to locally start and debug the server, and tests to verify the implementation of the server. Any functional server tests should be put in this module under src/test/java.



```
▼ server
    init-db.sh
    pom.xml
    README.txt
  ▼ src
    ▼ main
      ▼ java
        ▼ com
          ▼ magnet
            ▼ test
              ▼ server
                  SetupUsersService.java
      ▼ resources
        ▼ public.dir
          ▶ apidocs
          ▶ console
    ▼ test
      ▼ java
        ▼ com
          ▼ magnet
            ▼ test
              ▼ server
                  ServerSanityTest.java
      ▼ resources
        ▼ config.dir
            AppConfig-1.cproperties
            com.magnet.http.jetty.JettyConfigBean-1.cproperties
            logging.properties
            MySqlJdbcPersister-1.cproperties
            Security-1.cproperties
            security.policy
    start-server-debug.sh
    start-server.sh
    test-hello.sh
```

aws-server

The **aws-server** module produces a server zip file that you can upload to the Developer Factory to produce mobile assets, or deployed to aws instances. This zip file contains all controller API JARS and cproperties files that are located in the server module at server/src/test/resources. The zip file is located under **<project-root>/aws-server/target/<project-name>-server-aws-<version>-deploy.zip.**



```
aws-server
    pom.xml
    README.txt
    src
        main
            assembly
                aws.xml
            bin
                auditService.sh
                installAuditService.sh
                installService.sh
                magnetService.sh
                setupDatabase.sh
                setupSchema.sh
                startAudit.sh
                startInstance.sh
                stopAudit.sh
                stopInstance.sh
            cloud
                AutoScalingGroupsWithRDS.template
                baselineUbuntu.sh
                CloudFormationBasic.template
                installApplication.sh
                mysqlUbuntu.sh
                updateAmi.sh
            resources
                audit.config.dir
                    AuditUploader-default.cproperties
                config.dir
                    logging.properties
                    MagnetCustomer-license.cproperties
                    OutputRedirector-default.cproperties
```

By default, the zip file contains all controller API JARs that were created for you by the Developer Factory (based on your input). If you add new controllers submodules for which you want to generate mobile assets, you must include controllers API module as provided dependencies in the aws-server's pom.xml.

For example:

If you add the new mycontroller mentioned earlier, add the following dependency in the <project-root>/aws-server/pom.xml

```
<dependencies>
...
    <dependency>
        <groupId>com.example</groupId>
        <artifactId>mycontroller-api</artifactId>
        <version>1.0.0</version>
    </dependency>
...
</dependencies>
```

Compiling Your Maven Project

1. Compile it with the **mvn install** command.
2. Start the server with the script located under the server module.

   ```
   sh ./start-server.sh
   ```

3. Verify that controllers are registered on the Magnet Server.

   ```
   http://localhost:<port>/rest/controllers.json
   ```

   Where port is the same port you entered in Developer Factory.

Change to https if SSL is enabled on the magnet Mobile Enterprise Server or it is specified in Developer Factory.
When prompted, enter the same user credential as that of Developer Factory.

4. If your project contains the helloworld controller, verify that it works by running the script:

   ```
   sh ./test-hello.sh
   ```

   "Hello Magnet" should echo back to you.