PLE Server Application Development Kit

User's Manual

Andrei Birjukov, Ivari Horm

Table of Contents

About This Document

This manual describes the development kit components and server application development process for Artec Triskan TS8 mobile data terminal.

Intended Audience

This document is intended for software engineers and system integrators who are planning to develop PiccoLink ™ compatible applications running on terminal server and hosting Triskan TS8 devices equipped with PLE firmware.

Required Skills

To use this development kit, a developer must be familiar with the following issues and possess the following skills.

- General principles of client-server software architecture and TCP/IP communication
- Java SE skills and knowledge of API for Java 6

Conventions

This document uses the following conventions.

`Courier`	Used to identify source code samples, commands and their parameters, expressions, data types, etc.
Italic	Used for emphasis and identification of file names and new terms.

References

The following documents are referenced from this document.

1. SADK library API documentation.
2. PLE Protocol Specification.

PiccoLink ™ is a trademark of Nordic Identification OY.

Vocabulary

Term	*Definition*
JDT	Java Development Tools, refers to the Eclipse IDE JDT distribution.
PLE	PiccoLink™ Emulation, refers to a compatible implementation of a terminal protocol.
RFID	Radio Frequency Identification, refers to RFID reader function of the Triskan TS8 terminal.
SADK	Server Application Development Kit, a library provided by Artec to help implementing server-side applications for Triskan TS8 terminals.
SIM	Subscriber Identification Module, refers to the GSM SIM card.

1. Introduction

Basic Triskan TS8 unit comes with the PLE firmware that integrates a terminal emulation application running a PiccoLink-compatible protocol stack. The Triskan terminal communicates with a server backend application via GPRS or WLAN bearers. The backend application implements a customer specific business logic and a database, and provides the presentation interface to Triskan terminals by means of specially crafted forms sent over a binary PiccoLink compatible PLE protocol.

Server Application Development Kit (SADK) is designed to simplify the development of server applications, enabling for a rapid deployment of Triskan PLE terminals. The following principles are followed throughout the development cycle.

- Simplicity. The server architecture was designed from scratch, allowing for a clean and modular implementation. The developer is conveniently served with a small number of well-documented interfaces that are exposed by the SADK.
- Rapid development. Comfortable Eclipse IDE, local and remote debugging support, Java type safety, object and interface oriented approach of SADK simplifies development of server applications.
- Cross-platform. Being completely portable, the server supports Windows, Linux, Mac OS X and other OS-es capable of running Java SE virtual machine.
- UI abstractions. Application developer is not burdened with internals of the communication protocol. Instead, SADK provides convenient abstractions for application forms and input fields.
- Multiple online terminals. The server is multi-threaded, enabling for a large quantity of simultaneous connections without a significant performance penalty. Each Triskan terminal connection is isolated: incoming data is processed separately, keeping the terminal context on per-connection basis.
- Context persistence. Application instance with user data may be preserved on the server when a terminal is disconnected unexpectedly (e.g. due to coverage issues). Upon reconnection, current form and associated data will be restored automatically.
- Multiple applications, single server. The server supports many different applications available simultaneously on one TCP port. Before launch, the specific application is looked up from the database basing on the terminal serial number.

- Self-contained. Java SE runtime and MySQL database driver are integrated into SADK, no other dependencies are required.

2. Package Contents

SADK is distributed as a ZIP archive file containing all necessary documentation, source code, libraries and samples. Given below is the list of files included with SADK.

File	*Description*
doc/manual.pdf	PLE Server Application Development Kit User's Manual (this document)
doc/ple_protocol.pdf	PLE Protocol Specification
doc/sadk-doc.zip	Full documentation for SADK classes.
doc/license.txt	SADK distribution license (MIT license)
src/pleSADK-*nn*.zip	SADK Java source code with examples
jar/pleSADK-*nn*.jar	Pre-built SADK library JAR without source code.
jar/pleServer.ini	Server configuration file for stand-alone daemon setup

2.1. System Requirements

Given below are technical requirements for development of server based applications with SADK.

- PC running Windows XP/Vista/7, Linux (32-bit, 64-bit) or Mac OS X
- Java 6 SE Development Kit (http://java.sun.com/javase/downloads)
- NetBeans IDE for Java SE (http://www.netbeans.org/downloads)

  -or-
- Eclipse IDE for Java Developers (http://www.eclipse.org/downloads)

The following is also required to deploy and test developed applications:

- Triskan TS8 mobile data terminal with integrated GPRS communication module.
- Operator contract or a prepaid GSM SIM card with GPRS traffic allowed.
- External IP address with possibility of mapping an inbound TCP port to your development machine (or private network agreement with a cellular operator).
  -or-
- Triskan TS8 mobile data terminal with integrated WLAN communication module.
- Standard IEEE 802.11b/g/n access point or router.

- Possibility of mapping an inbound TCP port to your development machine.

3. Installing SADK

Extract the provided redistributable package (*sadk-revnn.zip*) to any suitable location on your development machine. Further, extract the SADK source code package located under *src/pleSADK-nn.zip*. Extract the library API documentation located under *doc/sadk-doc.zip*.

Finally, import the SADK projects into your Java development IDE.

For NetBeans IDE for Java SE:

- Select File → New Project
- Choose the option "Java Project with Existing Sources"
- Browse to the location where you extracted the SADK source code (*pleSADK-nn.zip*).
- Check the *Open as Main Project* option and click Open Project
- Explore the newly appeared pleSADK project – the IDE builds it automatically.
- The following compile-time libraries must be available for a successful build (Select File → Project Properties → Libraries):
    - MySQL JDBC connector (mysql-connector-java-NNN.jar)
    - EclipseLink(JPA 2.0)
- Note that default installation from Linux repositories (i.e. Ubuntu) does not include the EclipseLink nor MySQL JDBC connector.

For Eclipse JDT IDE:

- Select File → Import → General / Existing Projects into Workspace
- Click Next and browse to the location where you extracted the SADK source code (*pleSADK.zip*)
- Click Finish and explore the newly appeared pleSADK – the IDE builds it automatically.

For further development, you might not need the complete source code of SADK classes. In such case, feel free to create separate projects for your applications and link them to the pre-built JRE library version of SADK (*pleSADK-nn.jar*).

4. PLE Terminal Application Concept

From a general point of view, an application that employs PLE protocol is based on a client-server model. The data model and complete business logic of the PLE application reside on the server. User interface of the application is based on text forms and is rendered on remote Triskan terminals via special messages of the PLE protocol. The Triskan device thus acts as a simple terminal interpreting server messages and displaying text as instructed. Also, user input from both terminal keypad and integrated readers is sent back to the server for further processing.

The following constraints apply to PLE based applications.

- All applications are inherently on-line, assuming that network connection is available at all times.
- Server implements full business logic of the application: processes data and makes decisions.
- All application data is stored in a server database.
- Terminal screen contains text, rendered via fixed width font.
- Visible size of the terminal screen contains 20 rows and 8 columns of characters, total buffer size is 20x12, vertical scrolling is allowed.
- 1D/2D scanner and RFID reader input is redirected to on-screen form fields.
- Server has limited control of Triskan peripherals.
- Local storage of Triskan terminal is not accessible to applications.

Firmware of the Triskan terminal takes care of power management and networking, keeping a connection to server established. These and other basic tasks are running in the background of the terminal and do not interfere with user applications. Developer does not need make modifications to Triskan firmware and may concentrate on the application functionality instead; only connectivity related parameters may need changing.

5. Running Demo Applications

All example application code is located in the *ple.application* package. There are two samples provided with this version of SADK.

- Default application launcher: ple.application
  Provides menu entries to launch different applications located in subpackages under this package.

- Hotel check-in demonstration: ple.application.checkin
  This application implements a simple guest check-in screen. User scans a guest's badge and the status is sent back to the terminal and reflected in the table of a GUI front-end.
- Ticket control demonstration: ple.application.checkticket
  This application implements a simple ticket validation routine. User scans a ticket ID and the ticket status is sent back to the terminal. GUI front-end allows adding new tickets.
- Warehouse inventory demonstration: ple.application.warehouse
  Implements a simple inventory solution. Items can be added to warehouse both from the terminal and GUI front-end. The inventory can later be performed by scanning the item entering the proper quantity. The checkout routine enables to quickly scan items and reduce their quantity in the warehouse.
- Field test demonstration: ple.application.fieldtest
  Shows different input fields and their behaviour provided by PiccoLink protocol and implemented in Triskan TS8 firmware.
- Protocol test demonstration: ple.application.protocoltests
  Provides demos for testing various PiccoLink protocol capabilities.

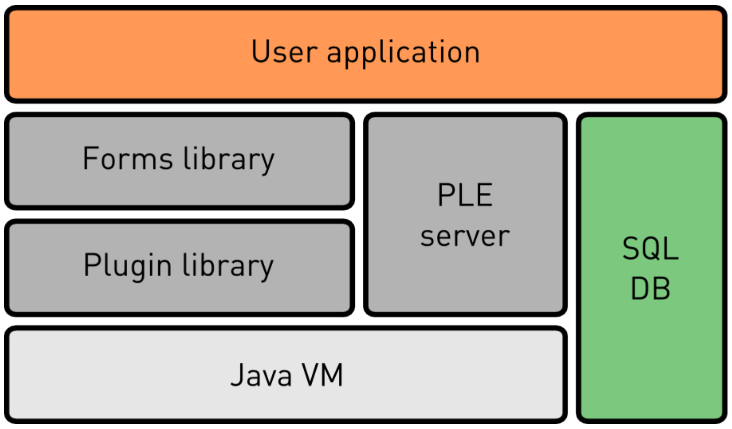
The source code of provided demo applications is organized as follows.

- *<PackageName>* class – provides the entry point to the application. This class should be specified in configuration file or called dynamically from another applicaton in order to launch the launch the application instance.
- *Database* class – implements simple data model pattern. If application uses a database the database to object model mapping is done here.
- *DefaultForm* class – the main form that is loaded in Triskan terminal when the application is first started.
- *Form* classes – implement PLE input forms and contain application business logic.

6. Architecture of SADK

SADK library is built in a modular fashion. It provides a number of classes to build a PLE application server that hosts a user application. Also, user is provided with flexible forms and plugin libraries that create a foundation for building diverse hosted applications. The diagram below illustrates a top-level overview of the SADK stack with relationship to external Java VM and SQL database.

Figure 1. General architecture of SADK.



Listed below are packages that the SADK library consists of.

- **ple.server** – contains classes and interfaces required to run the PLE server and manage connections to remote Triskan terminals.
- **ple.data** – contains server data related classes: configuration and logging facilities and general database connector handler that encapsulates Java Persistence routines.
- **ple.protocol** – contains classes and interfaces required to decode binary PLE protocol and create protocol frames.
- **ple.plugin** – contains foundation classes and interfaces used to develop user applications hosted by the PLE server.
- **ple.application** – contains examples of PLE applications. Developers may use this package for own applications as well.

6.1. Application Framework

Triskan SADK proposes an event and form based approach to the application design. The most important classes of the application framework are *FormApplication* and *Form*. All PLE applications are created by the PLE server and implement a certain interface (*CommandHandler*). Through this interface, applications react to events submitted by the PLE server. Further, the form abstraction layer is added, and protocol command events are translated to form events and routed to form elements implemented by the *Field* class and its subclasses.

Given below is a class diagram illustrating the internal design of the PLE application framework. For simplicity, class methods and attributes are omitted and only the most important relationships are shown.

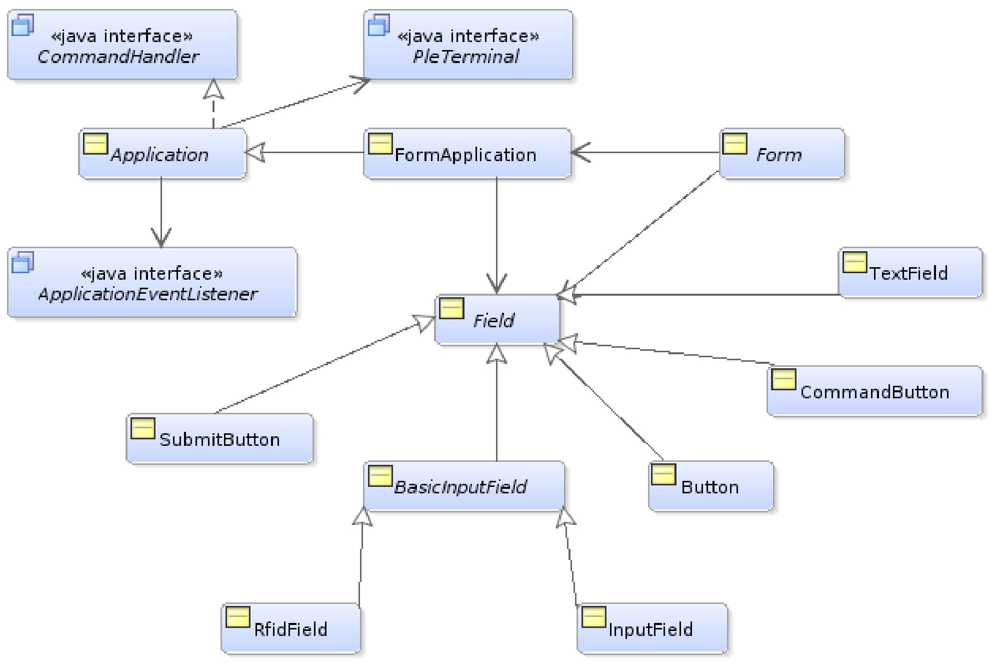


Figure 3. PLE application framework simplified class diagram.

- *Application* class provides the base for implementation of all applications hosted by PLE server which should inherit from this class. Application receives raw protocol command events from the remote terminals relayed by the PLE server via the *CommandHandler* interface.
- *FormApplication* class builds a form abstraction on top of the *Application* basis. *FormApplication* encapsulates a number of user-defined forms that implement both presentation and business logic layers of a PLE application. Also, *FormApplication* transparently calls rendering functions of contained fields to generate appropriate PLE protocol commands and display forms on the screen of a Triskan terminal. It is recommended that all user applications are inherited from this class, as the developer is not bothered with internals of the PLE communication protocol in such case.
- *Form* class implements an input form displayed on the screen of a remote Triskan terminal. The form may contain a number of input fields. The instance of the *Form* class receives events from an associated remote terminal translated to a convenient format by *FormApplication*. All forms of a form-based application should inherit from this class.
- *Field* class implements a base for user input controls of a form displayed on the screen of a remote terminal. Subclasses of the field class implement buttons (*Button*), submit buttons (*SubmitButton*), command buttons *(CommandButton),* text labels (*TextField*), input fields

(*InputField*) and RFID input fields (*RfidField*). Fields implement functions to "render" themselves on the screen by creating a sequence of terminal protocol commands.

6.2. Server Design

Given below is a class diagram illustrating the internal design of the PLE application server. For simplicity, class methods and attributes are omitted and only the most important relationships are shown.

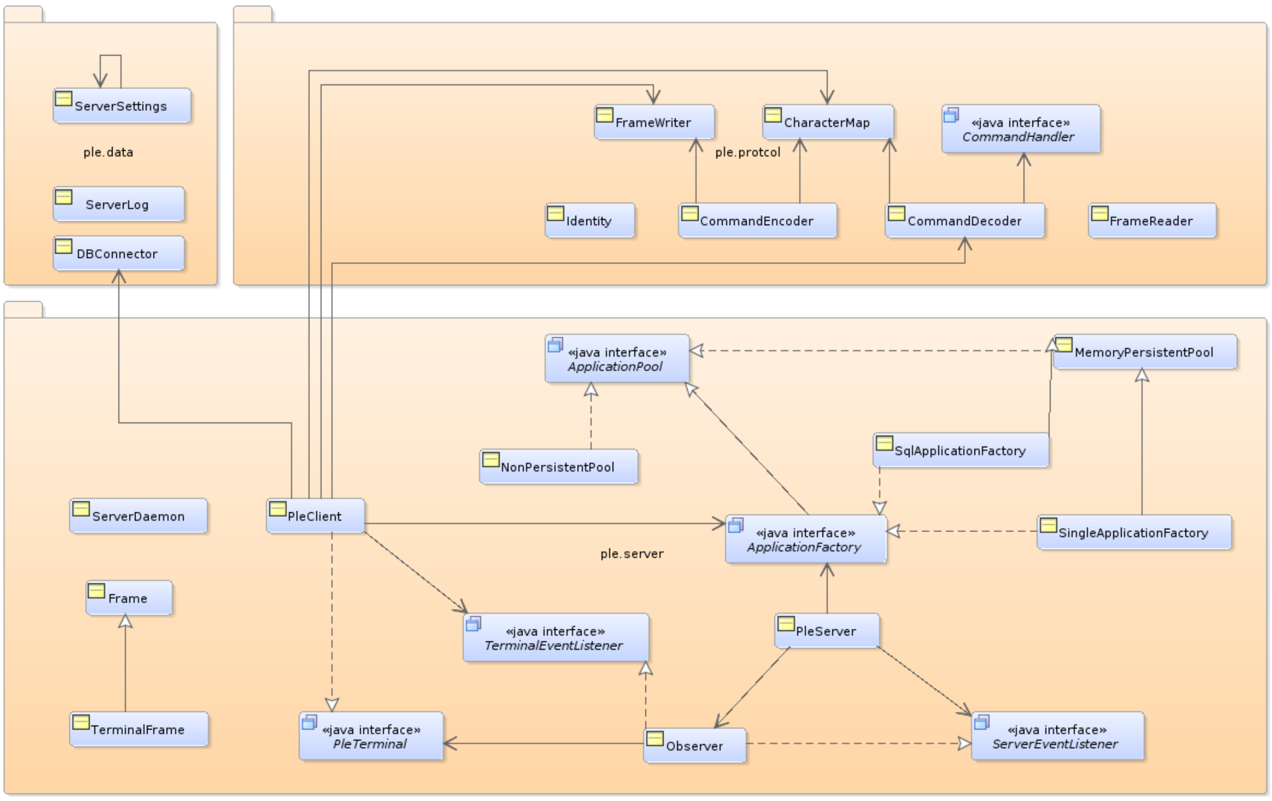


Figure 2. PLE server simplified class diagram.

Normally, a developer does not need to go into details of the PLE server implementation. The server is started by instantiating the *PleServer* class and calling initialize() method. The server has configuration stored to an external file .

Developer may use a stock application factory, such as *SingleApplicationFactory* or *SqlApplicationFactory*. Alternatively, developer may implement a custom application factory and instruct the PLE server to use it instead.

When using multiple applications, developer can use *SqlApplicationFactory* to instruct the server to load the proper application based on the Triskan station ID. To provide access to multiple applications the default loader menu can be developed. It is possible to switch applications run-time by calling the special method.

6.3. Server Configuration

The server reads its configuration from the *pleServer.ini* file located in the current working directory. Samples of configuration files are provided in the distributed package. The format of the configuration file is described below.

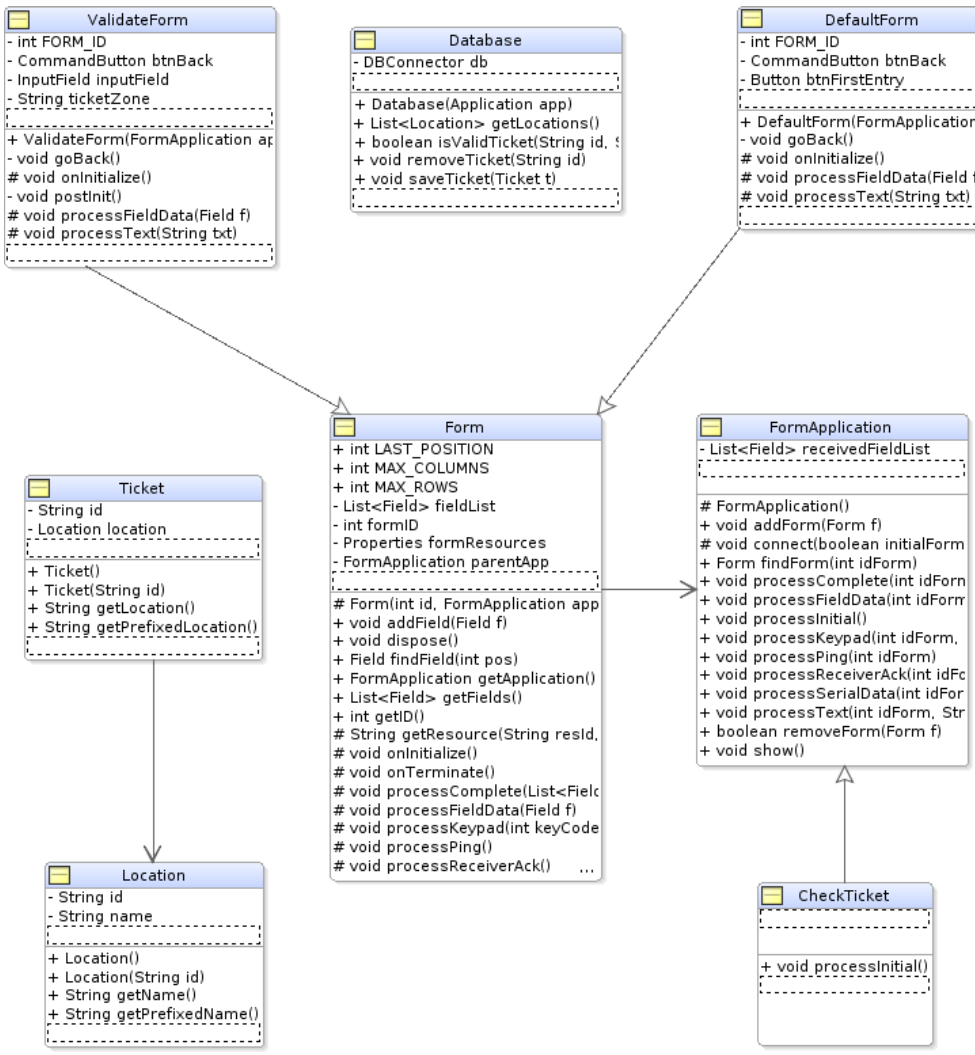
Parameter	*Description*
serverPort	Specifies the TCP port that the PLE server listens on.
idleTimeout	Specifies the idle timeout of a connected PLE terminal, in seconds. The terminal is disconnected if no new data is received within this time interval.
logFile	Server log file name or full path to file.
applicationMode	Specifies server application hosting mode, applies only to stand-alone daemon setup. Supported modes are "single" (use one type of applications for all terminals) and "multi" (use different applications basing on terminal serial number, look up from SQL database).
applicationClass	Contains the application class for a single application mode. Also specifies the "default" fallback application class if the server fails to find a matching application for a connected terminal. When changing applications dynamically, this is also the default application which is loaded if no application name is specified for the application switcher.
persistTimeout	Context persistence timeout of a disconnected PLE terminal, in seconds. NB! This option is only available to applications that use application factories with persistent pools (e.g. MemoryPersistentPool).
sqlUrl	Contains the JDBC URL pointing to a server database for application class lookup. Applicable only to the multi-application mode of operation and when using Java Persistence API.
sqlUser	Contains a user name for authorization to the JDBC database.
sqlPasswd	Contains a password for authorization to the JDBC database.
charMapMid	Sets the character map to use for codes (128...159).
charMapCode	Sets the character map to use for codes (160...255).

- Character map settings do not change the character map in the terminal but provide reasonable defaults for custom character map configurations.

- SQL settings specified here are used both when application mode is set to "multi" as well as for Java Persistence API. Note that when developing database applications the Java Persistence API is configured using the META-INF/persistence.xml file. Nevertheless, the database server name, user name and password can also be obtained from the pleServer.ini file.

7. Developing New Applications

The class diagram below illustrates the structure of a sample ticket validation application (see *ple.application.ticket* package). This example is reviewed here for better understanding of the PLE application design.


ValidateForm
- int FORM_ID
- CommandButton btnBack
- InputField inputField
- String ticketZone
+ ValidateForm(FormApplication ap
- void goBack()
# void onInitialize()
- void postInit()
# void processFieldData(Field f)
# void processText(String txt)
Database
- DBConnector db
+ Database(Application app)
+ List<Location> getLocations()
+ boolean isValidTicket(String id, S
+ void removeTicket(String id)
+ void saveTicket(Ticket t)
DefaultForm
- int FORM_ID
- CommandButton btnBack
- Button btnFirstEntry
+ DefaultForm(FormApplication ap
- void goBack()
# void onInitialize()
# void processFieldData(Field f)
# void processText(String txt)
Ticket
- String id
- Location location
+ Ticket()
+ Ticket(String id)
+ String getLocation()
+ String getPrefixedLocation()
Form
+ int LAST_POSITION
+ int MAX_COLUMNS
+ int MAX_ROWS
- List<Field> fieldList
- int formID
- Properties formResources
- FormApplication parentApp
# Form(int id, FormApplication app
+ void addField(Field f)
+ void dispose()
+ Field findField(int pos)
+ FormApplication getApplication()
+ List<Field> getFields()
+ int getID()
# String getResource(String resId,
# void onInitialize()
# void onTerminate()
# void processComplete(List<Fielc
# void processFieldData(Field f)
# void processKeypad(int keyCode
# void processPing()
# void processReceiverAck() ...
FormApplication
- List<Field> receivedFieldList
# FormApplication()
+ void addForm(Form f)
# void connect(boolean initialForm
+ Form findForm(int idForm)
+ void processComplete(int idForn
+ void processFieldData(int idForm
+ void processInitial()
+ void processKeypad(int idForm,
+ void processPing(int idForm)
+ void processReceiverAck(int idFo
+ void processSerialData(int idFor
+ void processText(int idForm, Str
+ boolean removeForm(Form f)
+ void show()
Location
- String id
- String name
+ Location()
+ Location(String id)
+ String getName()
+ String getPrefixedName()
CheckTicket
+ void processInitial()

Follow the steps below to build a new integrated application hosted by the PLE server.

1. Create a new application class (e.g. *YourApplication*) and derive it from *FormApplication*.

```
import ple.plugin.FormApplication;
```

```
public class YourApplication extends FormApplication {
}
```

2. Create a new initial form class (e.g. *MainForm*) and derive it from *Form*. Create a suitable constructor to instantiate the form with a concrete identifier. The form identifier must be unique for each kind of form.

```
import ple.plugin.Form

public class MainForm extends Form {
        private static final int FORM_ID = 1;

        public MainForm(FormApplication app) {
                super(FORM_ID, app);
        }
}
```

3. Override the *processInitial()* function in *YourApplication* class to create and render *MainForm*.

```
@Override
public void processInitial() {
        Form f = new YourForm(this);
        f.render();
}
```

4. Override the *onInitialize()* function in *MainForm* class to create user interface fields.

```
@Override
private inputField;
protected void onInitialize() {
        addField(new TextField(0, "   -- Welcome --   "));
        addField(new TextField(60, "Enter text:"));
        addField(inputField = new InputField(80, 16));
        addField(new Button(148, "[OK]"));
}
```

5. Override the *processFieldData()* function or *processComplete()* function in *MainForm* class to respond to terminal input events.

```
@Override
protected void processFieldData(Field f) {
        if (f == inputField) {
                // process data here
                String data = f.getData();
        }
}
```

6. Instruct the PLE server to launch your application by defining the application class in the server configuration file.

   ```
   applicationMode=single
   applicationClass=your.package.YourApplication
   ```

If you prefer to run your application stand-alone as a desktop program, follow the steps below to build a GUI front-end for the PLE server.

1. Create a new desktop application class with main() entry point.
2. Develop your GUI front-end class using Java Swing, AWT or any other GUI toolkit.
3. Launch the PLE server by creating an instance of *PleServer* class and calling its *initialize()* function.

   ```
   private PleServer pleServer;

   try {
           pleServer = new PleServer();
           pleServer.initialize();
   } catch (PleServer.ServerException e) {
           // error processing
   }
   ```

7.1 Using Java Persistence API

The Java Persistence API (JPA) can be used to provide database storage for the applications.

1. Create the META-INF/persistence.xml file with the proper configuration:

   ```
   <?xml version="1.0" encoding="UTF-8"?>
   <persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
           http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">

   <persistence-unit name="default" transaction-type="RESOURCE_LOCAL">
           <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
           <class>ple.application.MyPackage.EntityClassName</class>
           <properties>
   ```

```
                <property name="javax.persistence.jdbc.driver"
                      value="com.mysql.jdbc.Driver"/>
                <property name="javax.persistence.jdbc.url"
                      value="jdbc:mysql://localhost:3306/table_name"/>
                <property name="javax.persistence.jdbc.user" value="db_user"/>
                <property name="javax.persistence.jdbc.password"
                      value="db_password"/>
        </properties>
</persistence-unit>
</persistence>
```

2. Note that the persistence unit name must be used below to select proper database connection. The persistence unit with name "default" can be used as a fallback solution when actual unit is not found. The server connection URL, user name and password fields are optional. If the pleServer.ini file contains valid credentials they will be used instead.

3. Create entity classes that provide object-relational mappings. Note that all entity classes that have JPA annotations mus be specified by *<class></class>* attributes:

```
@Entity
@Table(name="badges")
public class Badge implements Serializable {
        @Id
        @Column(name="id", nullable=false)
        private String id;

        @Column(name="checkInDate")
        @Temporal(TemporalType.TIMESTAMP)
        private Date checkInDate;

        @Column(name="checkOutDate")
        @Temporal(TemporalType.TIMESTAMP)
        private Date checkOutDate;

        @Column(name="description")
        private String description;

        public Badge() { }

        public Badge(String id) {
                this.id = id;
        }
}
```

4. Create the database model class that implements database-to-object mapping:

```
import ple.plugin.Application;
import ple.data.DBConnector;

public class Database {
        private DBConnector db;

        public Database(Appliction app) {
        }
}
```

5. Get the database connector instance from the terminal session and store it inside the class. Also select the proper persistence identity from META-INF/persistence.xml to use with the application. Example applications use package name as the identity string. The package name can be obtained using the *this.getClass().getPackage().getName()* call.

```
public class Database {
        private DBConnector db;

        public Database(Appliction app) {
                db = app.getTerminal().getDB();
                db.selectDB(getClass().getPackage().getName(),"database_name");
        }
}
```

The selectDB() method accepts the database name as a last argument. If the database name is specified, the server properties in META-INF/persistence.xml are ignored and database settings from pleServer.ini will be used instead.

6. Create the methods to retrieve the data from database and store them with entity models (e.g Badges) etc.