SAP
C_FIORD
SAP Certified - SAP Fiori Application Developer

Latest Version: 4.1

1. Unified Scenario Simulation
2. Micro Skill Drill Exam

Topic: 1
Unified Scenario Simulation
Levant Bakehouse Multi-Site SAPUI5 App Release Readiness
Business Context
Levant Bakehouse Group is an industrial bakery in Beirut, Lebanon, producing packaged breads and pastries for supermarkets across the country. To modernise how production supervisors manage daily output, the company built a freestyle SAPUI5 application that runs on the SAP Fiori launchpad. Supervisors open a worklist of production orders and drill into one order's detail view to review quantities and record progress. The workforce is mixed: the Beirut plant runs largely in English, while teams at two other sites work primarily in Arabic and French.
System Landscape
The application reads its production orders over an OData model and is organised into XML views, controllers, and a single component. It began as a one-site pilot, so several button and column labels were written directly into the views as English text during the first build. Supervisors open the app both on desktop workstations in the planning office and on handheld tablets carried across the shop floor. Today the app renders one fixed layout regardless of the device on which it is opened.
Implementation Progress
Management now wants to roll the app out from the single Beirut pilot to the two additional sites and has set a short release window. A small team of three developers maintains the codebase together. What worked comfortably for one English-speaking plant is now being asked to serve three sites, several languages, and two very different devices at once, and the team is preparing the first multi-site release.
Language Preview
During a preview at the second site, supervisors noticed that the button and column labels still appeared in English even though their launchpad and profile language were set differently. A developer checking the app found no resource model wired in; the translatable texts are the ones typed straight into the views. Each new label added since the pilot has followed the same pattern, so the wording is scattered across many view files.
Device Feedback
On the shop-floor tablets, supervisors reported that the lists and forms feel cramped, with controls sized for a mouse and tap targets that are hard to hit while wearing gloves. The same screens look perfectly comfortable on the office desktops. Because the app produces one layout for every device, the tablet users and the desktop users are served the identical sizing.
Release Preparation
The team wants to cut the multi-site release from a shared working copy, but all three developers currently commit to a single main line and there are no automated checks. Each change is eyeballed once and shipped. A recent edit to the worklist quietly broke the navigation into the order-detail screen, and nobody noticed until a supervisor reported that opening an order did nothing. The lead now wants confidence that a change to one screen has not broken another before the release reaches any site.
Validation Outlook
Before the rollout each area will be checked in turn: whether the labels follow each supervisor's working language, whether the same screens are usable on both the office desktop and the shop-floor tablet, and whether an automated check runs before code is merged. The team's task is to choose approaches

that keep the app a single, maintainable code base across three sites rather than quick fixes that solve one screen while multiplying the work behind the others.

Your assignment

You are a SAPUI5 developer supporting Levant Bakehouse Group as it prepares the first multi-site release of its production-order application. The app runs on the SAP Fiori launchpad, reads over an OData model, and is used on both office desktops and shop-floor tablets by teams working in different languages. Management wants the app to serve all three sites from one maintainable code base, released through a process that catches a regression before supervisors see it. Advise the team on the soundest approach for each area below, respecting standard SAPUI5 practice and the short release window

1. CHALLENGE 1 — Presenting the App in Each Site Team's Working Language

• Recommend how the app should present its labels so each supervisor sees them in their own working language, given that texts were typed straight into the views during the pilot.

• Advise how to structure the texts so that adding a further language later is low-effort and does not require changing the views.

• Position any quick pre-preview translation as a stopgap and explain what still has to happen for the actual multi-site release.

Checkpoints

• Labels follow the supervisor's launchpad and profile language across every screen, not only the busiest ones.

• Adding a language is a contained change rather than an edit spread across many view files.

• The chosen approach keeps a single app for all sites instead of parallel per-language copies.

2. CHALLENGE 2 — Making Order Screens Usable on Shop-Floor Tablets and Desktops

• Recommend how the same screens can be comfortable on both the office desktop and the shop-floor tablet from one code base.

• Weigh a contractor's offer to detect the tablet in code and hard-code larger fixed sizes against the standard device-adaptation approach.

• Advise how the team should confirm, before release, that the screens are usable on both device types.

Checkpoints

• The desktop experience is preserved while the tablet becomes comfortable for touch.

• The solution avoids a separate per-device version or fragile size-branch logic that must be maintained apart.

• Usability is confirmed by using the app on both a real desktop and a real shop-floor tablet.

Question: 1

CHALLENGE 1 — Presenting the App in Each Site Team's Working Language

During the second-site preview, supervisors saw English labels although their profile language differed, and a developer confirmed no resource model is wired in.

What is the most appropriate way to make the app show each supervisor the labels in their own working language?

A. Add a language switch button on each screen that lets the supervisor re-label the controls themselves at runtime, and confirm it works by toggling the labels during the site preview.

B. Detect the browser language in the controller and swap each hard-coded label with an equivalent string in code, verifying that the office desktop then shows the expected words.

C. Move the translatable texts out of the views into a resource model, bind each control's text to it, and confirm the labels follow the launchpad language during the site preview.
D. Keep the texts in the views for now and give the site teams a printed glossary of terms, then revisit translation after the multi-site release has stabilised.

Answer: C

Explanation:
A resource model holds the translatable texts separately and serves each control the string for the user's language, so labels follow the launchpad and profile language across every site without touching the views again. It is the standard SAPUI5 localization approach and is confirmed when the preview shows the site's language.

Question: 2

CHALLENGE 1 — Presenting the App in Each Site Team's Working Language

The texts are currently inline across many view files, and the team wants adding a third language later to stay low-effort.
Which approach best supports adding a further language with the least rework?

A. Keep all translatable texts in per-language resource files that the resource model selects, so adding a language means adding one more file without changing the views.
B. Store the texts in a custom table read over the OData model at application startup and merge them into the views, revalidating the service each time a language is added later.
C. Duplicate each XML view once per language and let routing open the matching copy for the user, checking that the right copy appears for each site during preview.
D. Concatenate the translated words inside formatter functions on each control so the formatter returns the correct language for the user when the view renders.

Answer: A

Explanation:
Per-language resource files behind the resource model are the standard, upgrade-safe place for translatable texts. Adding a language is one new file with no view changes, which keeps the growing multi-language requirement low-effort and consistent across sites.

Question: 3

CHALLENGE 1 — Presenting the App in Each Site Team's Working Language

There is pressure to show the second site something quickly, and a developer offers to hand-translate only the two busiest screens before the preview.
What is the best response to the offer to hand-translate only the two busiest screens?

A. Accept it as the release approach, since the busiest screens carry most of the supervisors' daily work and the remaining screens can simply stay in English permanently.
B. Accept it but copy the translated screens into a second app built for that site, so the original app is left completely untouched for the other two sites.
C. Reject it and postpone the whole release until every screen in every planned language has been fully translated and formally signed off by each site.
D. Treat it only as a preview stopgap and still move the texts into the resource model, so all screens follow the user's language for the actual release.

Answer: D

Explanation:
Hand-translating two screens can unblock a preview, but only the resource model makes every screen follow each user's language for the real release. Keeping the stopgap while still doing the proper move avoids shipping a half-translated app.

Question: 4

CHALLENGE 2 — Making Order Screens Usable on Shop-Floor Tablets and Desktops

The shop-floor tablets feel cramped while the office desktops are comfortable, and the app currently produces one fixed layout for every device.
What is the most appropriate way to make the same screens comfortable on both the office desktop and the shop-floor tablet?

A. Build a separate tablet-only version of each screen and route tablet users to it, verifying the behaviour by opening the app on a tablet on the plant floor at each site.
B. Let the app apply the content density and layout suited to the device it runs on, so one code base adapts for desktop and tablet, confirmed by using both devices.
C. Ask supervisors to zoom the browser on the tablets until the controls are large enough, and note the zoom level that each site should use for its devices.
D. Increase every control's size globally so tap targets are large everywhere, and accept that the office desktop will then show the same larger sizing as well.

Answer: B

Explanation:
SAPUI5 can apply the appropriate content density and adapt the layout to the device type, so a single code base is comfortable with a mouse on the desktop and with touch on the tablet. It is confirmed by using the app on both devices.

Question: 5

CHALLENGE 2 — Making Order Screens Usable on Shop-Floor Tablets and Desktops

A contractor offers to detect the tablet in controller code and hard-code larger fixed sizes on the shop-floor screens so the release can ship quickly.
How should the team weigh the contractor's hard-coded per-device size offer?

A. Adopt it, because it ships fastest and the hard-coded sizes give supervisors large tap targets immediately on the shop-floor screens.
B. Adopt it for now and plan to remove the hard-coded sizes in a later release once time allows, tracking the temporary branch of code as a known item to clean up.
C. Reject it and instead fix the tablet feel by reducing the number of columns shown on every device, including on the office desktop that already works well.
D. Prefer the device-adaptation and density mechanism, since a hard-coded size branch bypasses the standard approach and becomes fragile code to maintain per device.

Answer: D

Explanation:
The device-adaptation and density mechanism is the standard, maintainable way to fit each device, so the app stays one code base. A hard-coded size branch keyed on device detection is extra fragile logic that must be revisited whenever devices or screens change.
Topic: 2
Micro Skill Drill Exam

Question: 6

Stauffen Automotive Systems, a components supplier in Stuttgart, Germany, has two SAPUI5 apps on its SAP Fiori launchpad: a supplier-scorecard app and a separate purchase-order app. From the scorecard, buyers need a link that opens the purchase-order app already showing orders for the selected supplier. A developer implemented the link by hard-coding the other app's current URL into the scorecard, and it worked in the developer's system. After the purchase-order app was reassigned to a different launchpad configuration, the link began opening a page that no longer existed. The team follows SAP Fiori guidelines and wants cross-app navigation that stays valid when target apps are re-catalogued, rather than a fixed address embedded in the source app.
Which navigation approach aligns with SAP Fiori guidelines for opening the other launchpad app?

A. Use the launchpad's cross-application navigation service with an intent, so the framework resolves the target app and the link survives re-cataloguing.
B. Open the target app in a new browser tab from a stored absolute link, since launching it separately avoids depending on the launchpad to route the request.
C. Keep the hard-coded target URL but update it in the scorecard code whenever the other app is re-catalogued, since correcting the address restores the broken link.
D. Duplicate the purchase-order view inside the scorecard app instead of navigating out, since embedding a copy removes the need to reference the external app at all.

Answer: A

Explanation:

The launchpad's cross-application navigation service resolves a target by intent rather than a fixed URL, so the framework locates the current app and passes the supplier context. The link keeps working when the target is re-catalogued, which is the guideline-aligned behavior required.

Question: 7

Fjordline Maritime Logistics, a vessel-scheduling operator based in Oslo, Norway, is starting a new freestyle SAPUI5 application that dispatchers open from the SAP Fiori launchpad to plan port calls. The frontend team wants a maintainable structure: the framework should load first, and then a single declarative entry point should own the application's configuration, routing metadata, and model setup, so that new views can be added later without rewiring startup logic. A junior developer has instead placed all initialization in an inline script on the shell HTML page — creating the root view, registering routes, and instantiating models line by line — and the lead reviewer has flagged that this will not scale as more object pages are added. The team asks how the initialization should be organized so that one component, rather than page-level script, owns the application lifecycle.
Which approach best structures the application's initialization?

A. Move every view and model creation into one startup controller, since keeping all initialization inside a single controller file centralizes the logic clearly.
B. Define an application Component as the single entry point and configure its routing and models declaratively in the app descriptor, letting the framework own startup.
C. Distribute the configuration across several on-demand modules with no central owner, since loading each piece lazily at runtime removes the need for a formal entry point.
D. Register all routes and instantiate the root view from the inline bootstrap script on the shell page, since a small app can start without defining an additional component artifact.

Answer: B

Explanation:
Defining an application Component gives the app one declarative entry point: the descriptor owns routing and model configuration, and the framework instantiates and manages the lifecycle. New views are then added through configuration rather than by editing startup script, which is the scalable structure the team wants.

Question: 8

Kyomi Precision Instruments, a measurement-device manufacturer in Kyoto, Japan, runs a SAPUI5 quality-inspection app on the SAP Fiori launchpad. Inspectors open a worklist and then navigate to the detail page of one specific inspection lot. The URL must be bookmarkable so a supervisor can reopen the exact lot later, and pressing browser Back must return to the worklist. A developer configured the detail route with a fixed pattern that always points to the same lot, so every navigation and every bookmark resolves to that one record instead of the lot the inspector selected. The team needs the route to carry the selected lot's identifier so navigation targets the correct object and deep links reopen it reliably.
How should the detail route be configured so navigation and bookmarks resolve to the selected inspection lot?

A. Define the detail route with a mandatory parameter for the lot identifier and pass the selected id when navigating, so each target and bookmark resolves to that lot.
B. Remove routing from the detail view and swap its bound context in code on each selection, since setting the model context directly will still update what the page shows.
C. Point every inspector to one shared hard-coded pattern and filter the worklist afterwards, since narrowing the list first makes the single detail route sufficient.
D. Keep the fixed route pattern and instead reload the whole application whenever a different lot is chosen, since a full reload will refresh the page to the new record.

Answer: A

Explanation:
A route with a mandatory parameter binds the selected lot's identifier into the hash, so navigation targets the chosen object and a bookmarked URL re-resolves to the same lot. Back returns to the worklist because each state is a distinct routed entry, which meets the bookmarkable, correct-target requirement.

Question: 9

Karoo Retail Group, a grocery chain headquartered in Cape Town, South Africa, needs a new application for merchandisers to browse purchasing documents and open any one to see its details. The requirements match a standard pattern: a filterable list of documents and a details page per document, with SAP's standard Fiori look, responsive behavior, and no unusual custom interactions. The backend already exposes an OData service, and the team wants to minimize hand-written UI code and long-term maintenance. One developer proposes building the entire UI as a freestyle SAPUI5 app with custom views and controllers for the list and the detail page. The lead asks whether a more standardized, metadata-driven approach would deliver the same result with far less custom JavaScript.
Which development approach best fits this standard list-and-detail requirement?

A. Use SAP Fiori elements List Report and Object Page templates driven by annotations, so the standard UI is generated from metadata with minimal custom code.
B. Assemble the pages from individual layout controls in one large view, since composing the screen manually still reproduces the standard list-and-detail layout.
C. Build freestyle views and controllers for the list and detail pages by hand, since writing the UI directly gives the team the most direct control over every element.
D. Generate the app but replace the templates with custom controls right away, since starting from a template and overriding it everywhere keeps full design freedom.

Answer: A

Explanation:
SAP Fiori elements List Report and Object Page templates render a filterable list and a detail page from the OData service and its annotations, producing SAP's standard, responsive UI with minimal custom JavaScript. That is the fastest, most maintainable fit for a standard pattern.

Question: 10

Cooperativa AgroVerde, a grain-trading cooperative in Curitiba, Brazil, is extending a SAPUI5 application that lets field agents review delivery contracts. A developer must display a table that lists every harvest line item belonging to a single selected contract. The contract header is already shown on the page, and the OData model exposes the line items as a collection associated with that contract. The developer needs the table's rows to repeat automatically, one row per line item, and to refresh whenever a different contract is opened. In an early attempt the developer bound the table to a single line-item entry, and only one row appeared regardless of how many items the contract held. The team wants the row set driven by the collection so the table population stays declarative rather than built in controller code.
Which binding approach correctly populates the table rows from the contract's line-item collection?

A. Bind each line-item property with expression binding on the header area, since resolving the values inline will make the table grow to the full set.
B. Read the collection in the controller and construct each row in code on every selection, since building rows manually gives precise control over the contents.
C. Bind the table's items aggregation to the line-item collection, so the framework renders one row per entry and refreshes when the bound context changes.
D. Bind the table to a single line-item entity using element binding, since directing the control at one context is enough for its rows to render.

Answer: C

Explanation:
Binding the table's items aggregation to the line-item collection lets the framework repeat one row per entry and re-resolve the set when a new contract context is bound. That delivers the declarative, self-refreshing table the requirement calls for.

Thank You for Trying Our Product

For More Information – Visit link below:

https://www.examsboost.com/

15 USD Discount Coupon Code:

G74JA8UF

FEATURES

- ✓ 90 Days Free Updates
- ✓ Money Back Pass Guarantee
- ✓ Instant Download or Email Attachment
- ✓ 24/7 Live Chat Support
- ✓ PDF file could be used at any Platform
- ✓ 50,000 Happy Customer