In Chapter 13, you’ll find out about the wide world of SAPUI5. Via a detailed example, learn how to create an SAPUI5 application that uses SAP Gateway to expose the data model.
What if you could find brand new worlds right here on Earth where anything’s possible? Same planet—different dimension.
I’ve found the gateway!
—Sliders (television show)

13  SAPUI5

I once read an article from the head of SAP usability in which he described an experiment: SAP had created a new application and had some guinea pig users in one room being trained on this new product and some SAP staff in another room watching them via camera and counting how many times the users smiled while using the software. Well, forgive me for being cynical, but if the SAP people were using their fingers to count, then I don’t think they would need both hands. Or even one hand. Maybe the users would smile when they were let out of the room.

Due to this sort of reaction, SAP is determined to shed its image of having the most hideous, user-unfriendly user interface (UI) ever. After many false starts, SAPUI5 could be the breakthrough SAP’s looking for. This chapter talks about the basics of SAPUI5, including its enabler, which is the SAP Gateway add-on for SAP ERP and which comes bundled with your existing license. SAP has finally taken this problem seriously and (in my opinion, at least) nailed it.

Before getting into the details, though, there are two misconceptions to clear up here. The first centers on the crazy fact that there are two utterly different programming languages that both start with the word “Java.” There are historical reasons for this, but Java resembles JavaScript in the same way a duck resembles duct tape. JavaScript is not Java, but it’s still a fully functional programming language. (That’s sort of a pun, because JavaScript is a functional language, whereas Java and ABAP are imperative languages.)

The second misconception is much more common and seems to be shared by developers and management alike. When confronted with the fact that SAPUI5 (or anything else for that matter) is written using JavaScript, people throw up
their hands and say, “Well, we cannot use our ABAP programmers, then, because they just don’t have the right skillset.” Do you know, amazing as it may seem, that there was a time when I didn’t know ABAP? When I was 14, I programmed in BASIC, and then at university Pascal was the go. When I started work (as an accountant, as opposed to as a programmer), on my very first day I discovered how to record macros and then use the programming language in the spreadsheet (which would become Visual Basic) to automate boring tasks. Then, when SAP came along it was time to start writing programs in ABAP, and then the jump from procedural programming to OO programming was rather like learning a whole new language. (Also there are bits of Java all over SAP products now, such as the graphical mappings in SAP PI, and I had to adjust those from time to time.) The point is, nowhere in all of this did I throw up my hands and scream “I only know BASIC! There’s no point in going on in life. I might as well throw myself off the nearest bridge.” You are a programmer. This is more a way of thinking and problem solving than knowledge of the syntax of a particular programming language. If you need to learn something new to solve a problem, learn something new! And, you should learn JavaScript.

The killer argument is that if you can program on both sides of the fence—the ABAP side that produces the data and the JavaScript side that consumes and formats the data—then that puts you in an incredibly small minority and that has to be a good thing for your career. At the end of this chapter is a link to a wonderful article from SAP aimed at introducing ABAP programmers to JavaScript. There are many differences (e.g., variables that change their data types when you give them a different value), but it’s not the end of the world. It may be worth taking a look at that so that the next section (which shows some JavaScript code) makes a bit more sense.

In this chapter, Section 13.1 looks at the technical architecture of how you can expose your SAP applications to appear as SAPUI5 applications in web browsers and mobile devices. It also describes SAP Gateway, which is an SAP add-on that can be used for this purpose, among many others. Section 13.2 discusses the prerequisites for creating an SAPUI5 application: what you will need installed in your SAP system and what you will need installed on the local machine on which you will be developing.

The bulk of the chapter will then describe the process of creating the ever-popular Monster Monitor as an SAPUI5 application. This process has several facets:

1. Use SAP Gateway to expose the data model to the outside world, a process that involves both configuration and ABAP coding (Section 13.3).
2. Create the SAPUI5 part of the application (the view and the controller), which needs to be done on your local machine (the frontend; Section 13.4).
3. At this point, you’ll have a working SAPUI5 application, and you’ll see how easy it is to enhance such an application with all sorts of fancy UI elements (Section 13.5).
4. Next, you’ll look at how you can import your finished SAPUI5 application into the ABAP system so that it can hang out with all its non-UI application friends (Section 13.6).

Finally, the chapter ends by clearing up some confusion between the terms “SAPUI5” and “SAP Fiori,” which seem to mean different things to different people (Section 13.7).

13.1 Architecture

When writing programs within SAP, one of the traditional ways to avoid performance problems was to minimize the amount of data transferred between the database and the server, because the effort involved in moving from one place to another created bottleneck. The SAP HANA platform avoids this entirely by having all the data in memory.

In the same way, the curse of web-based applications (at least from the end user’s point of view) is the roundtrip that occurs every time the user presses a button or enters a piece of data. The information about what the user has done is sent back to the backend system, which then interrogates the database or performs some business logic and sends back instructions to the frontend as to what to display next. This time, the bottleneck is the time spent going to and from the browser to the server and back again, which is why you often spend so much time in web applications looking at a whirling circle (hourglasses are clearly old-fashioned).

Naturally, then, in an application that is going to run in a browser (be it on a desktop or a mobile device) you only want to do a roundtrip when you have a question only the backend can answer (e.g., some information from the database or some decision that has to be made based on complicated business logic, like whether to approve a loan or how many snails to put inside a monster).
In this section, you’ll see how a UI application deals with the frontend processing on the device running the application and the backend processing inside the SAP system.

13.1.1 Frontend: What SAPUI5 Is

I could easily say “SAPUI5 is a JavaScript library,” but would that make any sense to a traditional ABAP developer? Probably not. JavaScript is a programming language—quite different from ABAP, but a programming language nonetheless. A library is simply a collection of reusable classes people have built over time. We do the same thing with reusable Z utility classes and function modules in our ABAP systems, but most SAP programmers don’t group these classes and modules into packages.

The SAPUI5 application is a JavaScript program that runs on the client side—that is, within the web browser or mobile device (i.e., on the machine that’s physically right in front of you, as opposed to the SAP system that could be on the other side of the world).

In model-view-controller (MVC) terms, the view and the controller are on the frontend. The view handles anything explicitly related to what the screen looks like, and when the user does something, the controller decides if the view can handle the request by itself (e.g., shrinking an area of the screen or showing some other static screen) or if it needs to bother the model, which is lurking in the backend SAP system. The idea is that the total amount of roundtrips should therefore be reduced, because you don’t have to go to the backend every single time the user does anything (as is the case with Web Dynpro, for example). This should lead to faster performance times and happier users.

13.1.2 Backend: What SAP Gateway Is

Most articles that talk about SAP Gateway show a diagram with three boxes—the SAP system, SAP Gateway, and the outside world—with arrows pointing between them. I could have included such a diagram, but I’m not sure what it would have proved. You get the point: SAP Gateway lets SAP talk to the outside world.

You are probably thinking to yourself at this point that SAP already has millions of ways to talk to the outside world: IDocs, remote function calls (RFCs), proxy calls to SAP PI, direct invocation of web services, and so on. What’s so special about SAP Gateway?

According to SAP, the problem with all the other integration techniques is that they are proprietary—that is, SAP-specific. In recent years, SAP has made a 180-degree turn away from such bespoke solutions in favor of open-source equivalents, which are much more widely used and not owned by anybody in particular. In the case of SAP Gateway, the protocol it uses to expose SAP data is called OData, as in, “Oh, data, you’re so fine, you’re so fine you blow my mind, oh data.” (Some people say it actually stands for “Open Data Protocol.”)

To go acronym crazy for a second, SAP used to hope that its enterprise system architecture (ESA) vision would catch on, but that was based on SOAP, and in recent times no one likes SOAP anymore. People prefer something called REST, and OData is based on REST. SAP decided that made the ESA thing pretty much dead in the water. If you can’t beat them, join them, and so SAP came up with SAP Gateway to expose SAP data as OData.

You could say that OData is the glue that binds SAP Gateway to SAPUI5. SAPUI5 expects data to be exchanged using such a protocol, and SAP Gateway enables such an exchange to be performed from within an SAP system.

Later on in the chapter, this will all make much more sense, when you see how you actually go about exposing a model class using transactions within SAP.

13.2 Prerequisites

Before you can get up and running with SAPUI5, you have to make sure your system has everything it needs. This section discusses the prerequisites for SAPUI5.

13.2.1 Requirements in SAP

SAP Gateway started life in 2011; it’s a component of the ABAP system, and it can be used in any SAP system based on SAP NetWeaver 7.0 and above. Before 7.4, you had to download and install it explicitly, but in SAP NetWeaver 7.4 and beyond it comes as standard.
In systems earlier than 7.4, you will see some of the components shown in Figure 13.1 when taking the System • STATUS option and looking at the component information—specifically, IW_FND, GW_CORE, and IW_BEP. In a 7.4 system, you will just see SAP_GWFND. Leaving meaningless names aside, what this means is that if you see those components in your system, then you will have some new transaction codes, which you can use to model your SAP data so that SAPUI5 applications and the like can use this data. You will notice that these are the transaction codes used by the 'and the like’; this is not just for SAPUI5. Data exposed in this format can be used by a wide variety of consumers. One example is the new incarnation of Duet, which allows SAP to talk to Microsoft Office applications.

Eagle-eyed readers will notice that in Figure 13.1 there are two components with “UI” in their names: UISAPUI5 and UI_INFRA. These have nothing to do with SAP Gateway (the job of which is to send and receive information to and from the SAP system); they’re concerned with enabling frontend SAPUI5 components to be stored inside the SAP system. This is discussed in Section 13.7.

13.2.2 Requirements on Your Local Machine
In order to use SAPUI5, you have to have Eclipse installed on your local PC (refer back to Chapter 1 to see how to do this). Open Eclipse, navigate to Help • INSTALL NEW SOFTWARE, and enter the following in the box at the top of the screen (Figure 13.2): “https://tools.hana.ondemand.com/[name of latest Eclipse version supported by SAP]/”. Then click the Add button. (You can find the latest Eclipse version supported by SAP by visiting https://tools.hana.ondemand.com.)

It takes a little while to load, but once done you have both sides of the coin installed and are ready to start developing an SAPUI5 application.

13.3 Backend Tasks: Creating the Model Using SAP Gateway
Now, you’ll find out how to take an existing business object that lives inside SAP (a monster, in this example) and enable it to be exposed for use in SAPUI5 applications.

There are two parts to this:
1. The configuration part, in which you define the data structures used in the model that represents the business object and the relationships between them.
2. The coding part, in which you implement methods that will be used to create, retrieve, update, and delete instances of your business object (monster) in the SAP database.

13.3.1 Configuration
In this section, you’ll see how to use SAP Gateway to set things up such that when a URL request is made from a web browser or from a desktop or mobile device for information that resides in our model in SAP that information gets sent back. The steps are as follows:
1. Create a project, which will contain the details of your monster data model.
2. Create so-called entities, which describe the data structure of the monster header and the monster items.
3. Link together these two data structures by means of an association.
4. Create some generated classes based upon that data model and a related service.
5. Make sure that the system configuration is set up correctly so that you can proceed with adding the new service.
6. Add the new service to the list of available services.
7. Perform a basic test to make sure that the service has been added correctly and thus is able to be called by an SAPUI5 application.

Each of these steps is discussed in more detail next.

Creating a Project Using the Service Builder

When you install the SAP Gateway add-on into your SAP system, you get a new transaction code—specifically, Transaction SEGW, which is the Service Builder. Run this transaction now. (Admittedly, it doesn’t look very impressive—just a big gray empty screen with a few icons in the top-left-hand corner.)

On this screen, the top nodes are called projects (which are groupings of one or more business objects). Start by choosing the piece of paper icon in the top-left-hand corner that says Create Project (Figure 13.3).

![Figure 13.3 Creating an SAP Gateway Project](image)

Now, you can get cracking!

Creating Entities for the Monster Header and Monster Items

As you know by now, your monster object has both a header structure and an item structure. You’re going to declare both of these as entities, and then link them together.

As always, there are lots of ways to do this. In this case, right-click the Data Model node and choose Import ➤ DDIC Structure. While doing that, you’ll no doubt notice the other options, most notably the option to use a remote-enabled function module or a BOR object. (A BOR object from Transaction SWO1 is a bona fide business object as far as I’m concerned, and the fact that this shows up in a brand-spanking-new SAP technology means that the BOR is back from the dead, despite SAP trying to unsuccessfully kill it off in 2004.)

Once you’ve created your monster project via the screen shown in Figure 13.3, click the green checkmark; the screen shown in Figure 13.4 appears.

![Figure 13.4 Creating a Monster Entity: 1 of 3](image)

In Figure 13.4, you give your entity a name and pick a data structure. As always, use the BOFF combined header structure. The name here will be used in URLs, so sticking to the convention of lowercase except the first letter is the way to go. Click the green checkmark, and Figure 13.5 appears.
In Figure 13.5, you indicate what fields you want. The 16-digit UIDs are not of much interest to the outside world; they’re only used internally by BOPF. The rest of the data is what you want to show people. Again, click the good old green checkmark, and the screen shown in Figure 13.6 appears.

In Figure 13.6, you nominate what the key field is (this is a structure, so the system has no idea). You can change the text descriptions if you want; they default from the DDIC descriptions, but if you’ve been naughty and used generic data elements in your structure definition, then you can change the name here to something meaningful.

When you click Finish, your empty screen looks a lot better. Next, navigate to the Entity Sets node. An entity set is a grouping of monsters in this case, so an entity set is like a table of class instances, and an entity is one row in that table, a single instance of that class. Tell the system you want to be able to do CRUD operations in regard to monsters and also to search for monsters (Figure 13.7).

Repeat the exact same sequence of steps for your monster item structure; this too will be an entity. As an aside, if you get something wrong, you’ll be confronted by the usual meaningless error messages, as in “Ha, ha, ha, you’ve done something wrong, and I’m not going to tell you what.” Let me take this opportunity again to stress the importance of presenting the users with meaningful information to help them work out what they did wrong and how to correct it. As an example of what not to do with error messages, if you give your entity the same name as one of the fields in the structure, then you get a weird message about children and parents. Clicking the question mark to get the long text on the message is no help, because it’s “self-explanatory,” but luckily Google will help you out.

Calming down for a second, you want to pay attention to the field on the far right side when you click on the Properties node of an entity: This field is called SEMANTICS. If a DDIC field is an email address, a geocoordinate, the start date of a holiday, or another one of many such things (the F4 help will give you a list), then you should state that here. You may wonder why this is important. What it’s all about is that SAP Gateway is communicating with the outside world, and external applications like Microsoft Outlook care very much if something is an
email address or a telephone number, and some applications care about geocoordinates. Because you have no idea what types of things may end up accessing this data, you’re writing in letters of fire 1,000 miles high that something is a telephone number—so that any application that does something with telephone numbers is in no doubt about what field to look at.

**Linking Header and Item via an Association**

Next, you need to say that the items are related to the header, rather like defining a foreign key relationship in a DDIC table. Right-click the Data Model node, and choose Create • Association.

The screen shown in Figure 13.8 appears. Here, choose what to link together (header to items in this case), and fill in the Cardinality fields to say that one header record needs at least one item record but can have many.

![Figure 13.8 Creating an Association: 1 of 3](image)

When you’re done, click the green checkmark. The screen shown in Figure 13.9 appears. Here, link the key fields together (i.e., both Monster_Header and Monster_Item have the same key field, MonsterNumber, and that’s how you can tell which items relate to which header record).

![Figure 13.9 Creating an Association: 2 of 3](image)

At this point, if you expand the node called Service Implementation, then you will see the CRUD operations: Create, Read, Update, and Delete. It makes me want to cry that whoever invented the OData design decided to call the read operations GET, thus turning CRUD into CGUD. There are two read operations: GETENTITY (Read), which returns either a single object or an error, and GETENTITYSET (Query), which returns between zero and multiple objects. However, would it have hurt the OData inventor (I think his name was Seamus O’Data) so much to go with the industry standard? Obviously, yes.

Anyway, you cannot perform any of those operations without some sort of ABAP coding, and for that you need the system to generate some ABAP classes, which will form a service.

**Creating the Service and Classes for Your Monster Project**

Now that you’re done with defining everything, the data is in a form in which you can let the system generate classes and a related service based upon that data. Do this by clicking the red and white crash test dummy icon at the top of the SEGW screen (you can see it at the top of the screen in Figure 13.11, behind the pop-up box) with the hover text Generate Runtime Objects. When you click this button, the pop-up box shown in Figure 13.11 appears.
In Figure 13.11, you’ll see the pop-up box proposing the names of some generated classes that will be created in SE24 world and a service that will be created in SICF world. The classes will do the work within the SAP system, and the service will enable the outside world to access the functionality these classes provide in a controlled manner.

When you click the ever-popular green checkmark, all these things (classes and the service) are generated, and afterwards there are a lot of green lights at the bottom of the screen to tell you that this has occurred.

You’ll find out what to do with the generated classes in Section 13.3.2, which deals with coding. For the time being, you need to do some configuration to get the service into an active state.

**Configuring the Service: General Settings**

The very first time you create an SAP Gateway service, there are some configuration steps that need to be taken, which will remain valid for all subsequent services that you create.

To start, you may have heard it said that talking to yourself is a sure sign of madness. Given that logic, then, your SAP system needs to be mad in order for any SAP Gateway services to work. You’re going to set things up so that your system can indeed talk to itself if it so desires.

First, you need to set up a system alias, which is a human-friendly name for the RFC destination (SAP system) where SAP Gateway lives. In this case, LOCAL means that SAP Gateway is installed and running on the same SAP ERP system where the business data lives, as opposed to some separate hub system.

The system alias is set up via the IMG. You know that IMG menu paths tend to change frequently, so the path in this example won’t look like the one in your system, but if you do a search for “system alias”, then you should find some entries called **Manage SAP System Aliases** and from there be able to create an entry called LOCAL to connect to the current system (Figure 13.12).

In Figure 13.12, on the left of the screen, select some boxes to say that your SAP Gateway system is installed locally and not on a hub (so you don’t need an RFC destination to reach that hub). On the right of the screen, enter the details of the system (system ID and client number) where the SAP Gateway system resides, which is the same as the system in which the SAP data lives.

You’re halfway there; there’s one more configuration setting needed to enable your SAP system to talk to itself. In the IMG, navigate to SAP NetWeaver • GATEWAY SERVICE ENABLEMENT • BACKEND ODATA CHANNEL • CONNECTION SETTINGS TO SAP NetWeaver Gateway • SAP NetWeaver Gateway Settings. (What a nice, simple path to follow.)
Add an entry exactly the same as the one shown in Figure 13.13. The word GATEWAYLOCAL appears in the menu of project nodes (such as your monster node) in Transaction SEGW under SERVICE MAINTENANCE, and later you’ll see how this can be used to register the service.

This is going to sound silly, but you also have to ensure that SAP Gateway is actually active. In order to do this, go into a slightly different version of Transaction SPRO, called SIMGH. There, you have to choose the IMG Structure Project Gateway 1.0. Then, choose GATEWAY • ODATA CHANNEL • CONFIGURATION • ACTIVATE or DEACTIVATE SAP NetWeaver GATEWAY. If it’s not active, then you can switch it on; if it’s already active, then wonderful—nothing more to do here.

Adding the New Service

Earlier in the chapter, when you processed the screen shown in Figure 13.11 you sort of created the new SICF service. However, to really bring it out of the womb you need to go to another transaction that is delivered with the SAP Gateway add-on—namely, /IWFND/MAINT_SERVICE.

In Figure 13.14, you can see the initial list of SAP-supplied SAP Gateway services. Before you add your own, you need to make sure that the green light in the bottom-left-hand corner is on; otherwise, you’re not going anywhere. If by any chance that light is not green, then right-click on the ICF node icon and choose CONFIGURE SICF. That will take you into the relevant section of SICF, where you need to activate everything in sight—but especially the ADDRESSTYPE node that lives under ODATA and everything above it in the tree.

Back on the ACTIVATE AND MAINTAIN SERVICES screen (Figure 13.14), click the ADD SERVICE button at the top; the screen shown in Figure 13.15 appears.

When this screen first appears, the list of services at the bottom will be blank. To find your service, do an (F4) dropdown search on the SYSTEM ALIAS field; you’ll see the entry for LOCAL, which you created just now, so select that.

Enter “Z*” in the TECHNICAL SERVICE NAME field, because you’re only interested in custom services as opposed to standard, SAP-delivered ones. Those are all the fields you need to fill out. Click the GET SERVICES button at the top of the screen, and any created services that aren’t yet active are listed, as shown at the bottom of Figure 13.15. Select your monster service by clicking the selection box to the left of the service and pressing the ADD SELECTED SERVICES button. The screen shown in Figure 13.16 appears.

In the screen shown in Figure 13.16, accept all the default values. You can rename the service name and model name if you want, but there really isn’t any benefit to doing so. In this example, the PACKAGE ASSIGNMENT field is set as a local object. When creating a service for productive use, you would nominate an actual package.
After you've added your service, you can go back to the /IWFND/MAINT_SERVICE screen (Figure 13.14), and your monster service will have appeared somewhere on the list of services. You can search for this newly added service by using the Filter option—for example, to look for services with "monster" in the description. Once you've selected your monster service, you're ready to test it.

Tip: Faster Way to Add a New Service

You've seen how to add a new service by going to Transaction /IWFND/MAINT_SERVICE, pressing the Add button, searching for a service to add, adding that service, and then searching for the newly created service. You'll have noticed that there was a whole lot of searching going on.

A much easier way to add a new service (possible due to the configuration settings you made earlier to link destination GATEWAYLOCAL to the current SAP system) is to jump straight from Transaction SEGW into the correct part of Transaction /IWFND/MAINT_SERVICE, bypassing all the searching.

In the SEGW screen (Figure 13.17), the Service Maintenance node of the monster project is expanded. On the right-hand side of the screen, you can see a Register button.

Clicking the Register button in the SEGW screen in Figure 13.17 is a shortcut through the steps you followed before. First, the screen shown in Figure 13.16 appears, allowing you to add the service (without having to search for it), and then you end up on the /IWFND/MAINT_SERVICE screen (Figure 13.14), with the newly created monster service selected (again, without having to search for it).

Testing the New Service

You've got to the stage where you've added a new service to the list of available services. You're on the /IWFND/MAINT_SERVICE screen, with the monster service selected. When you added the service in the last section, a service node in Transaction SICF was created and activated. You can see the details of this service node by making sure your monster service is selected in the top half of the /IWFND/MAINT_SERVICE screen, making sure the line with the green light and the word ODATA next to it is selected in the bottom half of the screen, and then choosing ICF Node • Configure SICF. The screen shown in Figure 13.18 appears. This screen shows the entry in Transaction SICF for your monster node.
Transaction SICF controls how incoming URLs are dealt with in SAP. A URL starts with the name of your SAP system, then the number 8000 (to be more precise, 8000 is the default value, but your SAP Basis people may have configured this to be something else), and then a bunch of parameters (e.g., SAP System/8000/this/that/the other).

The tree structure in an SICF node corresponds to a URL that someone (human or program) enters into a web browser, which then contacts your SAP system and uses the URL to find the SICF node and to call one or more ABAP handling classes, which are defined in the node definition. In this case, the system has defined a generic class called /IWFND/CL_SODATA_HTTP_HANDLER to respond to the incoming URL and forward the request to your SAP Gateway monster entity.

You could have a class that decides it cannot deal with the incoming request, but rather than raising an error just decides to let the next class in the chain have a go to see if it has better luck. This whole approach of using service nodes frees you from the task of having to create half a billion lines of code each time you want to expose a business object to the outside world by using the SICF incoming URL framework.

**Note**

For the examples in this chapter, make sure the services /sap/bc/ui5_ui5 and /sap/opu are active. You do this by going into Transaction SICF and expanding the menu path for each of these services (e.g., SAP/L54263 BC) until you find the service listed. If the service description is in bold, then it’s already active. If the description is not in bold, then the service is dormant. Right-click it, choose the Activate option, and then say Yes to the Are you sure? prompt.

Moreover, not only can you see the details of your new service from within SICF, but you can also test it. At the bottom of the /IWFND/MAINT_SERVICE screen is a Call Browser button. Click it, and up will pop a web browser with the URL of the SICF node filled in (Figure 13.19).

**Browser Support**

Not all browsers seem to like OData and SAPUI5 very much, though they all promise to support SAPUI5 in their latest versions. If you get into trouble in your version of Internet Explorer (for example), then try Google Chrome.

**Figure 13.19 Calling the Browser Test Tool**

This is all wonderful: An external request can see that you have various monsters that can be searched for, each with some items. However, at the moment, if the external application (which is going to be an SAPUI5 application in this case) wants to look up a particular monster, or all the monsters, or a subset of them, then it’s right out of luck—because your SAP Gateway monster entity doesn’t have any ABAP code in it yet, just a bunch of empty generated classes.

### 13.3.2 Coding

In the last section, you not only created and activated a service to expose your monster model data to the outside world, you also generated some ABAP classes that are to be used to manipulate your monster data within the SAP system. These classes won’t do anything much until you add some code to them. There are several steps to this process:

1. First, you have to understand the structure of an SAP Gateway service implementation, such as the one you will be creating for your monster data model. This will help you understand what code we need to add and where.
2. Code the data retrieval method `GET_ENTITY_SET`.
3. Test the data retrieval method `GET_ENTITY_SET`. This is going to be different than the testing you’ve been used to, because you have to test this from a web browser.
4. Code `GET_ENTITY_SET` for associated entries, where the entity set is the list of items associated with a monster entity.
5. Code error handling (i.e., how to pass error messages back to the calling application) so that they can appear in the SAPUI5 display.

Each of these steps is discussed in more detail next.
Understanding the Structure of an SAP Gateway Service Implementation

To get an idea of what an SAP Gateway service implementation looks like, go back to your SEGW transaction, select your Z_MONSTER service, and expand the Service Implementation and Monster Set nodes. You’ll see the CRUD operations (Figure 13.20): Each one is represented by a generated ABAP method, into which you have to put some code.

Figure 13.20 CRUD Classes that Need Some Code

In fact, there are four classes that have been generated by the system for each project. The first two are an abstract class called (in this case) ZCL_Z_MONSTER_DPC (where DPC stands for “data provider class”) and a class that inherits from this called ZCL_Z_MONSTER_DPC_EXT, which you can and will change. In the abstract class, there are methods for create, read, and so on, but all they do is raise an exception, so you need to redefine them all. The second two classes have almost the same names as the first two, except they have MPC instead of DPC (MPC stands for “model provider class”); take a look in the code for ZCL_ZMONSTER_MPC, and you will see generated code that sets up the data model based upon the configuration settings you’ve made.

To reiterate, the data provider class stores transactional methods, which you need to redefine, and the model provider class stores the data model, which you do not need to change, because it was created based on the settings you made setting up the data model.

Note that there are two types of entity (monster and monster item), but both entities live in the same project and are thus manipulated by the same service. Thus, the CRUD methods for each type of entity are all in the same DPC class.

Coding the CRUD methods is not actually going to be a major problem, because you already have the monster model class, which does all of these operations. The only task is to translate the interface of the generated OData classes to the equivalent signatures of the corresponding monster model methods.

If you are coding CRUD methods and you do not have an existing model class, then (a) shame on you and (b) you will have to code the database reads and updates directly in the respective methods—which is not very reusable.

Take a look at a few examples of how to code some of these transactional methods, starting with the query operation GET_ENTITY_SET.

Coding the GET_ENTITY_SET Query Method

Start by selecting the GET_ENTITY_SET (Query) node, right-clicking, and choosing Go To ABAP Workbench. That takes you to SE24 (after a message telling you that you haven’t yet created the thing that you’re trying to create, which you already knew). You’ll see the ZCL_Z_MONSTER_DPC_EXT class, with half a million methods.

The first thing you want to do is give is to give this class the attribute MO_MONSTER_MODEL, which references your monster_model class, and a CONSTRUCTOR, in which you set up a monster model as soon as an instance of this is created by the SAP Gateway framework. The code for this is shown in Listing 13.1.

```
METHOD constructor.
  super->constructor( ).
  CREATE OBJECT mo_monster_model.
ENDMETHOD.
```

Listing 13.1 Embedding the monster_model Class into the Data Provider Class

Now, you can move along to the inherited methods; start with the MONSTERSET_GET_ENTITYSET method. As is normal in SE24, select the method and click the REDEFINE icon. Then, you can add some code.

Delete the generated code (all that does is call an empty method that raises an exception), and then proceed to add your own coding. You will have noticed by now in this book the benefits of abstracting our monster logic into its own model class: for every new framework you look at, you can reuse the same class as before.
The purpose of the \texttt{GET\_ENTITY\_SET} method is to pass back a list of all the entities (monsters) that have been requested by an incoming URL. As can be seen in Listing 13.2, first you start dealing with instructions that have been passed in to the method (from the URL), such as the sort order (like the \texttt{SORTCAT} table in the ALV), which is here stored in \texttt{LT\_SORTORDER}, then the selection options, and finally some more obscure things, like paging.

In regard to the selection options, they’re listed in the URL in the form of a string, but the handler class in the SICF service node transforms this into the form of a table of selections (\texttt{IT\_FILTER\_SELECT\_OPTIONS}), which is passed into the \texttt{GET\_ENTITY\_SET} method. Translate that selection table into the format that your monster model class data retrieval method likes, and then delegate getting the actual data to the monster model method \texttt{RETRIEVE\_HEADERS\_BY\_ATTRIBUTE}.

One you have the data, check if the incoming URL requested the data to be sorted in a particular order. If so, then build up a dynamic table to sort the retrieved data. Now check to see if the incoming URL specified that you only bring back certain lines (e.g., the first 10 rows of the table).

Next, an export table is filled up with all the entities (monster headers) that you want to send back to the caller. Right at the end is some code to enable testing using Eclipse.

\begin{verbatim}
METHOD monsterset_get_entityset.
  * Extract any instructions as to how to sort the result list from
  * the incoming request
  DATA: lt_techorder TYPE /iwbep/t_mgw_tech_order,
        lt_sortorder TYPE abap_sortorder_tab.
  IF io_tech_request_context IS BOUND.
    lt_techorder = io_tech_request_context->get_orderby( ).
  ENDIF.
  * See if we have any selection criteria passed in
  * We adapt the DATA structure to the BOPF selection structure
  DATA: ls_filter_select_options

  LIKE LINE OF it_filter_select_options,
  ls_select_options
  LIKE LINE OF ls_filter_select_options:select_options.
  DATA: lt_selections TYPE /bobf/t_frw_query_selparam,
        lt_result TYPE ztt_monster_header.
  FIELD-SYMBOLS: <ls_selections> LIKE LINE OF lt_selections.
  IF it_filter_select_options[] IS NOT INITIAL.
    LOOP AT it_filter_select_options INTO ls_filter_select_options.
      APPEND INITIAL LINE TO lt_selections
      ASSIGNING <ls_selections>.
      <ls_selections>-attribute_name =
      ls_filter_select_options:attribute_name.
      <ls_selections>-option = ls_filter_select_options:option.
      <ls_selections>-sign = ls_filter_select_options:sign.
      <ls_selections>-low = ls_filter_select_options:low.
      <ls_selections>-high = ls_filter_select_options:high.
    ENDLOOP.
    mo_monster_model->retrieve_headers_by_attribute( EXPORTING
      lt_selections = lt_selections
      IMPORTING
      et_monster_headers = lt_result ).
  ELSE.
    * No selection criteria have been passed in
    * Set selection criteria so that all records are returned
    APPEND INITIAL LINE TO lt_selections
    ASSIGNING <ls_selections>.
    <ls_selections>-attribute_name = 'MONSTER_NUMBER'.
    <ls_selections>-option = 'GT'.
    <ls_selections>-sign = 'I'.
    <ls_selections>-low = '0000000001'.
    ENDF.
    mo_monster_model->retrieve_headers_by_attribute( EXPORTING
      lt_selections = lt_selections
      IMPORTING
      et_monster_headers = lt_result ).
  ENDIF.
  * Now we build a dynamic table which we will then use to
  * sort the result a la SORTCAT in the ALV
  FIELD-SYMBOLS: <ls_tech_order> LIKE LINE OF lt_techorder,
  <ls_sort_order> LIKE LINE OF lt_sortorder.
  LOOP AT lt_techorder ASSIGNING <ls_tech_order>.
    APPEND INITIAL LINE TO lt_sortorder
    ASSIGNING <ls_sort_order>.
    <ls_sort_order>-name = <ls_tech_order>-property.
    IF <ls_tech_order>-order = 'desc'.
      <ls_sort_order>-descending = abap_true.
  ENDLOOP.
\end{verbatim}
SORT lt_result BY (lt_sortorder).

* Query the incoming URL to see if we have to start from a specific point, and how many rows to display
DATA: ld_start_row TYPE sy-tabix,
     ld_end_row TYPE sy-tabix.

* The URL may contain text like "$skip=5"
 IF is_paging-skip IS NOT INITIAL.
   ld_start_row = is_paging-skip + 1.
 ELSE.
   ld_start_row = 1.
 ENDIF.

* The URL may contain text like "$top=10"
 IF is_paging-top IS NOT INITIAL.
   ld_end_row = is_paging-skip + is_paging-top.
 ELSE.
   ld_end_row = lines( lt_result ).
 ENDIF.

* Export the final result
FIELD-SYMBOLS : <ls_entity_set> LIKE LINE OF et_entityset,
                 <ls_result> LIKE LINE OF lt_result.
LOOP AT lt_result FROM ld_start_row TO ld_end_row
  ASSIGNING <ls_result>.
  APPEND INITIAL LINE TO et_entityset
  ASSIGNING <ls_entity_set>.
  MOVE-CORRESPONDING <ls_result> TO <ls_entity_set>.
ENDLOOP.

* In development only change security settings to allow local testing
DATA: ls TYPE ihttpnvp.
  ls-name = "Access-Control-Allow-Origin".
  ls-value = '*'.
 /iwbep/if_mgw_conv_srv_runtime-set_header( ls_header = ls ).
ENDMETHOD.

Note
Say that you really did want the second and third rows only for some reason (e.g., you have a table of days of the week, and you want Tuesday and Thursday). In such a case, it would be crazy to do what was discussed previously (i.e., get all the records and then throw most of them away). That goes against everything you’ve ever been taught, most specifically to minimize the amount of data transferred between the database and the application server.

If you really wanted to do that, then you would need to have an extra field in the database table (day number, for example) and to use that field as a selection.
OData queries have been described as very database-centric. Because they are open source and not SAP-specific, they assume you have the full range of SQL options available, which is not the case in ABAP. For example, the $TOP/$SKIP options have a direct counterpart in full SQL, which is the LIMIT addition to the SQL query, as in `SELECT * FROM monster table LIMIT 1 OFFSET 2`.

**SAP HANA**

In an SAP HANA environment, if you didn’t have such a SQL option available to use in native ABAP, then you’d push down the $TOP/$SKIP logic into a procedure in the database layer so that the discarding of unwanted rows would happen before sending the result back to the application server, thus complying with the golden rule. This is discussed in more detail in Chapter 15.

**Testing the GET_ENTITY_SET Method**

For now, go back into the /IWFND/MAINT_SERVICE transaction, and look for your monster service. Once you’ve found it, once again make sure that your cursor is on the ODATA line at the bottom of the screen, and click Call Browser. Naturally, the exact same screen as before pops up, showing the data structure you’ve defined.

Now, take things to the next stage: Change the URL so that the suffix (i.e., the bit after `/Z_MONSTER_SRV/`) is $metadata. Now, you can see a more detailed description of the data structures (Figure 13.21).

That may not seem very exciting, but the $metadata bit is important, because it’s how the OData service implements its self-describing characteristic. For example, you can see that MonsterNumber is the key: it’s a string field of a maximum length of 10 characters, and it cannot be null. This is how a UI control can dynamically figure out how big to make a field on the screen and so on.

Now, take things one step further and change the URL once again so that it ends in `/odata/sap/Z_MONSTER_SRV/MonsterSet?sap-ds-debug=true`. You have to be very careful here: URL entries are case sensitive, and you need to make sure you type in the name of the entity set exactly as it appears in the $metadata list.

As can be seen in Figure 13.22, the results are more impressive this time. You’ll see a big list of all the monsters, which is proof that the ABAP code has been called. You could (and you really should) put an external breakpoint in the method to see for yourself what’s happening.

**Coding GET_ENTITY_SET for Associated Entities**

Now you want to be able to see the items for an individual monster, so it’s time to redefine another method in your entity set. Listing 13.3 reads `IT_KEY_TAB`,
which gets filled in based on what you pass into the URL, and uses this to display the item details.

METHOD monster_itemset_get_entityset.
  * Local Variables
  DATA : ls_key_tab LIKE LINE OF it_key_tab,
         ld_monster_number TYPE zde_monster_number,
         lt_result TYPE ztt_monster_items.

  READ TABLE it_key_tab INTO ls_key_tab
  WITH KEY name = 'MONSTER_NUMBER'.

  CHECK sy-subrc = 0.

  ld_monster_number = ls_key_tab-value.

  TRY.
    mo_monster_model->retrieve_monster_record(
      EXPORTING
      id_monster_number = ld_monster_number
      IMPORTING
      et_monster_items = lt_result ).

    et_entityset[] = lt_result[].

  CATCH zcx_monster_exceptions.
    RETURN.
  ENDTRY.

  ENDMETHOD.

Listing 13.3 Coding the Method to Get All the Monster Items

As before, put an external breakpoint in the method so that you can see what's happening, and call up the browser as described previously. This time, change the URL so that it ends in Z_MONSTER_SRV/MonsterSet('0000000007')/Monster_ItemSet?sap-ds-debug=true, and off you go again. You have to make sure that you get the uppercase letters right and the underscores in the correct positions.

The result (Figure 13.23) looks very similar to what you saw before. The method you coded brought the data back correctly from the database, and you can see all the items for a monster with one call.

Coding Error Handling

Now you need to redefine all the other methods of both entities (create, delete, and so forth), but because all that involves is passing the call on to the equivalent method in your monster model class, there's no need to go through each one here in any detail.

This example puts a small piece of code in the delete method for you to see how to pass back exceptions to the calling application. It also sets things up so that trying to delete a monster always fails, so that you can see an error message in your SAPUI5 application later on. The code for this is shown in Listing 13.4.

METHOD monsterset_delete_entity.
  * Local Variables
  DATA: lo_message_container
        TYPE REF TO /iwbep/if_message_container.

  lo_message_container =
    /iwbep/if_mgw_conv_srv_runtime-get_message_container( ).

  lo_message_container->add_message_text_only(
    EXPORTING
    iv_msg_type = `/iwbep/if_message_container=>gcs_message_type-error`
    iv_msg_text = `This monster does not want to be deleted` ).

  RAISE EXCEPTION TYPE `/iwbep/cx_mgw_busi_exception`

Figure 13.23 Monster with One Head
If you look at the definition of exception class /IWBEP/CX_MGW_BUSI_EXCEPTION, you’ll see that there are a whole bunch of parameters that you can pass into the constructor, as well as some standard error messages that you could use, like resource not found. Exception classes were discussed back in Chapter 7.

Once you’ve redefined the CRUD methods for your monster entity, you’ve finished the first half of the process. Any sort of web application can now send a URL that points to your SAP system, and the SICF framework will invoke the entity set methods you’ve written to read or change the data in SAP.

### 13.4 Frontend Tasks: Creating the View and Controller Using SAPUI5

Section 13.2.2 talked about installing the add-ons in Eclipse that are needed for SAPUI5 development. You’re now at the stage in which you’ll create the frontend components that will run on the device where the SAPUI5 application is displayed. These components will be a view and a controller, and they live together inside an SAPUI5 application project.

To start this process, open Eclipse, and navigate to New • Other • SAPUI5 APPLICATION DEVELOPMENT • APPLICATION PROJECT. You may be puzzled by the labels on the radio buttons in Figure 13.24 in the Library box. In some versions of Eclipse, the labels tell you what they mean, but in others (like the one shown), you get secret codes. To put you out of your misery, the first one (sap.ui.commons) means that your target device is a desktop and the second one (sap.m) means that your target device is a mobile. Choose the second (mobile) option so that later on you can see how the screen rearranges itself as the display area gets smaller.

The Options area in Figure 13.24 contains a Create an Initial View checkbox. Leave it selected, because you really need a view in order for the user to be able to see anything. As a result, another pop-up box appears asking questions about the view (this time, asking for the name of the view, e.g., Monster Overview), and then it asks you what language you want to program the view logic in. In fact, although you want the controller to be written in JavaScript, you want the view to be in XML. Choose the XML option, and you’ll see that an XML file will be created for the view and a JavaScript file will be created for the controller.

After you click Next, the box shown in Figure 13.25 appears to tell you what’s going to happen. In ABAP, you’re used to programs being saved straight in the database. In other languages, the code is usually stored in a file structure on your local device. Here, you can see where the files are going to be stored. Then, you’ll see the target device, which is a mobile device in this case (not that you would know from the secret code).

Next, there are a whole bunch of libraries that are going to be added to your SAPUI5 program. As mentioned earlier, you can think of this as a bunch of INCLUDE statements giving you access to assorted functions, the way you add an INCLUDE statement at the start of function modules or classes that are going to add data to a workflow container so that you can use assorted macros.
Finally, you’ll see that a view file and a controller file are going to be created. Your model lives in SAP and uses ABAP, and the view and controller will run on the local device. Hopefully, you can now see how valuable you can become if you know more than one language.

You might be asked if you want to open the project in a Java EE perspective, which is a puzzling question for the average ABAPer. The correct answer is yes. (Once you understand that the word “perspective” in that sentence refers to the way the screen will look, things make a lot more sense.)

If you remember the discussion of Eclipse back in Chapter 1, the screen in Figure 13.26 shouldn’t be too scary. There are tabs for the view and controller, in the same way that you would have two classes in ABAP to represent these constructs. You can see that SAP has already generated some code (e.g., to tell the view what its controller is), and there is explanatory text at the start of each method definition.

![Figure 13.26 Eclipse Screen for Coding View and Controller](image)

Just to recap, the model lives inside your SAP system, written in ABAP. The controller is on the local device, written in JavaScript. The view is also on the local device, but is written in XML, and as you’ll see it will contain no code at all—just instructions on how to look pretty. Having the model, view, and controller in totally different languages enforces the separation of concerns by putting 1,000-mile high walls between them. That’s a more effective way of doing this than saying to the programmer “Do not do such and such in the view; it’s naughty.”

Now the time has come to start coding, and this is where the fact that you’re in a new language will start to make some people shake and sweat. If you’re starting to worry about your job, remember that all the complicated business logic is still being programmed inside the ABAP system, and the SAPUI5 coding is solely concerned with making the display look nice (and SAP provides functions to make this as easy as possible) and handling user interaction.

This section first talks about the view (the part of the application in charge of making sure the display looks nice) and how it’s organized technically. It then moves on to the controller, which uses JavaScript code to handle communication between the view and the model living in your SAP backend system. Finally, it explains how to test your newly created SAPUI5 application in order to make sure everything is working as expected.
There are three different files that will be created in your SAPUI5 application: an HTML file, an XML file, and another XML file, called a fragment.

The HTML file is the web page that every SAPUI5 application needs in order to run in a web browser on a laptop or on a mobile device. The XML file is the “proper” view; the example presented only has one, but complex applications can have many. The XML fragment is sort of like an INCLUDE file that can be reused by other view files. Fragments are used in the majority of the SAPUI5 examples presented in Section 13.5.

HTML File

An SAPUI5 application is, for all intents and purposes, a web page—so you need to define an HTML file that calls up your view when you access it via a URL. This was created for you automatically when you created a new SAPUI5 application; so in that new application follow the path Monster Demo • Web Content • Index.html. The code to create this file is shown in Listing 13.5.

```html
<!DOCTYPE HTML>
<html>
<head>
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<meta http-equiv='Content-Type' content='text/html;charset=UTF-8'/>
<script src="resources/sap-ui-core.js"
    id="sap-ui-bootstrap"
data-sap-ui-libs="sap.m"
data-sap-ui-xx-bindingSyntax="complex"
data-sap-ui-theme="sap_bluecrystal">
</script>
<!-- only load the mobile lib "sap.m" and the "sap_bluecrystal" theme -->
<script>
sap.ui.localResources("monster");
var app = new sap.m.App({initialPage:"idmyView1"]);
var page = sap.ui.view({id:"idmyView1",
    viewName:"monster.myView",
type:sap.ui.core.mvc.ViewType.XML});
app.addPage(page);
app.placeAt("content");
</script>
</head>
<body class="sapUiBody" role="application">
<br />
</body>
</html>
</head>
<body class="sapUiBody" role="application">
<br />
</body>
</html>
Listing 13.5 Monster HTML File

In Listing 13.5, there are several sections, including two <script> sections. Somewhat like ON LOAD OF PROGRAM, the first <script> section loads the core JavaScript library and any other libraries (groups of functions) you need. Sometimes, you have to add these manually; if you do, it will be mentioned in the generated comments. In this example, the comments tell you to only load one library and a theme, and the lines to do this are generated for you.

Then there is another <script> section, which performs the following tasks:

- States the folder where the view and controller live (in this case, monster).
- Creates a new mobile application with the variable name APP and specifies the name of the view that is going to appear first. This is somewhat like when you create a DYNPRO application in ABAP and say what screen comes first.
- Another variable, PAGE, says to use the monster.myView XML file.
- Links the two variables (they’re objects, really) together by adding the initial page to the application.
- Indicates that the application is going to live in the content section of the web page.
- Defines the body of the web page, and inside that declares that the content of the web screen is indeed called content.

XML File

In Listing 13.5, you defined the view with the witty name of MyView, and that is indeed what the view file is called (its full name is MyView.view.xml). At this point, you’re going to write some XML code to create a screen with a search box at the top and a list of monster header records in the main area of the screen, which can be filtered based on what the user puts in the search box. Once you’re done with this process, the finished screen will look like Figure 13.27.
In order to end up with Figure 13.27, you have to do some coding. Every view needs a controller, so the very first thing that happens in your view file is to say what the name of the controller file is. The controller is going to have all the code; all you’re doing in the view is laying out the screen and sometimes saying that a screen element will respond to being clicked. The controller will worry about what to do if a user does click such a field or button.

In Listing 13.6, you set the header level information for the view—not only the name of the controller but also some SAPUI5 libraries that it will need to run, such as SAP.M. This is similar to having INCLUDE files in old-fashioned ABAP programs full of reusable subroutines.

```
<core:View xmlns:core="sap.ui.core"
    xmlns:mvc="sap.ui.core.mvc"
    xmlns="sap.m"
    controllerName="monster.myView"
    xmlns:html="http://www.w3.org/1999/xhtml">

Listing 13.6 A View Defining the Name of its Controller
```

All the code in Listing 13.6 was automatically generated for you at the time the view file was created (which was when you clicked the Finish button in Figure 13.25), so now you can move on to defining your own code to define what your monster screen looks like. All good screens need a title at the top; for this one, the title will be Search for Monsters. Also, put a button next to the title, for no good reason other than to show that you can put buttons in the title bar, just like the icons at the top of an ALV screen. Listing 13.7 shows the code for adding the title, and Figure 13.28 shows the result.

```
<Page title="Search for Monsters" class="marginBoxContent">
    <headerContent>
        <Button icon="sap-icon://action" />
    </headerContent>

Listing 13.7 Adding a Title
```

The first line on your screen (header) is the title. The next line down on that screen (subheader) is going to be a box in which you can enter the name of a monster. When you type “F”, for example, the list will filter to show all the monsters with names starting with “F”. This is a standard SAPUI5 screen element: a box with the word SEARCH on the left and a magnifying glass on the right. The good thing about using standard elements is that it gives your applications a uniform look.

Listing 13.8 presents the code to add the search field. You’re going to place the search field inside the tool bar. You could also add a button next to the search field inside the tool bar if you wanted, just like in standard SAP screens that have two lines of icons at the top of the screen. The cumulative result is shown in Figure 13.29.

```
<subHeader>
    <Toolbar>
        <SearchField width="200px" liveChange="onSearch" />
    </Toolbar>
</subHeader>

Listing 13.8 Adding a Search Field to the Screen
```

You’re done with the header section of the screen (headerContent in Listing 13.7 above). Now, you’ve come to the body (content) of the screen as shown in Listing 13.9, which will be somewhat like an ALV grid—only better looking. You can have as many different tables of data as you want on your screen, but in this case there’s just the one. Time to start determining what this table is going to look like.
In Listing 13.9, you indicate that you’re starting a new table called MonstersTable and then say where you’re getting the data from. The view says that it wants to display the data that lives in a URL that ends with /MonsterSet. The controller takes care of working out the rest of the URL (i.e., where the model data comes from, which you know is inside your SAP system, but the view doesn’t need to know any of this).

Next, there’s another toolbar. You had one in the header of your screen, and now you have one for your table. This is just the same as when you have a GUI screen with several ALV grids. The main screen has an application toolbar and each grid has a title and its own row of icons at the top as well. In your SAPUI5 screen, set the title to MONSTERS. You could add some buttons here as well, if you so desired. Some are all in favor of filling the top of the screen with icons (e.g., Microsoft), and some say doing so confuses the user. In the SAPUI5 world, we’re free to try and strike a happy medium.

```xml
<content>
<Table id="idMonstersTable" inset="false" items="{path: '/MonsterSet'}">
  <headerToolbar>
    <Toolbar>
      <Label text="Monsters"/>
    </Toolbar>
  </headerToolbar>
</Table>
</content>
```

Listing 13.9 Creating a Table and Giving It a Title

Next, in Listing 13.10 you will define the column headings, which naturally you can name anything you want. In traditional ALV reports, you’re used to the column heading defaulting from the texts of the data elements that are used in the columns, but half the time you end up manually changing these anyway. In SAPUI5 world, you explicitly define the texts for all the column headings.

```xml
<columns>
  <Column>
    <Text text="Name and Number" />
  </Column>
  <Column>
    <Text text="Color" />
  </Column>
  <Column>
    <Text text="Sanity" />
  </Column>
  <Column>
    <Text text="Hat Size" />
  </Column>
</columns>
```

Listing 13.10 Defining the Column Titles

Now, it’s time to decide what values should be displayed inside each column. In Listing 13.11, the first thing that you do is to say that the rows in this column can be navigated; that is, they do something when someone clicks the row. In this case, they tell the controller to invoke its OnMonsterSelected function.

Now, move on to the contents of each cell in the rows in the table. Start with the identifier: the title is the monster name, which people can search for, and the text description is the number. Usually, in SAP reports this is the other way around, but in real life nobody knows monster numbers—but they do know names (i.e., no one ever screams, “Oh no! Here comes Number 12678!”, but they do scream, “Oh no! Here comes the Creeping Terror!”).

The next three lines say what fields from your monster structure are going to be queried for their values to go into the row cells. There are two things to note here. First, you’ll notice that each field from the structure (e.g., color), which is going to be replaced by a value, is enclosed in curly brackets ({$}). This is just the same as the way string processing works in ABAP now: If you enclose a variable in a string, then in that same set of brackets it gets replaced with the variable’s value at runtime. Second, you may notice that there are fields from the monster header structure that have been used throughout this book: Color is a field stored in the database, but SanityDescription and HatSizeDescription are transient fields that get determined at runtime by looking up the text description of a related database field. Normally, you’d have to do some coding to get those text values. Here, you don’t, because your model uses BOPF, which fills out such fields automatically without your having to worry your pretty little head about anything. The cumulative result is shown in Figure 13.30.

```xml
<items>
  <ColumnListItem type="Navigation"
      press="onMonsterSelected">
    <cells>
      <ObjectIdentifier title="{Name}" text="Monster Id {MonsterNumber}" class="sapMTableContentMargin" />
      <Text text="{Color}" />
      <Text text="{SanityDescription}" />
      <Text text="{HatSizeDescription}" />
    </cells>
  </ColumnListItem>
</items>
```

Listing 13.11 Defining the Data for Your Table
Listing 13.11 Defining the Cell Values for Each Column

<table>
<thead>
<tr>
<th>Monsters</th>
<th>Name and Number</th>
<th>Color</th>
<th>Status</th>
<th>Hat Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>HUBERT 1</td>
<td>GREEN</td>
<td>BOOKENDS</td>
<td>NORMAL HAT</td>
<td></td>
</tr>
<tr>
<td>HUBERT 2</td>
<td>GREEN</td>
<td>BOOKENDS</td>
<td>NORMAL HAT</td>
<td></td>
</tr>
</tbody>
</table>

Figure 13.30 SAPUI5 Application: Including Monster Table

At the bottom of the page is the footer. Once again, you can put a toolbar with buttons here, and this is just what you do in Listing 13.12. As in the header, the buttons don't do anything. They're just there to prove that they can be there. The buttons are shown in Figure 13.31.

Listing 13.12 Defining the Footer: Adding Some Buttons

Figure 13.31 Action Buttons in Footer of SAPUI5 Application

You’re finished with the main view!

Fragment XML File

Maybe that last exclamation mark was a tad strong; you should know that few, if any, applications have only one screen. In this case, what you want when a user clicks on a monster is for a dialog box to open with the full monster header details at the top and a table of monster items at the bottom. This is technically possible to achieve in the SAP GUI, though when someone asks you to code this, you probably waggle your finger at the requester and say “Tut, tut, now, now, come, come,” or words to that effect, and try to talk them out of the request because there’s a fair bit of work involved. Happily, in the SAPUI5 world this is fairly painless—which is just as well, because this is the sort of extra requirement that pops up ten minutes before go-live on a regular basis.

To optimize the performance of your application, load only the main view to start off with. Any other views you need are loaded on demand; to do this, you define the other views (pop-up boxes, in this case) as fragments. Here, you’re going to display your monster detail information in some style using a fragment, a process that could be described as a fragment style monster. Ultimately, you’ll see that the controller function `onMonsterSelected` calls the fragment that you’re about to define—a fragment the purpose of which is to pop up a big dialog box full of monster details (Figure 13.32).

Figure 13.32 Monster Item Pop-Up Box
Start your fragment definition in Listing 13.13 by indicating that this will be a dialog box, giving this box a title, and saying that it takes up 60% of the screen. The result is shown in Figure 13.33.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<core:FragmentDefinition xmlns="sap.m"
xmlns:core="sap.ui.core" xmlns:f="sap.ui.layout.form">
<Dialog title="Monster Details" contentWidth="60%">
<content>
Listing 13.13 Defining Header Details of the Fragment (Pop-Up Box)

Figure 13.33 Dialog Box Title

In Listing 13.14, you’ll set up the header details of the selected monsters. At the start of the code, declare that those details are going to be laid out in SimpleForm, one of the many UI elements available for you to use. Then, say how many columns the form is going to have (one in this case), and so on; this is just like passing values into a function module or method.

You’ll recognize from Listing 13.11 the technique for declaring what the name of the fields are going to be. For each such field, add the name of an SAP DDIC field with {} brackets around it to store the value that is going to be dynamically determined at runtime. The runtime system knows what monster header you’re talking about, because you’ve just navigated from that particular monster, and the code in the onMonsterSelected handler passes in the correct monster. The result is shown in Figure 13.34.

```xml
<Table inset="false" items="{path: 'Monster_ItemSet'}">
<headerToolbar>
<Toolbar>
<Label text="Monster Items" />
</Toolbar>
</headerToolbar>
<columns>
<Column width="100px">
<Text text="Item" />
</Column>
<Column width="100px">
<Text text="Part Category" />
</Column>
<Column width="100px">
<Text text="Part Category Description" />
</Column>
<Column width="100px">
<Text text="Part Qty" />
</Column>
</columns>
</Table>
Listing 13.14 Defining Layout and Field Texts for the Pop-Up Box

Figure 13.34 Dialog Box Header Details

Next, in Listing 13.15 you’ll define a table that shows all the items for the monster you’re drilling into: how many arms, heads, tails, and so on. The way to define such a table is exactly the same as the way you did this for the table of header details in Listing 13.10 and Listing 13.11. First, declare a path so that the controller can ask the model to get some monster item data. Then, there’s a title at the top. Finally, define the column names and their widths, and say what values need to go in the columns at runtime. The result is shown in Figure 13.35.

```xml
<Table inset="false" items="{path: 'Monster_ItemSet'}">
<headerToolbar>
<Toolbar>
<Label text="Monster Items" />
</Toolbar>
</headerToolbar>
<columns>
<Column width="100px">
<Text text="Item" />
</Column>
<Column width="100px">
<Text text="Part Category" />
</Column>
<Column width="100px">
<Text text="Part Category Description" />
</Column>
<Column width="100px">
<Text text="Part Qty" />
</Column>
</columns>
</Table>
Listing 13.15 Defining the Table for the Pop-Up Box

Figure 13.35 Dialog Box Table Details
Figure 13.35 Dialog Box with Item Table

In Listing 13.16, you add some buttons at the bottom. Two of them ask the controller to call a function when the button is pressed; the Rampage button is just there for fun. The completed dialog box, along with the buttons at the bottom, is shown in Figure 13.36.

Listing 13.15 Defining the Table of Monster Items for a Given Monster

### Monster Details

<table>
<thead>
<tr>
<th>Monster Name</th>
<th>Color</th>
<th>Strength</th>
<th>Number of Heads</th>
</tr>
</thead>
<tbody>
<tr>
<td>HUBERT (3)</td>
<td>GREEN</td>
<td>REALLY STRONG</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Item</th>
<th>Part Category</th>
<th>Part Category Description</th>
<th>Part Qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>000010</td>
<td>HD</td>
<td>Head</td>
<td>1</td>
</tr>
<tr>
<td>000011</td>
<td>AR</td>
<td>Arm</td>
<td>1</td>
</tr>
<tr>
<td>000012</td>
<td>AR</td>
<td>Arm</td>
<td>1</td>
</tr>
<tr>
<td>000013</td>
<td>LG</td>
<td>Leg</td>
<td>1</td>
</tr>
<tr>
<td>000014</td>
<td>LG</td>
<td>Leg</td>
<td>1</td>
</tr>
<tr>
<td>000015</td>
<td>TA</td>
<td>Tail</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 13.36 Dialog Box with Action Buttons at the Bottom

You may be wondering how to know what value to pass into the ICON field. In normal SAP, you use Transaction ICON to call up a list of all the pretty icons you can use in your screens. The equivalent in SAPUI5 is to go to the following URL, where you will get such a list:

https://openui5.hana.ondemand.com/test-resources/sap/m/demokit/icon-explorer/index.html

Listing 13.16 Adding Action Buttons at the Bottom of the Pop-Up Box

You may be wondering how to know what value to pass into the ICON field. In normal SAP, you use Transaction ICON to call up a list of all the pretty icons you can use in your screens. The equivalent in SAPUI5 is to go to the following URL, where you will get such a list:

https://openui5.hana.ondemand.com/test-resources/sap/m/demokit/icon-explorer/index.html
In this case, I wanted a picture that looked like a monster for my Rampage button and the nearest I could find was the Physical Activity button. Rampaging through the village terrorizing the peasants is indeed a physical activity.

13.4.2 Controller

That does it for the view files; now it's time to code the controller. As mentioned earlier, the views are in XML and the controller is in JavaScript, so the file ends in “JS”. In a functional language, which JS is, functions are first-class citizens, so don’t be surprised to see lots of functions in the following coding strutting all about the place and looking pleased with themselves.

Code in the controller consists of a series of function definitions—and unlike ABAP methods, here this means the definition (signature) and implementation (coding) all together. The steps for coding the controller are as follows:

1. Create the initialization method that’s called when the controller is first created.
2. Define the search box, which filters the list of monsters.
3. Define the dialog box that appears when a user clicks on a particular monster.
4. Define the functions to respond to any buttons the user presses once that dialog box is open.
5. Define a function you need in the controller to help with testing.

Initializing the Controller

The controller brokers communication between the model and the view, so when you start coding the controller the first thing you’re going to do is say what model you’re interested in, and then move on to handling events that the view tells you have happened.

Listing 13.17 starts off by saying “Hello, I am a controller and my name is…” so that the view can refer to it. Then, you swiftly move on to saying what model you want. When the SAPUI5 application queries your SAP system to get the SAP Gateway service, it needs a URL that can get to the correct place. What you see in the declaration of variable OMODEL (starting with /SAP/OPU/ODATA) is the path to the SICF node where your SAP Gateway service is located. The bit at the start of the URL is determined by another function at the end of the controller file (this will be covered in due course).

jQuery.sap.require("sap.m.MessageBox");
sap.ui.controller("monster.myView", {  
/**  
 * Called when a controller is instantiated and its View controls (if available) are already created.  
 * Can be used to modify the View before it is displayed, to bind event handlers and do other one-time initialization.  
 * @memberOf monster.myView  
 */  
onInit: function() {  
   var oModel = new sap.ui.model.odata.ODataModel(this.getUrl("/sap/opu/odata/sap/Z_MONSTER_SRV"), true);  
   this.getView().setModel(oModel);  
  },
Listing 13.17 Controller Declaring the Model It Uses

You will see a whole bucket load of generated code in the controller file, including a lot of commented-out function skeletons with comments above them saying what those functions are used for. For example, you can see the comments that describe the onInit function, which is pretty much the same as its namesake in Web Dynpro. (Indeed, the example in Chapter 12 involved the creation of an instance of the monster model when the WDDOINIT method was called, which was then used throughout the application; this is what’s happening here as well.)

There’s no need to go through all those generated comments here, because they’re all self-explanatory, so skip straight to the functions you define that are specific to your monster application.

Defining the Search Box Function

Back in your view, you defined a search field at the top of the screen and said liveChange = onSearch. The right-hand side of the assignment (onSearch) is the name of the function you are about to define, and the left-hand side of the assignment (liveChange) indicates that the function is called each time the user changes even one letter in the search box (as opposed to waiting for the user to click the SEARCH icon).

To recap what the search box does, if you type in “F”, then all monsters with names containing “F” pop up, and all others vanish from the list. If you type in “D”, then all monsters with D in their name pop up, and so forth. This would be quite a complicated bit of coding normally. Here, you have to do virtually nothing.
You code the `onSearch` function in Listing 13.18, and as you can see, you’re not passing very much into the boilerplate code. You say what field you want to filter on (monster name, in this case). In the second part, you say what table you’re talking about from the view (the table of monsters). The other two input parameters are generic values.

In the first part of Listing 13.18, the variable `SQUERY` is filled with the contents of the search box. If there is anything in the box, then the filter object is filled with a list of all monsters whose names contain the value in the search box. In the second part, the result list is retrieved into variable `OTABLE`, and that table is filtered by the filter object created in the first half of the listing.

```javascript
onSearch: function(oEvent) {
  // add filter for search
  var aFilters = [];
  var sQuery = oEvent.getSource().getValue();
  if (sQuery && sQuery.length > 0) {
    var filter = new sap.ui.model.Filter("Name",
      sap.ui.model.FilterOperator.Contains, sQuery);
    aFilters.push(filter);
  }
  // update list binding
  var oTable = this.getView().byId("idMonstersTable");
  var binding = oTable.getBinding("items");
  binding.filter(aFilters, "Application");
},
```

Listing 13.18 Coding the `onSearch` Event Handler

**Defining the Dialog Box Function**

Next, you may recall that in the view you set the column list items to be of type navigation, which would call `onMonsterSelected` when a user selected a given row. This is what controllers are all about; this function is called from one view and then goes about linking another view (the dialog box) with the model. Now, it’s time to code that function using the code in Listing 13.19.

```javascript
onMonsterSelected: function(oEvent) {
  // set context of dialog box
  if (this._monsterDetailDialog === undefined) {
    this._monsterDetailDialog = new sap.ui.xmlfragment("monster.MonsterDetail",
      this);
    this._monsterDetailDialog.setModel(this.getView().getModel());
  }
  this._monsterDetailDialog.open();
  this._monsterDetailDialog.bindElement({
    path: oEvent.getSource().getBindingContextPath(),
    parameters: {
      expand: 'Monster_ItemSet'
    }
  });
},
```

Listing 13.19 Coding the Function `onMonsterSelected`

**Defining the Buttons in the Dialog Box**

Next, you have to deal with the buttons at the bottom of your dialog box. Only two buttons do anything; the definitions of those buttons point to `onMonsterDetailDialogCancel` and `onMonsterDelete`, respectively. Naturally, these are functions you have to code as well.

```javascript
onMonsterDetailDialogCancel: function() {
  console.log('onMonsterDetailDialogCancel');
},
```

Listing 13.20 shows the `onMonsterDetailDialogCancel` function, and as you can see, the name is almost as long as the code inside the function. You just call a standard SAPUI5 method to close the dialog box.
onMonsterDetailDialogCancel: function (oEvent) {
  this._monsterDetailDialog.close();
},

Listing 13.20 Coding the onMonsterDetailDialogCancel Function

That wasn’t very exciting; time to move on to deleting a monster. The code to do this in Listing 13.21 is a bit meatier. First, once again you get the model and then the source to see which monster you’re talking about. The source is the line in the monster list you selected, the context of that line is what refers to the specific monster, and the path of that context is what you need to pass to the model. Hence, oEvent.getSource().getBindingContext().getPath().

When the remove method of the model is invoked, the application goes to the SAP system and calls the MONSTERSET_DELETE_ENTITY method of your SAP Gateway service. This will raise an exception if something goes wrong, and it certainly will in this case. This example codes the delete method in such a way that an exception is always raised, and thus the error function will be called.

onMonsterDelete: function (oEvent) {
  var oModel = this.getView().getModel();
  var oPath = oEvent.getSource().getBindingContext().getPath();
  oModel.remove(oPath, {
    success: this._monsterDeleteSuccess,
    error: this._monsterDeleteError
  });
},

_monsterDeleteSuccess: function (oEvent) {
  this.onMonsterDetailDialogCancel();
  sap.m.MessageToast.show('Deleted', {
    duration: 5000,
    my: 'center center',
    at: 'center center'
  });
},

_monsterDeleteError: function (oEvent) {
  var oResponse = JSON.parse(oEvent.response.body);
  if (oResponse.error.message) {
    sap.m.MessageBox.alert(oResponse.error.message.value);
  } else {
    sap.m.MessageBox.alert('An Unknown Error Detected');
  }
},

Listing 13.21 Coding Functions Involved in Deleting a Monster

As it turns out, you cannot test the DELETE function from Eclipse. The online SAP help gives all sorts of excuses as to why this is. Regardless, you have to wait until you have imported the application into your SAP system, which is discussed in Section 13.6.

Had the deletion succeded, then the success message would have appeared as a toast message, which means the notification pops up (hence the name) on the screen for a given number of seconds and then fades away, like the notification you get when a new email arrives. In case of an error, the alert is like an information message; that is, it stays on your screen until you click on the pop-up box to acknowledge that you’ve seen the message.

Defining a Function to Help with Testing

The final function in your controller (Listing 13.22) has nothing to do with user interaction. It concerns making sure the start of the URL is correct when trying to contact the SAP system.

getUrl : function (sUrl) {
  if (sUrl == '')
    return sUrl;
  if (window.location.hostname == 'localhost') {
    return 'proxy' + sUrl;
  } else {
    return sUrl;
  }
},

Listing 13.22 Coding the getUrl Function to Aid with Testing

To reiterate, the first 95% of the controller code was doing the normal job of a controller, but Listing 13.22 related solely to helping you test the application, which leads in nicely to the next section.

13.4.3 Testing Your Application

Right at the start of the controller, when the model was being retrieved, the getUrl function was called, which specified the last half of the URL (the path to the SICF node) but was looking for the first half of the URL, which says what SAP system you’re pointing to.
When testing earlier in the chapter, you just typed in a big long URL with the address of your SAP host at the start and the path to your SAP Gateway service at the end. When testing the application, the Eclipse environment is going to put the words “local host” at the start of the URL, so you need to substitute this for the actual host name of your SAP system.

How do you do that? What is this “proxy” of which I speak?

You need to navigate to Webcontent • Web-Inf • Web.xml, and make a quick change to that file. When looking at this, you’ll see that everything has been generated automatically. The bit you need to change is shown in Listing 13.23.

```xml
<!-- UI5 proxy servlet -->
<!========================================================== -->
<servlet>
  <servlet-name>SimpleProxyServlet</servlet-name>
  <servlet-class>com.sap.ui5.proxy.SimpleProxyServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>SimpleProxyServlet</servlet-name>
  <url-pattern>/proxy/*</url-pattern>
</servlet-mapping>
<context-param>
  <param-name>com.sap.ui5.proxy.REMOTE_LOCATION</param-name>
  <param-value>http://my_sap_host_name:8000</param-value>
</context-param>
Listing 13.23 Defining the Actual SAP System for a Local Host

You can find the host name of a given SAP system by following the menu path System • Status and looking at the Database Data section. You can ask your SAP Basis person if the number at the end is 8000 or something else.

Why go through all this rigmarole, you may ask? Because to start off with you want to test this on your local device, but eventually your application will become productive, and the local host prefix will not apply. More importantly, if you look on the Internet you will see dozens of people trying to test SAPUI5 applications, getting screens with no data, and wondering why. It’s because they’re not using the simple proxy servlet just described and are running into the so-called CORS problem. (Recall that you have to make a change in the code to get the server side to work; for a refresher, refer back to Listing 13.2.)

Due to all the effort you just made to define proxy settings and the like, you can now select Monster Monitor in the top-left-hand corner of the Eclipse screen, and choose Run As • Web App Preview. A new tab appears, and after a few seconds your application screen appears (Figure 13.37).

![Figure 13.37 Testing an SAPUI5 Application in Eclipse](image)

If you click on the blue ball in the top-right-hand corner, then your default Internet browser will open (as mentioned earlier, if your default browser is Internet Explorer, then this may not work), and you can see the screen in a full web page.

### 13.5 Adding Elements with OpenUI5

You may have heard that the Eskimos have three million words for snow. In the same way, there are often many different terms referring to the same (or almost the same) thing in the SAP sphere, and this confuses people to no end. For example, you may have heard the terms “UI Development Toolkit for HTML5,” “SAPUI5,” and “OpenUI5” and wondered what the difference was.

The first two are one and the same. SAPUI5 is a JavaScript library of assorted functions that relate to UI elements. OpenUI5 is the open-source version of SAPUI5. The only difference is that SAPUI5 contains functions that solely relate to SAP-specific concepts and wouldn’t make sense in any other environment. The functions in OpenUI5 can be used by anybody at all, even if they never have and would not touch SAP with a 10-foot bargepole. As might be imagined, OpenUI5 contains a very large percentage of the SAPUI5 functions.
OpenUI5 and Open Source

OpenUI5 is often described as “open source,” but many people would say that’s not quite true. The product is open source in that it’s free to download and you can use it any way you want without having to pay anybody a license, but that’s only half of the open-source story.

With real open-source products, not only can you view the source code, you are actively encouraged to fix bugs and make enhancements and to send those changes to the project so that the whole world can benefit. SAP is not quite willing to go that far yet, but SAP claims that it will make this a proper open-source product eventually.

Thanks to OpenUI5, there is an absolutely stunning online resource to help you start adding all sorts of exotic features in no time at all, available at http://sap.github.io/openui5/index.html (Figure 13.38). This will really come in handy in real life—because, let’s face it, your users are going to want the moon on a stick when it comes to the look and feel of their applications.

If you click the Download link shown in Figure 13.38, you can get the latest version of OpenUI5 (a new version comes out every few weeks), and more importantly for the purpose at hand there’s a link to the developer guide. (When you open the developer guide, it seems like all references to “open” UI5 are gone, and the logo and description at the top of the screen refer to SAPUI5, as seen in Figure 13.39. However, you’re really still in the OpenUI5 developer guide.)

Adding Elements with OpenUI5

At this point, to explore the vast array of options open to you, navigate to the Explored tab. As you can see in Figure 13.40, on the left is a huge list of UI elements. If you select one, then the right-hand side of the screen displays a description of what this entity does. If you click Samples, you can see a working example, and then you can click the Code button to see the code needed to make this work. Usually, you’ll see some code samples: a view, a fragment, and a controller. By now, you hopefully understand which one does what.

Just as an example, if you want to add a dropdown menu at the top of your monster application in the form of an action sheet (as shown in Figure 13.40), it’s the work of minutes to copy the sample code from the website into your controller and view files, add a fragment file, and then make some monster-related changes.
First, create a new XML file called ActionSheet.fragment.xml, as shown in Listing 13.24.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<core:FragmentDefinition
xmlns="sap.m"
xmlns:core="sap.ui.core">
<ActionSheet
title="Choose Monster Action"
showCancelButton="true"
placement="Bottom">
<buttons>
<Button text="Create" icon="sap-icon://lab" />
<Button text="Give Award" icon="sap-icon://competitor" />
<Button text="Remove Head" icon="sap-icon://wrench" />
<Button text="Award Degree" icon="sap-icon://study-leave" />
<Button text="Inject" icon="sap-icon://syringe" />
<Button text="Other" />
</buttons>
</ActionSheet>
</core:FragmentDefinition>
```

Listing 13.24 ActionSheet.fragment.xml

This code defines a list of push buttons with icons that will appear in a context menu when the user clicks the Action button. Now, change the code near the top of your view. You had a button that did nothing; change it to one that invokes an action from the controller (Listing 13.25).

```javascript
<headerContent>
<Button
text="Open Action Sheet"
press="handleOpen" />
</headerContent>
```

Listing 13.25 Coding a Button that Triggers an Action when You Press It

Finally, add a function to your controller to respond to the user pressing the button and to call up your fragment of possible monster actions (Listing 13.26).

```javascript
handleOpen : function (oEvent) {
  var oButton = oEvent.getSource();
  // create action sheet only once
  if (!this._actionSheet) {
    this._actionSheet = sap.ui.xmlfragment("monster.ActionSheet.", this);
    this.getView().addDependent(this._actionSheet);
  }
  this._actionSheet.openBy(oButton);
}
```

Listing 13.26 Call Up Fragment

This is just cutting and pasting for the main part, though naturally you always have to change something, and the more experienced you get, the more you will change things.

The assorted buttons that appear when you call up the action sheet do nothing. In real life, the fragment would have press definitions like the one in Listing 13.25 to call functions in the controller based on what icon the user clicked (e.g., for the Remove Head button, you’d say press = handle head removed and then code a function in the controller called handle head removed).

Adding that code and changing a few things took maybe six or seven minutes, and when you test your application again you can see that a dropdown menu has appeared at the top, as shown in Figure 13.41.

![Figure 13.41 Adding a Dropdown Menu](image)

As you can imagine, this makes adding features to an application a breeze, but I would urge you to actually look at any code you copy and try to figure out what’s actually going on. If you do, it will make troubleshooting so much easier, and before you know it you’ll have your head round XML and JavaScript syntax.
13.6 Importing SAPUI5 Applications to SAP ERP

It’s possible that throughout this chapter you’ve been horrified by the idea of developing in Eclipse, where the files are stored on your local computer, as opposed to in the SAP ABAP repository, which is where everything you normally develop lives.

The good news is that it is in fact possible to move your new SAPUI5 application inside the SAP system, where it can join the repository object club. Technically, how this is done is that the SAPUI5 runtime leverages the BSP framework to store the SAPUI5 artifacts on the ABAP server. BSP applications look a bit like HTML pages with other programming languages inside them, and by usage of the mime repository and a special ICF handler, the SAPUI5 application can be stored in such a way that it looks like a BSP application, even if it isn’t one really—like those insects that disguise themselves as bees to get inside the hive and steal the honey.

This section explains how to store SAPUI5 applications in SAP ERP for releases both prior to and after 7.31. The section ends by discussing how to test SAPUI5 applications from within SAP ERP.

13.6.1 Storing the Application in Releases Lower than 7.31

For releases below 7.31, you cannot directly move an SAPUI5 application definition into the SAP ERP system. However, the good news is that SAP Note 1793771 provides a workaround, giving you an ABAP report to upload and download files. After about SP 3 for release 7.02, this report is available as standard in the SAP ERP system and is called /UI5/UI5_REPOSITORY_LOAD (the irony here is that this standard SAP report supplied to upload SAPUI5 applications contains an awful lot of WRITE statements).

To get started with the process of moving your SAPUI5 application definition into the ABAP system, select your main monster node in Eclipse, right-click on it, choose EXPORT, select FILE SYSTEM, and the whole application structure will be saved in a folder on your local machine. Next, go to SE38 and run program /UI5/UI5_REPOSITORY_LOAD; there is no transaction code.

The screen shown in Figure 13.42 appears. Be careful here; the length of the input box might make you think you can have a really long name, but only 12 characters are allowed. If you put a long name in, then you will only get an error at the end of the process.

Click Execute, and the screen shown in Figure 13.43 appears. As you can see, this shows you all the things the system is about to do. Click CLICK HERE TO UPLOAD.
Another box appears, asking for a name for the BSP application and a development package (or you could choose $TMP$ for a local object). After this point, your application will be snugly inside the SAP ERP system.

### 13.6.2 Storing the Application in Releases 7.31 and Above

Release 7.31 SP 4 and above can import SAPUI5 application definitions directly from Eclipse into the SAP ERP system. In Eclipse, select your main Monster Monitor node, right-click, and choose Team > Share Project. The reason for this naming is that the component that moves things from Eclipse to SAP is called a team provider, on the grounds that if something is in Eclipse it’s on your local machine and only available to you, but if it’s inside the SAP system, it’s available to anyone on the programming team.

The first pop-up is shown in Figure 13.44. There are a few options; choose SAPUI5 ABAP Repository. In the next box, you will be asked to browse to the SAP logon pad on your local machine to get the details of the SAP system you want to connect to. All the logon pad details will be copied over, and then you will be asked for the client and user name and password (unless single sign-on is active). If all goes well, then the connection is established. Remember, if your backend SAP system is not on a high enough level, then you will get error messages at this point about not supporting Eclipse tools or not supporting team providers.

If there are no errors, then the next box will talk about creating a new BSP application and what package you want to put it in. If this is not a local object, then you will be asked for a transport request, as might be expected. After that, the dialog is finished, and you would expect that you’ve moved your application into ABAP—but no, you haven’t. All you’ve done is to express your strong desire to do so.

You have to select the main Monster Monitor node in Eclipse once more, and this time choose Team > Submit. Then, you’ll see a box with a big list of files to move into the SAP system. You may wonder why this is a two-step process. The whole idea of Eclipse is strange to an ABAP developer; you can have several developers changing the same program on their local machines and then they both try to
import their new versions into the SAP system. You just don’t have that problem inside ABAP, but tools like Eclipse have to live with this on a daily basis and as such provide a means to deal with such conflicts.

13.6.3 Testing the SAPUI5 Application from within SAP ERP

Once the SAPUI5 application has arrived inside your SAP system disguised as a BSP, you can test it in the exact same way you would test any other BSP. You can use SE80 and choose BSP Application and then hunt for the name of the application you just created, and there it will be (Figure 13.45).

The code in Listing 13.27 will not work if you put it in the index file inside Eclipse; conversely, the index file inside SAP ERP will not work without the code being changed. This is because both environments need to get their resources in slightly different ways when starting the application.

After transferring the data to the SAP ERP system, you’ll also see the name of your application in Transaction SICF, under the path sap/bc/ui5_ui5. This means that if you type “<myhost>/<myport>/sap/bc/ui5_ui5/monster_monitor” into a web browser (note that Internet Explorer doesn’t seem to work on versions 9 and below; Chrome works just fine), then the application will start.

You could also select the top node of your BSP application in SE80 and click the Test button. In Figure 13.46, you can see the result when selecting a monster and clicking the Delete button. The error message is shown, which did not work when you tested this inside Eclipse.

Clearly, the benefit of having the SAPUI5 definition within the ABAP repository is that you can have this in the same transport as the pure ABAP objects that make up your SAP Gateway service. This way the model, view, and controller can all be transported from development to test together, even if they’re in different programming languages.
13.7 SAPUI5 vs. SAP Fiori

If you’ve heard about SAPUI5, it’s likely that you’ve also heard about SAP Fiori. Although it seems like the terms are sometimes used interchangeably, SAPUI5 is not the same thing as SAP Fiori. SAP Fiori refers to a set of applications that were built using SAPUI5 and that can be customized using SAPUI5. (Kind of like how all the standard SAP transactions are written in ABAP, and you can write your own Z ones.)

The idea here is the diametric opposite of how SAP has approached building UIs in the past. Traditional SAP screens are viewed using your desktop, and they are crawling with fields and tabs and context menus, enough to satisfy every company in every industry in every country of the world. Each individual company would only use a few of those fields and tabs, however. The applications using those screens have an enormous amount of functionality (i.e., you can do about 34 different things from one screen).

An SAP Fiori screen, in contrast, is designed to be viewed on a smartphone or some such and performs one task only, showing you the absolute bare minimum of screens and buttons; one screen and one button is the ideal (e.g., to approve a leave request).

SAP initially came out with about 25 SAP Fiori apps to handle common SAP tasks, and this was going to change the way people thought about using SAP. Sadly, initially SAP was going to charge for this in addition to the normal license fee, when their competitors were giving out UI improvements for free as part of standard maintenance. In early 2014, common sense prevailed, and since then SAP Fiori and another product called Screen Personas have been bundled into the annual maintenance fee with no extra charge. More SAP Fiori applications are being released over time in “waves.”

Synactive must have been very sad when SAP came out with its own equivalent, called SAP Screen Personas, which enables you to do the same sort of thing but to a much greater extent and in a web browser. You have a tool to drag and drop elements of SAP screens around, and then you can write code (JavaScript) to add whatever functionality you feel like (e.g., changing a three-screen transaction into one screen). Using SAP Screen Personas used to cost you money, but it’s now as free as GuiXT has always been.

13.8 Summary

This chapter introduced you to the latest player in SAP’s UI strategy, SAPUI5, and showed you how to go about building applications with it. This concludes the book’s coverage of UI technologies; the last part of the book will focus on the database aspect of ABAP programming—ending with everyone’s favorite lady, SAP HANA.

Recommended Reading

- Getting Started with SAPUI5: http://scn.sap.com/docs/DOC-48897
- End-to-End How-To Guide: Building SAPUI5 Applications on SAP NetWeaver AS ABAP 7.31 Consuming Gateway OData Services: http://scn.sap.com/docs/DOC-33792 (Betram Ganz and Bernhard Siewert)
- JavaScript for ABAP Developers: http://scn.sap.com/community/abap/blog/2014/02/18/javascript-for-abap-developers (Chris Whealy)
- Getting Started with SAPUI5 (Antolovic, SAP PRESS, 2014)
# Contents

Foreword ......................................................................................................... 19  
Acknowledgments ............................................................................................ 21  
Introduction ..................................................................................................... 23  

## PART I Programming Tools

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ABAP in Eclipse</td>
<td>35</td>
</tr>
<tr>
<td>1.1</td>
<td>Installation</td>
<td>37</td>
</tr>
<tr>
<td>1.1.1</td>
<td>Installing Eclipse</td>
<td>37</td>
</tr>
<tr>
<td>1.1.2</td>
<td>Installing the SAP-Specific Add-Ons</td>
<td>39</td>
</tr>
<tr>
<td>1.1.3</td>
<td>Connecting Eclipse to a Backend SAP System</td>
<td>41</td>
</tr>
<tr>
<td>1.2</td>
<td>Features</td>
<td>42</td>
</tr>
<tr>
<td>1.2.1</td>
<td>Working on Multiple Objects at the Same Time</td>
<td>47</td>
</tr>
<tr>
<td>1.2.2</td>
<td>Bookmarking</td>
<td>48</td>
</tr>
<tr>
<td>1.2.3</td>
<td>Creating a Method from the Calling Code</td>
<td>50</td>
</tr>
<tr>
<td>1.2.4</td>
<td>Extracting a Method</td>
<td>53</td>
</tr>
<tr>
<td>1.2.5</td>
<td>Deleting Unused Variables</td>
<td>58</td>
</tr>
<tr>
<td>1.2.6</td>
<td>Creating Instance Attributes and Method Parameters</td>
<td>59</td>
</tr>
<tr>
<td>1.2.7</td>
<td>Creating Class Constructors</td>
<td>60</td>
</tr>
<tr>
<td>1.2.8</td>
<td>Getting New IDE Features</td>
<td>61</td>
</tr>
<tr>
<td>1.3</td>
<td>Testing and Troubleshooting</td>
<td>63</td>
</tr>
<tr>
<td>1.3.1</td>
<td>Unit Testing Code Coverage</td>
<td>63</td>
</tr>
<tr>
<td>1.3.2</td>
<td>Debugging</td>
<td>67</td>
</tr>
<tr>
<td>1.3.3</td>
<td>Runtime Analysis</td>
<td>69</td>
</tr>
<tr>
<td>1.4</td>
<td>Customization Options with User-Defined Plug-Ins</td>
<td>71</td>
</tr>
<tr>
<td>1.4.1</td>
<td>UMAP</td>
<td>73</td>
</tr>
<tr>
<td>1.4.2</td>
<td>Obeo</td>
<td>78</td>
</tr>
<tr>
<td>1.5</td>
<td>Summary</td>
<td>79</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>New Language Features in ABAP 7.4</td>
<td>81</td>
</tr>
<tr>
<td>2.1</td>
<td>Database Access</td>
<td>82</td>
</tr>
<tr>
<td>2.1.1</td>
<td>New Commands in OpenSQL</td>
<td>82</td>
</tr>
<tr>
<td>2.1.2</td>
<td>Buffering Improvements</td>
<td>85</td>
</tr>
<tr>
<td>2.1.3</td>
<td>Creating while Reading</td>
<td>87</td>
</tr>
<tr>
<td>2.1.4</td>
<td>Inner Join Improvements</td>
<td>88</td>
</tr>
</tbody>
</table>
2.2 Declaring and Creating Variables ................................................... 90
  2.2.1 Omitting the Declaration of TYPE POOL Statements ............ 90
  2.2.2 Omitting Data Type Declarations ....................................... 92
  2.2.3 Creating Objects Using NEW ............................................ 92
  2.2.4 Filling Structures and Internal Tables while Creating Them Using VALUE ............................................ 93
  2.2.5 Filling Internal Tables from Other Tables Using FOR .......... 94
  2.2.6 Creating Short-Lived Variables Using LET ........................ 95

2.3 String Processing ................................................................. 96
  2.3.1 New String Features in Release 7.02 ................................. 96
  2.3.2 New String Features in Release 7.4 ................................. 97

2.4 Calling Functions ........................................................................ 98
  2.4.1 Method Chaining ............................................................. 98
  2.4.2 Avoiding Type Mismatch Dumps when Calling Functions .......... 99
  2.4.3 Using Constructor Operators to Convert Strings ............. 100
  2.4.4 Functions That Expect TYPE REF TO DATA .................. 101

2.5 Conditional Logic ...................................................................... 102
  2.5.1 Using Functional Methods in Logical Expressions .......... 103
  2.5.2 Omitting ABAP_TRUE ..................................................... 103
  2.5.3 Using XSDBOOL as a Workaround for BOOLC .......... 105
  2.5.4 The SWITCH Statement as a Replacement for CASE ........ 106
  2.5.5 The COND Statement as a Replacement for IF/ELSE ....... 108

2.6 Internal Tables .......................................................................... 109
  2.6.1 Using Secondary Keys to Access the Same Internal Table in Different Ways ............................................ 109
  2.6.2 Table Work Areas .......................................................... 112
  2.6.3 Reading from a Table ..................................................... 114
  2.6.4 CORRESPONDING for Normal Internal Tables ............. 115
  2.6.5 MOVE-CORRESPONDING for Internal Tables with Deep Structures ............................................ 116
  2.6.6 New Functions for Common Internal Table Tasks .......... 120
  2.6.7 Internal Table Queries with REDUCE ......................... 122
  2.6.8 Grouping Internal Tables ............................................. 122
  2.6.9 Extracting One Table from Another ............................. 124

2.7 Object-Oriented Programming ............................................... 126
  2.7.1 Upcasting/Downcasting with CAST .............................. 126
  2.7.2 CHANGING and EXPORTING Parameters ................... 127
  2.7.3 Changes to Interfaces .................................................. 128

2.8 Search Helps ............................................................................ 129
  2.8.1 Predictive Search Helps ................................................. 129
  2.8.2 Search Help in SE80 ..................................................... 130

2.9 Unit Testing ............................................................................. 200
  2.9.1 Creating Test Doubles Relating to Interfaces ................. 200
  2.9.2 Coding Return Values from Test Doubles .................... 200
  2.9.3 Creating Test Doubles Related to Complex Objects ....... 201

3 ABAP Unit and Test-Driven Development .................................. 137
  3.1 Eliminating Dependencies .................................................... 139
    3.1.1 Identifying Dependencies ........................................... 140
    3.1.2 Breaking Up Dependencies ......................................... 141
  3.2 Implementing Mock Objects .................................................. 143
    3.2.1 Creating Mock Objects ................................................ 144
    3.2.2 Injection .................................................................. 145
  3.3 Writing and Implementing Unit Tests .................................... 147
    3.3.1 Defining Test Classes .................................................. 148
    3.3.2 Implementing Test Classes ......................................... 148
    3.3.3 Creating Test Doubles Related to Complex Objects ....... 150

3.4 Automating the Test Process .................................................... 161
  3.4.1 Automating Dependency Injection ................................. 162
  3.4.2 Automating Mock Object Creation via mockA .................. 167
  3.4.3 Combining Dependency Injection and mockA ............... 170

3.5 Behavior-Driven Development .................................................. 171

3.6 Summary .................................................................................. 173

4 ABAP Test Cockpit ................................................................. 175
  4.1 Automatic Run of Unit Tests ................................................. 177
  4.2 Mass Checks ........................................................................ 178
    4.2.1 Setting Up Mass Checks ............................................ 180
    4.2.2 Running Mass Checks ................................................ 182
    4.2.3 Reviewing Mass Checks ............................................. 185
    4.2.4 Dismissing False Errors .............................................. 188
  4.3 Recent Code Inspector Enhancements ................................. 192
    4.3.1 Unsecure FOR ALL ENTRIES (12/5/2) ...................... 193
    4.3.2 SELECT * Analysis (14/9/2) ........................................ 195
    4.3.3 Improving FOR ALL ENTRIES (14/9/2) ................. 196
    4.3.4 SELECT with DELETE (14/9/2) ............................. 198
    4.3.5 Check on Statements Following a SELECT without ORDER BY (14/9/3) ............................................ 199
    4.3.6 SELECTs in Loops across Different Routines (14/9/3) .... 200

4.4 Summary .................................................................................. 202
PART II Business Logic Layer

7 Exception Classes and Design by Contract ........................................... 255

7.1 Types of Exception Classes .......................................................... 257
7.1.1 Static Check (Local or Nearby Handling) ............................. 258
7.1.2 Dynamic Check (Local or Nearby Handling) ................. 260
7.1.3 No Check (Remote Handling) ........................................ 260
7.1.4 Deciding which Type of Exception Class to Use .............. 262

7.2 Designing Exception Classes ..................................................... 263
7.2.1 Creating the Exception ..................................................... 264
7.2.2 Declaring the Exception .................................................. 266
7.2.3 Raising the Exception ..................................................... 267
7.2.4 Cleaning Up after the Exception Is Raised ...................... 268
7.2.5 Error Handling with RETRY and RESUME ................. 270

7.3 Design by Contract ................................................................... 274
7.3.1 Preconditions and Postconditions ................................... 276

7.3.2 Class Invariants .................................................................... 278

7.4 Summary .................................................................................. 281

8 Business Object Processing Framework (BOPF) .................................. 283

8.1 Defining a Business Object ........................................................... 284
8.1.1 Creating the Object .......................................................... 285
8.1.2 Creating a Header Node .................................................. 286
8.1.3 Creating an Item Node ..................................................... 288

8.2 Using BOPF to Write a DYNPRO-Style Program ......................... 290
8.2.1 Creating Model Classes .................................................. 291
8.2.2 Creating or Changing Objects ......................................... 294
8.2.3 Locking Objects ............................................................... 304
8.2.4 Performing Authority Checks ........................................... 305
8.2.5 Setting Display Text Using Determinations ....................... 306
8.2.6 Disabling Certain Commands Using Validations ............... 316
8.2.7 Checking Data Integrity Using Validations ......................... 318
8.2.8 Responding to User Input via Actions ............................... 325
8.2.9 Saving to the Database .................................................... 335
8.2.10 Tracking Changes in BOPF Objects ................................. 342

8.3 Custom Enhancements ................................................................ 350
8.3.1 Enhancing Standard SAP Objects .................................... 351
8.3.2 Using a Custom Interface (Wrapper) ............................... 354

8.4 Summary .................................................................................. 355

9 BRFplus .................................................................................... 357

9.1 The Historic Location of Rules .................................................. 360
9.1.1 Rules in People’s Heads .................................................... 360
9.1.2 Rules in Customizing Tables .......................................... 362
9.1.3 Rules in ABAP ............................................................... 364

9.2 Creating Rules in BRFplus: Basic Example ............................... 365
9.2.1 Creating a BRFplus Application ...................................... 365
9.2.2 Adding Rule Logic .......................................................... 373
9.2.3 BRFplus Rules in ABAP .................................................. 386

9.3 Creating Rules in BRFplus: Complicated Example ..................... 388
9.4 Simulations ............................................................................. 394
9.5 SAP Business Workflow Integration ......................................... 397
9.6 Options for Enhancements ....................................................... 401
9.6.1 Procedure Expressions ..................................................... 401
9.6.2 Application Exits ............................................................. 402
PART IV Database Layer

14 Shared Memory ............................................................................. 631
  14.1 The Promises of Shared Memory ............................................ 632
    14.1.1 Database Access ............................................................ 633
    14.1.2 Memory Usage .............................................................. 633
  14.2 Creating and Using Shared Memory Objects ......................... 634
    14.2.1 Creating the Root Class ................................................. 635
    14.2.2 Generating the Broker Class ......................................... 640
    14.2.3 Using Shared Memory Objects in ABAP Programs .......... 643
  14.3 Updating the Database and Shared Memory Together ............ 647
  14.4 Troubleshooting ................................................................... 649
    14.4.1 Data Inconsistency between Application Servers ............ 649
    14.4.2 Short Dumps ................................................................. 652
  14.5 Summary ............................................................................... 654

15 ABAP Programming for SAP HANA ............................................. 655
  15.1 Introduction to Code Pushdown .............................................. 657
  15.2 Top-Down Development ....................................................... 658
    15.2.1 Building and Calling CDS Views .................................... 659
    15.2.2 Building and Calling ABAP Managed Database Procedures . 671

16 Conclusion .................................................................................. 703

Appendices ...................................................................................... 705
  A Improving Code Readability ................................................... 707
    A.1 Readability vs. Concision .................................................. 707
    A.2 The What vs. the How ...................................................... 708
  B Making Programs Flexible ......................................................... 711
  C The Author .............................................................................. 719

Index ................................................................................................ 721
## Index

### A

- ABAP 7.02, 90, 96, 98, 103, 109
- ABAP 7.4, 81, 97, 102, 114, 127, 131, 365, 675, 707
- new features, 81
- recommended reading, 136
- ABAP Managed Database Procedures (-> AMDP)
- ABAP Messaging Channels, 135
- ABAP Push Channels, 135
- ABAP Test Cockpit, 175–176
- and SAP HANA, 176
- recommended reading, 202
- ABAP Unit, 137
- ABAP Workbench, 35, 53, 359
- ABAP_TRUE, 103
- ABAP2XLSX, 457, 465
- conditional formatting, 474
- download, 461
- email, 493
- enhancement framework, 501
- enhancing custom reports, 466
- example programs, 465
- hyperlinks, 496, 498
- macros, 487
- multiple worksheets, 482
- printer settings, 471
- recommended reading, 505
- templates, 491
- testing, 478
- Accleo, 76
- AFTER_FAILED_EVENT, 551
- ALV, 466
- ALV grid, 515
- ALV SALV, 407
- AMDP, 659, 671–672
- Eclipse, 673
- Antifragile programs, 65, 541, 711
- ASSERT, 158, 276

### B

- BAdIs, 225–226, 235
- calling, 249
- defining, 235, 237
- filter checks, 240
- filters, 239
- implementing, 245
- interface, 244
- Behavior-driven development, 171
- BOOLC, 105
- Boolean logic, 105
- BOPF, 283–284, 541, 570
- action validations, 331
- actions, 325, 327
- and FPM, 556
- authority checks, 305
- callback subclass, 347
- change document submode, 346
- create header node, 286
- create item node, 288
- create model classes, 291
- create object, 285
- creating/changing objects, 294
- CRUD, 336
- custom enhancements, 350
- custom queries, 295–296
- delegated objects, 344
- determinations, 306
- locking objects, 304
- recommended reading, 356
- testing, 349
- tracking changes, 342
- validations, 316, 318
- wrappers, 354
- BOR object, 569
- Breakpoints, 207
- BRFplus, 357, 365
- call in ABAP, 386
- create application, 365
- decision tables, 378
- decision trees, 374, 377
BRFplus (Cont.)
  enhancements, 401
  example, 388
  recommended reading, 404
  rule logic, 373
SAP Business Workflow, 397
  simulations, 394
Broker class, 635
Buffering, 85
Business Object Processing Framework
  (BOPF)
  Business rules, 357, 360
  ABAP, 364
  BRFplus, 365
  main object model, 362
  Business rules framework, 357
CASE, 83, 102, 106, 665
CDS view
  buffering, 663
  built-in functions, 667
CDS views, 659
CE functions, 674
CHANGING parameters, 127
CHECK, 309, 313, 321
CHECK_DELTA, 309, 311, 321
CL_SALV_TABLE, 87, 409, 411
Class under test, 161
CLEANUP, 269
CMOD framework, 226
Code Inspector, 175, 200
  new features, 192
  pushdown, 682
  AMDP, 688
  CDS views, 687
  locating code, 684
  OpenSQL, 687
  techniques, 684
Complex objects, 133
COMPONENTCONTROLLER, 545
Concision, 707
COND, 108
Conditional logic, 102
Constructor operators, 100
CORRESPONDING, 115, 707
CORS (Cross Origin Resource Sharing), 584
Cross-program communication, 134
CRUD, 296, 336, 582
CX_DYNAMIC_CHECK, 260
CX_STATIC_CHECK, 258

D
Data definitions, 150
Data type declarations, 92
Database access, 82, 633
DDIC table, 572
DDL, 661, 666, 669
coding, 662
Debugging, 203
recommended reading, 223
Script tab, 204
Script Wizard, 209, 213
skipping authority checks, 222
tracing every line, 222
Decision logic, 357
Delegated objects, 344
Dependencies, 138
breaking up, 141
eliminating, 139
identifying, 140
Design by contract, 255, 274
class invariants, 278
postconditions, 276
preconditions, 276
DISPLAY method, 412
Downcast, 126
DYNPRO, 507

E
Early Watch reports, 175
Eclipse, 35–36
  AMDP, 673
  and SAP HANA, 661
  and SAPUI5, 592
  bookmarking, 48
  class constructors, 60
Eclipse (Cont.)
  connect to backend system, 41
  create attributes, 59
  create parameters, 59
debugging, 67
effect of method, 53
Extraction Wizard, 57
  features, 42, 61
  installation, 37
Modeling Framework, 40
  multiple objects, 47
  plugins, 71
  prerequisites, 37
  Quick Assist, 53
  recommended reading, 79
  refactoring, 59
  release cycle, 35, 47
  runtime analysis, 69
  SAP add-ons, 39
  SDK, 71
  unit tests, 63
  unused variables, 58
  Ellison, Larry, 655
  END, 206, 215
Enhancement framework, 225
recommended reading, 252
Enhancement spot, 227

F
FACTORY method, 411
FILTER, 124
FitNesse, 172
Floorplan Manager (FPM)
  FLUSH, 551
  FOT, 94
  FOR ALL ENTRIES, 193, 196
  FPM, 507, 546
  floorplans, 547
  FPM Workbench, 548
  GUIBBs, 549
  Guided Activity Floorplan, 547
  modify WDA components, 546
  Object Instance Floorplan, 547
  Overview Floorplan, 547
  Quick Activity Floorplan, 548
  recommended reading, 559
  UIBBs, 549
  Functional methods, 103

G
GET_ENTITY_SET, 583, 588–589
GitHub, 167
GUID, 286, 302
GuiXT, 626

H
HASHED table, 112
Helper methods, 152
Index

I
ICF, 497
IF/ELSE, 108
IF/THEN, 102
INIT, 206, 215
INITIALIZE, 423, 444
Injection, 145
automation, 162
constructor, 147
Inner joins, 88
Internal tables, 109, 302
grouping, 122
new functions, 120

J
JavaScript, 564

L
LET, 95

M
Mass checks, 178
reviewing, 185
running, 182
setup, 180
Memory, 633
Method chaining, 98, 169
Microsoft Excel (Excel), 461
Microsoft Open XML, 461
MIME repository, 490
Mock objects, 138, 143, 161
Model classes, 283
Model-view-controller (MVC pattern), 288
Model-view-controller (MVC pattern) → ADT, 659
MOVE-CORRESPONDING, 116, 708
MVC pattern, 284, 290, 409–410, 508, 557, 564
caller, 514
location of model, 509
model, 508
view, 511

N
NEEDS_CONFIRMATION, 551
NEW, 92

O
Obeo, 73, 78
object_configuration, 293
Object-oriented programming, 50, 126, 144, 233, 283, 354, 409, 533, 711
OData, 565
Open source, 458, 616
OpenSQL, 82, 658, 666
new commands, 82
OperUI5, 615
open source, 616
ORDER BY, 199

P
Persistency layer, 634
Plattner, Hasso, 631
PREPARE, 327
Private methods, 148
Procedural programming, 138
PROCESS BEFORE OUTPUT, 551
PROLOGUE, 206
READ TABLE, 114
Readability, 707
REDUCE, 122
Report programming, 407
REST, 565
RETrieve DEFAULT PARAM, 327
Return values, 152
Root class, 635
RS_AUCV_RUNNER, 178
Rules engines, 358

S
SALV, 408
add custom icons, 441
SALV (Cont.)
application-specific changes, 424
CL_SALV_GUI_TABLE_IDA, 454
create container, 416, 442
design report interface, 414
display the report, 435
data editing, 447
event handling, 421
initialize report, 416
recommended reading, 456
SAP HANA, 456
SAP Code Exchange, 167
SAP Decision Service Management, 359
SAP Fiori, 626
SAP Gateway, 561, 564
coding, 581
create model, 567
create service, 574
data provider class, 582
error handling, 591
model provider class, 582
service builder, 568
service implementation, 582
testing, 579
SAP GUI, 407
SAP HANA, 176, 456, 588, 631, 655
ABAP table design, 691
AMDP, 659, 671
and Eclipse, 661
artifacts, 658
bottom-up development, 677
CDS views, 659
CE functions, 674
code pushdown, 657, 681
database proxies, 679
database views, 657
database-specific features, 695
DDL, 661
e external views, 678
programming changes, 691
recommended reading, 701
redundant storage, 691
secondary indexes, 693
SAP HANA (Cont.)
SELECT coding, 697
top-down development, 658
transporting changes, 680
SAP NetWeaver Development Tools for ABAP (ADT) → Eclipse
SAP Screen Personas, 626
SAPUI5, 508, 561
and Eclipse, 566, 592
architecture, 563
browser support, 580
buttons, 611
controller, 608
fragment XML file, 603
functions, 609–610
HTML file, 596
importing applications, 620
JavaScript, 564
prerequisites, 565
recommended reading, 627
storing applications, 622
testing, 613, 624
view, 596
view and controller, 592
XML file, 597
SCRIPT, 206, 209, 215
Search helps, 129
predictive, 129
SELECT, 199
SELECT *, 195
Separation of concerns, 142
service_manager, 293
SETUP, 151
Shared memory, 631
broker class, 640
business transaction event, 651
data inconsistency, 649
objects, 634
read request, 643
recommended reading, 654
remote-enabled function module, 650
root class, 635
short dumps, 652
updating database, 647
write request, 645
Short dumps, 99–100, 632, 652
SORTED table, 112
We hope you have enjoyed this reading sample. You may recommend or pass it on to others, but only in its entirety, including all pages. This reading sample and all its parts are protected by copyright law. All usage and exploitation rights are reserved by the author and the publisher.

© 2015 by Rheinwerk Publishing, Inc. This reading sample may be distributed free of charge. In no way must the file be altered, or individual pages be removed. The use for any commercial purpose other than promoting the book is strictly prohibited.